

**WestminsterResearch**

<http://www.westminster.ac.uk/westminsterresearch>

**Design Methodology for Rich Web-based Applications**

**Dissanayake, N.R.**

This is a PhD thesis awarded by the University of Westminster.

© Mr Dissanayake Mudiyansele Dissanayake, 2024.

<https://doi.org/10.34737/ww439>

The WestminsterResearch online digital archive at the University of Westminster aims to make the research output of the University available to a wider audience. Copyright and Moral Rights remain with the authors and/or copyright owners.

---

**Design Methodology  
for  
Rich Web-based Applications**

---

**By**

**Dissanayake Mudiyansele Nalaka Ruwan Dissanayake**

**Doctor of Philosophy**

**July 2024**

**School of Computer Science and Engineering**

**UNIVERSITY OF  
WESTMINSTER** 

## Abstract

---

Rich web-based applications (RiWAs), like Facebook or Google apps, improve user experience over regular web applications with rich GUIs similar to desktop applications and the advanced delta-communication model (DC) to communicate faster with the server-side and update the current GUI without loading a new one. The RiWAs development tools, like libraries and frameworks, have evolved over the last two decades; however, conceptual artefacts like design patterns and design methods/methodologies have not evolved to cater to the RiWAs' specificity.

The Unified Modelling Language (UML) is the de facto standard General-Purpose Modelling Language (GPML). There are UML-based or UML-like designing languages available, like Arc42, SAP's TAM and OOA/OOM, ArchiMate, and SysML, where some, like the C4 model, UWE, IFML, and IAML are explicitly focusing on web applications. These available languages do not address modelling all the aspects of RiWAs and exclude features like distribution of the business logic and the *Application* elements to the tiers and platforms, details of the processing components such as controllers, and communication granularity, including DC-related processing.

This conceptual research is scoped for browser-based 3-tier RiWAs and focuses more on structural designing, aligning with the Rich Web-based Applications Architectural style (RiWAArch style). Real-world use cases demonstrate the use and adoption of the introduced design methodology. The design methodology is evaluated by triangulating the results of formal methods: a self-evaluation utilising the use cases as proof of concept, a contextualised comparison, and domain experts' evaluation.

The following are contributions of the research. A process for implementing a new design methodology is formulated, which can assist similar research. The study identifies the requirements for a *Domain-Specific Modelling Language* (DSML) for RiWAs and introduces a UML extension with new models, model-elements, and UML profiles. The new DSML introduces a new labelling format for the model-elements to include more details consistently to improve the designs' usability. Rules and guidelines for using the new language in RiWAs design and development are also provided. Then, this research contributes a design methodology that discusses RiWAs' design and engineering approaches based on the introduced DSML and also provides guidelines for integrating the RiWAs' design activities into an agile environment. The presented design methodology with the new DSML delivers a simple and adoptable solution (covering the aspects of comprehensiveness, learnability, readability/understandability, development support, and integrability) for the domain of RiWAs.

## Publications

---

### Publications Related to the Design Method/Methodology for RiWAs

1. **Nalaka R. Dissanayake** and Alexander Bolotov, "Applications Model: A High-level Design Model for Rich Web-based Applications," *19<sup>th</sup> International Conference on Evaluation of Novel Approaches to Software Engineering - ENASE 2024*, 2024, 28-29 April, 2024, Angers, France.

### Publication Related to the RiWAs and the RiWAArch Style

2. **Dissanayake N. R.**, Dias G.K.A. (2021) "RiWAArch Style: An Architectural Style for Rich Web-Based Applications". In: Arai K., Kapoor S., Bhatia R. (eds) *Proceedings of the Future Technologies Conference (FTC) 2020*, Volume 3. FTC 2020. *Advances in Intelligent Systems and Computing*, vol 1290. Springer, Cham. [https://doi.org/10.1007/978-3-030-63092-8\\_20](https://doi.org/10.1007/978-3-030-63092-8_20)
3. **Nalaka R. Dissanayake** and G. K. A. Dias, "Rich Web-based Applications: An Umbrella Term with a Definition and Taxonomies for Development Techniques and Technologies," *International Journal of Future Computer and Communication*, vol. 7, no. 1, pp. 14-20, 2018, <https://doi.org/10.18178/ijfcc.2018.7.1.513>
4. **Nalaka R. Dissanayake** and G. K. A. Dias, "Delta-Communication: The Power of the Rich Internet Applications", *International Journal of Future Computer and Communication*, vol. 6, No. 2, 2017, <https://doi.org/10.18178/ijfcc.2017.6.2.484>



# Table of Content

---

Abstract .....	i
Publications .....	ii
Table of Content .....	iii
List of Figures.....	viii
List of Tables.....	xii
List of Abbreviations .....	xiii
Acknowledgement .....	xiv
Chapter 1. Introduction .....	1
1.1. Background .....	1
1.2. Problem and Motivation .....	3
1.2.1. Design Method/Methodology for RiWAs.....	3
1.2.2. Quality Attributes Focused by the Research Problem.....	4
1.2.3. Summary of the Literature Review .....	6
1.3. Research Aim and Hypothesis .....	7
1.4. Research Objectives.....	8
1.5. Methodology .....	9
1.5.1. Research Type and Framework .....	9
1.5.2. Research Scope.....	12
1.5.3. Implementation of the Proposed Design Methodology.....	13
1.5.4. Demonstrating the Adoption of the RiWAsML/RiWAsDM.....	18
1.5.5. Evaluation.....	19
1.6. The Solution and Research Contributions.....	23
1.7. Structure of the Thesis .....	23
Chapter 2. Background .....	25
2.1. Related Software Engineering Concepts.....	25
2.1.1. Software Engineering Methods .....	25
2.1.2. Software Engineering Methodology .....	26
2.1.3. Software Engineering Approaches.....	27
2.2. Software Design Methodologies and Methods.....	29
2.2.1. Software Design Methodologies.....	29
2.2.2. Software Design Process .....	30
2.2.3. Software Design Approaches .....	32
2.2.4. Software Modelling Languages .....	33

## Table of Content

2.3.	Rich Web-based Applications.....	37
2.3.1.	Standalone Systems vs Distributed Systems.....	38
2.3.2.	Web-based Applications vs Rich Internet Applications vs Rich Web-based Applications.....	38
2.3.3.	Delta-Communication.....	39
2.3.4.	Types of Rich Web-based Applications.....	41
2.3.5.	General Characteristics and Essential Features of RiWAs.....	41
2.3.6.	Rich Web-based Applications Engineering.....	44
2.4.	Chapter Summary.....	45
Chapter 3.	Review of Available Related Solutions.....	46
3.1.	Architectural Styles for RiWAs.....	46
3.1.1.	SPIAR: A Component and Push-based Architectural Style for AJAX Applications (2008) [108].....	46
3.1.2.	jAGA: jQuery-based Ajax General Interactive Architecture (2012) [109].....	47
3.1.3.	RiWAArch Style: Rich Web-based Applications Architectural Style (2020) [12]...	47
3.1.4.	Summary of the Available Architectural Styles Review.....	48
3.2.	High-level Design Methods/Methodologies.....	49
3.2.1.	Informal Box-and-Line Drawing.....	49
3.2.2.	Formal Architectural Description Languages.....	50
3.2.3.	UML-based/UML-like Design Methods/Methodologies.....	53
3.2.4.	Summary of the Available High-level Design Methods/Methodologies Review.....	60
3.3.	Low-level Design Methods/Methodologies.....	61
3.3.1.	UWE (version 3.0, 2016) [23] and UWE-R (2009) [123].....	61
3.3.2.	WebML [107] and IFML (version 1.0, 2015) [124].....	63
3.3.3.	SysML (Version 2.0 Beta 1, 2023) [126] [127] [128].....	63
3.3.4.	Summary of the Low-level Design Methods Review.....	64
3.4.	Research Work Related to RiWAs Designing.....	65
3.5.	Chapter Summary: The Analysis of the Literature Review.....	70
Chapter 4.	Requirements for RiWAsML and RiWAsDM.....	71
4.1.	Attributes Required to be Satisfied by the RiWAsDM.....	71
4.1.1.	Attr 1. Simplicity.....	71
4.1.2.	Attr 2. Adoptability.....	72
4.2.	Requirements for Common Model-elements of RiWAs.....	74
4.2.1.	R1 – Model-elements Naming Label.....	74
4.2.2.	R2 – Communication Channels in RiWAs.....	76
4.3.	Requirements for High-level Design of RiWAs.....	78

## Table of Content

4.3.1.	R3 – High-level Processing Elements of RiWAs .....	78
4.3.2.	R4 – High-level Views Element .....	84
4.3.3.	R5 – Additional High-level Elements .....	85
4.3.4.	R6 – High-level Design Models .....	85
4.4.	Requirements for Low-level Design of RiWAs .....	86
4.4.1.	R7 – Low-level Modelling of Views .....	87
4.4.2.	R8 – Low-level Modelling of Components .....	90
4.4.3.	R9 – Low-level Modelling of Connectors .....	92
4.4.4.	R10 – Low-level Design Models for RiWAs .....	93
4.5.	Chapter Summary .....	97
Chapter 5.	RiWAsML: High-level Modelling Language .....	98
5.1.	Notations for General Model-elements of RiWAs .....	98
5.1.1.	R1 – Elements Label Notation .....	98
5.1.2.	R2 – Communication Channels Notations .....	99
5.2.	Notations for High-level Model-elements of RiWAs .....	101
5.2.1.	R3 – Notation for Tier Element .....	101
5.2.2.	R3 – Notation for Platform Element .....	102
5.2.3.	R3 – Notation for Application Element .....	105
5.2.4.	R3 – Notations for Components .....	106
5.2.5.	R3 – Notation for Connectors .....	108
5.2.6.	R4 – Notation for Views .....	108
5.2.7.	R5 – Notation for Additional High-level Elements .....	109
5.3.	R6 – High-level Design Models and UML Profiles for the RiWAsML .....	110
5.3.1.	Profile for Label Element .....	111
5.3.2.	Profile for Communication Channels .....	112
5.3.3.	Level 1 Applications Model .....	113
5.3.4.	Level 2 View-Process Model .....	116
5.3.5.	Level 1+2 Architectural Model .....	117
5.4.	Chapter Summary .....	119
Chapter 6.	RiWAsML: Low-level Modelling Language .....	120
6.1.	Notations for Low-level Model-elements of RiWAs .....	120
6.1.1.	R7 – Notations for Low-level Views and Related Elements .....	120
6.1.2.	R8 – Notations for Low-level AppControllers and Related Elements .....	133
6.1.3.	R8 – Notations for Low-level AppModel and Related Elements .....	141
6.1.4.	R9 – Notations for Low-level Connectors and Related Elements .....	143
6.2.	R10 – Low-level Design Models and UML Profiles for the RiWAsML .....	145

## Table of Content

6.2.1.	View-Navigation Model.....	145
6.2.2.	View Model.....	146
6.2.3.	AppControllers Model and ControllerClass Model .....	147
6.2.4.	AppModel Model and ModelClass Model .....	149
6.2.5.	DC-bus Model and EndpointsCollection Model.....	149
6.2.6.	View-Controller Model.....	150
6.2.7.	View-Process Sequence Model .....	151
6.3.	Chapter Summary .....	152
Chapter 7.	Rich Web-based Applications Design Methodology .....	154
7.1.	Introduction to the RiWAsDM.....	154
7.2.	RiWAsML Architecture, Language Reference, and Rules and Guidelines .....	155
7.2.1.	RiWAsML Architecture.....	155
7.2.2.	RiWAsML Language Reference.....	157
7.2.3.	Rules and Guidelines for Designing RiWAs with RiWAsML .....	163
7.3.	The RiWAsDM Process.....	168
7.3.1.	RiWAs Design Approach with RiWAsDM.....	169
7.3.2.	RiWAs Engineering Approach with RiWAsDM.....	169
7.3.3.	Guidelines for Adopting RiWAsML-based Designing into Agile Environments ..	169
7.4.	Chapter Summary .....	171
Chapter 8.	Use Cases .....	172
8.1.	High-level Design Attributes of the RiWAsML.....	172
8.1.1.	Shopping System: Improve Simplicity and Readability with Tiers.....	172
8.1.2.	Book Club App: UML Node vs RiWAsML Platform .....	176
8.1.3.	MiCADO-Edge – A Cloud-to-Edge Computing Architecture [180].....	177
8.2.	Learning Management System.....	180
8.2.1.	LMS – The Use Case Diagram.....	180
8.2.2.	LMS – High-level Design .....	181
8.3.	Learning Management System With a Web Service .....	184
8.3.1.	LMS with a Web Service – High-level Design .....	184
8.3.2.	LMS with a Web Service – Low-level Design .....	185
8.4.	Chapter Summary .....	193
Chapter 9.	Evaluation.....	194
9.1.	Self-Evaluation.....	194
9.2.	Contextualised Comparisons.....	202
9.3.	Expert Evaluation .....	208
9.4.	Triangulating the Results .....	210

## Table of Content

9.5. Chapter Summary .....	210
Chapter 10. Conclusion.....	212
10.1. Achievements of Research Aim and Objectives.....	212
10.1.1. Achievements of the Objectives .....	212
10.1.2. Proving the Hypotheses.....	213
10.1.3. Fulfilment of the Research Aim.....	213
10.2. Contributions .....	214
10.3. Reflections on Challenges.....	216
10.4. Limitations .....	217
10.5. Future Work .....	218
10.6. Chapter Summary .....	219
References .....	220
Appendices .....	I
Appendix A. Example: Shopping App – Level 1+2 Architecture Diagram (large).....	I
Appendix B. Use Cases: High-level Designing with RiWAsML .....	I
Appendix B.1. Shopping System – Original Architecture Without Using Tiers (large) .....	I
Appendix B.2. Shopping System – Architecture With Tiers (large).....	II
Appendix B.3. Shopping System – Level 1 Applications Diagram (large).....	III
Appendix B.4. MiCADO-Edge [180] Architecture drawn using the RiWAsML.....	IV
Appendix C. Use Case: Learning Management System (LMS) .....	V
Appendix C.1. Use Case: LMS – Use Case Descriptions .....	V
Appendix C.2. Use Case: LMS – Architecture (large).....	VIII
Appendix C.3. Use Case: LMS with Web Service – Architecture (large).....	IX
Appendix C.4. Use Case: LMS – View-Navigation Diagram (large) .....	X
Appendix D. Expert Evaluation.....	XI
Appendix D.1. Expert Evaluation Form.....	XI
Appendix D.2. Expert 1 Feedback .....	XXXVIII
Appendix D.3. Expert 2 Feedback .....	L
Appendix D.4. Expert 3 Feedback .....	LIX
Glossary .....	LXVII

## List of Figures

---

Figure 1.1 SERM framework [31].....	10
Figure 1.2 DSML implementation process by Ulrich Frank [17].....	14
Figure 1.3 The hybrid modelling language process model used in the development of IAML [42]	15
Figure 1.4 RiWAsDM implementation process .....	16
Figure 1.5 Micro Process ‘Evaluation and Revision’ of DSML [17] .....	19
Figure 2.1 Software engineering approaches .....	28
Figure 2.2 OMG’s meta-model hierarchy [86].....	35
Figure 2.3 Example of core elements of UML profile diagram [95] .....	37
Figure 2.4 Taxonomy for the development techniques and technologies of DC [10] .....	40
Figure 2.5 Taxonomy for the client-component(s) of RiWAs [11] .....	41
Figure 3.1 The Rich Web-based Applications Architectural style (RiWAArch style) [12].....	48
Figure 3.2 The list of UML diagrams [119] .....	54
Figure 4.1 Communication channel types between elements in the RiWAArch style .....	77
Figure 4.2 Processing elements in RiWAs.....	79
Figure 5.1 Proposed notation: communication channels.....	100
Figure 5.2 Proposed notation: communication channel with numbered connectors.....	100
Figure 5.3 Proposed notation: push-DC.....	101
Figure 5.4 Example: Tier elements.....	102
Figure 5.5 Example: using UML’s nested nodes for platforms [146].....	103
Figure 5.6 Proposed notation: Platform element .....	104
Figure 5.7 Example: nested platforms .....	105
Figure 5.8 Example: Application element.....	106
Figure 5.9 Proposed notation: component element.....	107
Figure 5.10 Example: high-level Controllers and Model components .....	107
Figure 5.11 Example: DC-engine and DC-bus connectors .....	108
Figure 5.12 Proposed notation: Views element (on the left) and an example of the use (on the right) .....	109
Figure 5.13 Proposed notation: additional high-level elements.....	109
Figure 5.14 Proposed notation: Notes element.....	110
Figure 5.15 UML meta-model’s labels [119].....	111
Figure 5.16 UML profile: RiWAsML Label element.....	112
Figure 5.17 UML profile: RiWAsML communication channels.....	113
Figure 5.18 Example: L1 Applications diagram.....	114
Figure 5.19 UML profile: applications model and its elements .....	115

## List of Figures

Figure 5.20 Example: View-Process diagrams for the Browser app (left) and Web server app (right)	116
Figure 5.21 UML profile: view-process model and its elements	117
Figure 5.22 Example: shopping app – L1+2 Architectural diagram	118
Figure 5.23 UML profile: L1+2 Architecture model	118
Figure 6.1 Example: View – a web page	121
Figure 6.2 Example: View – GUI Input/Output elements	122
Figure 6.3 Example: View – nested GUI elements	123
Figure 6.4 Example: View – GUI containers/sections	124
Figure 6.5 Example: View – GUI popup	125
Figure 6.6 Example: View – Viewpart notation for less complex scenarios	126
Figure 6.7 Example: View – Viewpart notation for moderate complex scenarios	127
Figure 6.8 Example: View – Viewpart notation for highly complex scenarios	128
Figure 6.9 Example: View – SharedViewpart	129
Figure 6.10 Example: View – SharedViewpart with nested Viewparts for different actors	129
Figure 6.11 Example: Views – menu-based navigation design	131
Figure 6.12 Example: views – menu-based navigation design with ViewPackage	132
Figure 6.13 Example: Views – process-based navigation	132
Figure 6.14 Proposed notation: ControllerClass element (on the left) and an example use of it (on the right)	133
Figure 6.15 Example: View and its ControllerClass – event-trigger notation for a less complex scenario	135
Figure 6.16 Example: View(on the left) and its ControllerClass (on the right) – event-trigger notation for a complex scenario	135
Figure 6.17 Example: View and its ControllerClass – event handler reading data from the view	136
Figure 6.18 Example: View and its ControllerClass – event handler showing output on the View	136
Figure 6.19 Example: View and its ControllerClass – event handler showing output on the View – communication channels with numbered flow connectors	137
Figure 6.20 Example: View and its ControllerClass – invoking a popup/toggle	138
Figure 6.21 Example: ControllerClass – communicating with the Client-model	138
Figure 6.22 Example: ControllerClass – communicating with the Client-model – numbered flow connector notation	138
Figure 6.23 Example: ControllerClass – sending a DC request	140
Figure 6.24 Example: ControllerClass – DC response handling	140
Figure 6.25 Proposed notation: ModelClass	141
Figure 6.26 Example: AppModel diagram of a ServerModel	142

## List of Figures

Figure 6.27 Example: ModelClass – communicating with the other elements .....	142
Figure 6.28 Proposed notation: EndpointsCollection element .....	143
Figure 6.29 Example: EndpointsCollection – communicating with the controller.....	144
Figure 6.30 Example: EndpointsCollection – communicating with ModelClass.....	144
Figure 6.31 UML profile: View-Navigation model.....	146
Figure 6.32 UML profile: View model.....	147
Figure 6.33 Example: AppControllers diagram.....	148
Figure 6.34 UML profile: AppControllers model, ControllerClass model, and their elements .....	148
Figure 6.35 UML profile: AppModel .....	149
Figure 6.36 Example: DC-bus diagram (on the left) and EndpointsCollection diagram (on the right) .....	150
Figure 6.37 UML profile: DC-bus model and EndpointsCollection model .....	150
Figure 6.38 UML profile: View-Controller model .....	151
Figure 6.39 Example: View-Process Sequence diagram.....	151
Figure 6.40 UML profile: View-Process Sequence model.....	152
Figure 7.1 RiWAsML architecture .....	155
Figure 7.2 Example: mapping between the RiWAsML’s Packaging layer and development layer .....	157
Figure 7.3 RiWAsML diagrams .....	158
Figure 7.4 The scrum framework [176] .....	170
Figure 7.5 The DevOps tool chain [3] .....	170
Figure 7.6 CI-CD life cycle [177].....	170
Figure 8.1 Use case: shopping system – original architecture.....	173
Figure 8.2 Use case: shopping system – architecture with tiers .....	174
Figure 8.3 Use case: shopping system – Level 1 Applications diagram .....	175
Figure 8.4 Use case: book club app – platforms designed using UML node elements [179] .....	176
Figure 8.5 Use case: book club app – platforms designed with RiWAsML .....	177
Figure 8.6 Use case: MiCADO-Edge – original architecture [180].....	178
Figure 8.7 Use case: MiCADO-Edge – Architecture drawn using the RiWAsML .....	179
Figure 8.8 Use case: LMS – use case diagram .....	181
Figure 8.9 Use case: LMS – original box-and-line architecture.....	182
Figure 8.10 Use case: LMS – architecture designed with the RiWAsML .....	183
Figure 8.11 Use case: LMS with web service – L1+2 Architectural diagram.....	185
Figure 8.12 Use case: LMS – View-Navigation diagram .....	186
Figure 8.13 Use case: LMS with web service – View-Process Sequence diagram .....	188
Figure 8.14 Use case: LMS with web service – browse repo View-Controller diagram .....	189
Figure 8.15 Use case: LMS with web service – DC-bus and an EndpointsCollection diagrams ...	191



## List of Figures

Figure 8.16 Use case: LMS with web service – an AppModel diagram.....	192
Figure 8.17 Use case: LMS with web service – an EndpointsCollection and ModelClass elements in one diagram.....	192
Figure Appendix A.1 Example: shopping app – level 1+2 architecture (large).....	I
Figure Appendix B.1 Use case: shopping system – original architecture (large).....	I
Figure Appendix B.2 Use case: shopping system – architecture with tiers (large).....	II
Figure Appendix B.3 Use case: shopping system – level 1 applications diagram (large).....	III
Figure Appendix B.4 Use case: MiCADO-Edge Architecture drawn using the RiWAsML (large).....	IV
Figure Appendix C.2 Use case: LMS – architecture (large).....	VIII
Figure Appendix C.3 Use case: LMS with web service – architecture (large).....	IX
Figure Appendix C.4 Use case: LMS – view-navigation diagram (large).....	X

## List of Tables

---

Table 1.1 Summary of the literature review .....	7
Table 1.2 Rating categories of SERM [31] .....	11
Table 1.3 Self-evaluation scale.....	20
Table 1.4 Expert evaluation feedback scale .....	21
Table 2.1 Web applications vs. RiWAs .....	39
Table 3.1 Summary of the architectural styles review .....	49
Table 3.2 Summary of the ADLs review .....	53
Table 3.3 Summary of the UML-based methods/methodologies review.....	59
Table 3.4 Summary of reviews of high-level design methods/methodologies for RiWAs .....	60
Table 3.5 Summary of low-level design methods review .....	64
Table 3.6 Summary of the review of related solutions .....	70
Table 4.1 Mapping the quality attributes to the requirements .....	97
Table 6.1 A suggested list of shortcodes for GUI elements .....	123
Table 7.1 Models for package and single size of different types of elements.....	156
Table 7.2 RiWAsML's general elements and their notations.....	158
Table 7.3 RiWAsML's high-level diagrams and their notations.....	159
Table 7.4 RiWAsML's low-level diagrams and their notations.....	161
Table 7.5 RiWAsML's recommended events handling coding style.....	167
Table 9.1 Evaluation scale (refer to Table 1.2 in Section 1.5.5.2).....	194
Table 9.2 The analysis of the self-evaluation results .....	195
Table 9.3 The analysis of the contextualised comparison.....	202
Table 9.4 The analysis of the expert evaluation .....	208
Table 9.5 The analysis of the RiWAsDM evaluation .....	210

## List of Abbreviations

---

ADL – Architecture Description Language  
AMDD – Agile Model-Driven Development  
ASE – Agile Software Engineering  
DC – Delta-Communication  
DSML – Domain-Specific Modelling Language  
FOMDD – Feature-Oriented Model-Driven Development  
GPML – General-Purpose Modelling Language  
GUIs – Graphical User Interfaces  
IoT – Internet of Things  
JS – JavaScript  
MBE/MBSE – Model-based Engineering/Model-based Software Engineering  
MDE/MDSE – Model-Driven Engineering/Model-Driven Software Engineering  
MOF – Meta Object Facility  
MVC – Model-View-Controller  
OODD – Object-Oriented Design and Development  
RIA – Rich Internet Application  
RiWA – Rich Web-based Application  
RiWAArch style – Rich Web-based Applications Architectural style  
RiWAsDM – Rich Web-based Applications Design Methodology  
RiWAsML – Rich Web-based Applications Modelling Language  
SE – Software Engineering  
SDLC – Software Development Life Cycle  
SPDC – Simple Pull Delta-Communication  
SSAD – Structured System Analysis and Design  
TTs – Technologies and Techniques  
WS – WebSocket  
XMI – XML Metadata Interchange

## Acknowledgement

---

First, I want to express my gratitude to Prof. Sriyani Wickramasinghe at the Rajarata University of Sri Lanka for motivating and pushing me to initiate my research life and enter academia.

I'm grateful to my director of studies, Dr Alexandar Bolotov, at the School of Computer Science and Engineering, University of Westminster, for valuing my previous research work and selecting me as a candidate for the PhD. Your ultimate support during the entire PhD program is invaluable in all aspects. I also thank Dr Simon Courtenage for accepting to be my secondary supervisor and for feedback where necessary.

Also, I'm grateful to the former PhD coordinator, Dr Andrzej Tarczynski, for working with Dr Alexander to offer me a scholarship; I'll never forget what you two did to keep the place for me. I also want to thank the University of Westminster for the offer. I would not be able to start a PhD without the scholarship.

I want to thank the current PhD coordinator, Dr Alexandra Psarrou, and the other staff members immensely for the tremendous support provided to convert the mode of the PhD after Covid duration and for offering me an option to complete the PhD as a full-time resident student; without that option, I would not be able to complete the PhD.

I must recognise the domain experts who contributed to this thesis by evaluating the research outcomes. They devoted their valuable effort and time amidst their busy schedules to read the materials and have deep discussions to accomplish the evaluation based on their expertise.

I have to thank my landlord in London, Mr Athula Badalage, for providing me with a comfortable environment in which to study without disturbances. Without his caring support, I wouldn't be able to manage everything with less stress.

I should thank my friends and colleagues for always motivating me and being there for me while tolerating my ignorance and mistakes during the heavy workloads. I'm glad you are all around me, providing a supportive environment.

Last but not least, I'm obliged to my parents, sisters Thushari and Anusha, and brother Surendra; I know how much this means to you, which drove me to move on till the end, bearing all the difficulties. Without the sacrifice made by Surendra during the last stage, I wouldn't have been able to make the great move to migrate to the UK and finish my PhD.

Thank you to everyone who supported me; by any means, they are all counted.

**Nalaka R. Dissanayake (2024)**

# Chapter 1. Introduction

---

*“A designer can mull over complicated designs for months.  
Then suddenly, the simple, elegant, beautiful solution occurs to him.  
When it happens to you, it feels as if God is talking!  
And maybe He is.”*  
Leo Frankowski [1]

This chapter first briefly discusses the core background concepts of the research in Section 1.1, laying a foundation. Then, Section 1.2 defines the problem in this background, and the motivations for addressing the problem are discussed. Section 1.3 sets the research aim and the hypotheses, and Section 1.4 states the research objectives. After that, Section 1.5 details the methodology used in the research. Next, the proposed solution and the research contributions are indicated in Section 1.6. Finally, the structure of the thesis is given in Section 1.7.

## 1.1. Background

This section briefs the core background concepts of the research to provide a foundation for the rest of the chapter. Chapter 2 includes a complete account of the background research.

### Related Software Engineering Concepts

This section gives a summary of the materials detailed in Section 2.1.

Software Engineering (SE) uses methods and methodologies to govern the engineering process. A SE **method** is a systematic procedure, concept, or tool that assists in performing a particular type of work at the global level or within a specific phase to reach a certain goal and/or produce a defined set of software artefacts [2]. Global methods such as *Software Development Life Cycle (SDLC)* models are applied to the entire engineering project, and phase-specific methods like development methods/tools or testing procedures are utilised within a phase.

**SE methodology** can be seen as a global framework that governs the engineering project by defining a set of methods selected for a software engineering project based on specific criteria. These methods systematically explain how particular tasks are executed within the complete software engineering project. *DevOps* [3] and *Continuous Integration, Continuous Delivery (CI-CD)* [4] are trending SE methodologies that assist in **agile SE (ASE)**. There are phase/task-specific methodologies, which mainly pay attention to a specific phase of the SDLC; for example, **design methodologies** are utilised in the design phase, which is a type of methodology that overlooks the software designing aspects. Design methodologies use various methods to design different aspects of software systems; for example, ER diagrams to model databases and use case diagrams to design requirements. Different

**SE approaches** like *Model-Driven Software Engineering (MDSE)* and *Agile Software Engineering (ASE)* [5] signify design aspects at different levels. However, regardless of the SE approach, software designing/modelling still plays an important role; thus, software design methods and methodologies are significantly needful.

### **Software Design Methodologies and Methods**

This section introduces the materials detailed in Section 2.2.

The software design process models different aspects of the software systems, like architecture, algorithms, and databases [6]. The software design process comprises two levels:

1. **High-level/preliminary design (architectural design)** is “*the process of analyzing design alternatives and defining the architecture, components, interfaces, and timing and sizing estimates for a system or component*” [6]. **Architectural styles** offer a framework for designing system architectures.
2. **Low-level/detailed design** is “*the process of refining and expanding the preliminary design of a system or component to the extent that the design is sufficiently complete to be implemented*” [6]. The *Object-Oriented Design and Development (OODD)* paradigm has become the de facto standard in software designing/modelling and development. This thesis also considers the low-level design as the **development level**, which helps map the low-level design elements to their development by providing enough development-supportive details (refer to Section 7.2.1).

Software **design languages** (also called **modelling languages**) offer **models** and **model-elements** to design various aspects of software systems like structures and behaviours. The *Unified Modeling Language (UML)* [7] is the widely accepted *General-Purpose Modelling Language (GPML)* [8], which mainly assists in designing low-level aspects of software systems using the OODD paradigm [9]. The generic characteristics of the UML within its broad scope do not cater to the specificity of some software systems, such as web-based applications [9]; hence, dedicated *Domain-Specific Modelling Languages (DSMLs)* and methodologies are required [8]. UML’s lightweight extension mechanism is a tool for developing new UML-based DSML **meta-models** using new **UML profiles** (refer to Section 2.2.4 for software modelling languages and related concepts).

### **Rich Web-based Applications**

This section briefs the materials detailed in section 2.3.

*Rich Web-based Applications (RiWAs)* [10] are a type of web-based system that provides a higher user experience similar to using desktop applications compared to traditional web applications. RiWAs provide rich *Graphical User Interfaces (GUIs)* – identical to the GUIs in desktop applications – which use an advanced communication model named *Delta-Communication (DC)*

[11] to communicate with the server-side components faster and asynchronously (refer to Section 2.3.2 for the definition of the RiWAs and comparison with the standard web applications, and Section 2.3.3 for the DC). Commonly used modern apps, such as Facebook, Google apps, and Microsoft apps, are examples of RiWAs. The RiWAs are complex systems; besides, the **general characteristics and essential features** of RiWAs are identified (refer to Section 2.3.5), which are realised by the *Rich Web-based Architectural Style (RiWAArch style)* [12] (refer to Section 3.1.3). RiWAs development tools like libraries, frameworks, IDEs, and dependency management and build tools have immensely evolved over the last two decades to cater to the RiWAs' specificity [10] by assisting in developing the rich GUIs, DC, and related components. However, other RiWAs engineering concepts – like design patterns and testing methods – and tools for them have not been advanced much [13].

## 1.2. Problem and Motivation

The problem this thesis addresses is the **unavailability of a simple and adoptable design methodology for the RiWAs**. The following subsections elaborate on the focused aspects of the problem.

### 1.2.1. Design Method/Methodology for RiWAs

The RiWAs' development issues are solved in ad-hoc manners by the RiWA engineers/developers, and the methods/techniques used to address these issues may have introduced some extra work to the development of the RiWAs [14] [13]. For example, the developers may decide to implement a new processing layer for a particular RiWA to handle client-side events and business logic – without using proper style or pattern – which may increase the development complexity and, thus, efforts. In this setting, the RiWAs' complex client-server communication using the DC model further increases the development efforts [15].

When the development issues are handled ad-hoc, the unsystematic nature of engineering causes the components – which are already constructed for a specific RiWA – to lower the reusability [14]. For example, due to the difference in business logic, the client-side events and business logic handling layer implemented in the above case may not be used for another RiWA. Also, the same technique of implementing a layer for client-side events and business logic handling might not be suitable when distinct development tools are utilised. In such an environment, the experience gained from engineering one RiWA would not be applied to other projects as the situations are handled in unique ad-hoc ways.

This nature of RiWAs engineering can be considered a result of the lack of abstract models that can realise the RiWAs' general characteristics and essential features, which can directly assist in development. The conceptual abstraction stipulated by a dedicated systematic design methodology

with a DSML – which provides enough models to realise the RiWAs – and rules and guidelines to use them can possibly support the development of the RiWAs to overcome the said issues.

Even though there are some dedicated design methods/methodologies for the web-based application, they are not strong enough to support the specificity of RiWAs. A few styles and design methods have been introduced that are trying to realise the complexity of the RiWAs. These available solutions are reviewed in Chapter 3, and a summary is given in Section 1.2.2. It is noted that none of these available design methods or methodologies provides a simple and adoptable set of tools to design the RiWAs' general characteristics and essential features discussed in Section 2.3.5.

The literature survey noted that the design research and methods for RiWAs were not very active. Since the early stages of the RIAs/RiWAs, the requirements for modelling solutions have been discussed [16]; however, development tools have rapidly evolved, but other aspects have not [13]. SE engineers following agile software engineering methodologies and tools more in the direction of rapid development might be the reason for the lack of conceptual evolution. In that setting, contemporary SE relies on agile methodologies, and the need for a design methodology can be questioned. Agile-SE-related aspects are discussed in Section 2.1.3, stating the importance of software designing and discussing the advantages of the hybrid approach: *Agile Model-Driven Development (AMDD)*, motivating this study to address the research problem.

It is understood that development activities are directly affected by design concepts such as styles and patterns, and also, software design knowledge can positively influence development. Busch and Koch [15] (page 9) say that “*we need a model language that provides model-elements that allow us to model these distinguishing features of RIAs. Depending on the level of abstraction, we require a different type of model-elements*”. They continue stating that “*on design level, these features need a refinement indicating, e.g. more precisely how is the user interaction and the client-server communication. Finally, when building the platform-specific model or at the code level, the technologies used need to be specified in detail*”. This discussion helps understand the importance of having a dedicated design methodology for RiWAs.

### **1.2.2. Quality Attributes Focused by the Research Problem**

The main attributes focused by the research problem are **simplicity** and **adoptability** [17] [8]. The adoptability is discussed in this thesis by 4 attributes: **comprehensiveness**, **usability**, **development support**, and **integration support**. These attributes are generally explained in Section 2.2.4.2, and their context for a RiWAs design methodology is set in Section 4.1. This section elaborates on the problem by discussing the issues related to the lack of simplicity and adoptability and the motivation for incorporating them by a design methodology.



### **Simplicity**

This attribute expresses the less complex nature of a system/solution maintained by applying the principle of *separation of concerns* [18] [19] [20]. Decomposing a system towards identifying and separating the system elements provides greater realisation and, thus, management, which helps reduce complexity [18] [19]. A modelling language must identify the separate elements of the target systems and provide enough models and model-elements to design the general characteristics of the target systems in a less complex manner, maintaining a higher simplicity, which leads the modelling language to be a comprehensive solution [17]. The available solutions do not precisely identify the abstract elements of the RiWAs– especially the elements that handle distributed logic and DC – which cause the adoptability of these solutions to be low.

### **Adoptability**

A solution should be practically adoptable into engineering without being limited to a conceptual artefact. The following attributes are elected to ensure the adoptability of a design methodology.

### **Comprehensiveness**

The general characteristics and essential features of the RiWAs have been understood in literature (refer to Section 2.3.5) [14] [21] [12]. Available design methods/methodologies introduced for RiWAs focus on specific and limited characteristics; for example, the RUX-model [22] only discusses the GUIs and presentation concerns. None of the available solutions addresses all these general characteristics and essential features of the RiWAs; thus, they cannot be considered complete design solutions for RiWAs. Several methods/methodologies may be learned and utilised to design RiWAs; for example, the RUX method can be combined with UWE [23] to facilitate the GUI designing aspect [24]. This approach may not be feasible due to time, cost, and complexity factors. Moreover, since the available methods use different approaches and are involved with various notations, integration compatibility issues may arise, and new complexity and learning efforts can be introduced. On the contrary, adopting a single comprehensive solution is more beneficial.

A complete design methodology with enough models for the systematic development of the RiWAs can be seen as a contemporary requirement for the RiWAs [17], where it can provide enough support for modelling all the general features of RiWAs, including the DC and the business logic distributed between the client and server [14]. The criteria that define the comprehensiveness of the proposed RiWAsDM are discussed in the relevant sections (refer to Sections 2.2.4.2 and 4.1.2.1).

### **Usability (Learnability, Readability/Understandability)**

The readability of designs is a vital attribute [25] towards reducing the overhead of understanding the models and model-elements [26]. Higher readability and understandability of a design language can immensely assist in learning the language in the direction of improving usability. The available

design methods/methodologies exhibit many usability issues, as reviewed in Chapters 3, 5, 6, and 9, which should be addressed to improve the adoptability.

The complexity of design language plays a crucial role in usability [17] since, in the context of software systems, complexity encloses the difficulties in understanding, where it speaks about the psychological complexity of the system [27]. A complex solution can be seen as composed of many elements that interact [28], and relationships between elements are not perfectly known or familiar [29]. This thesis values associating the available knowledge of RiWAs' general characteristics, essential features, and development practices in the direction of reducing the complexity of a RiWAs design language towards improving usability and, hence, adoptability.

### **Development Support**

The design of a system should assist in the system's development, not only by providing enough realisation of the system but also by supporting the conversion of the designs into working code. For example, UML class diagrams can be directly converted to code, and some CASE tools even offer features to generate code for a target language using the class diagram. Available design solutions primarily provide conceptual results, mostly eliminating development-related discussions.

This thesis expects a RiWAs design language to produce diagrams which can be mapped into development and to offer rules and guidelines to support development based on the designs. If a design language provides ample support for development, reducing the design to development mapping efforts, the language can be considered a more adoptable solution.

### **Integration Support (Integrability)**

Currently, SE projects are primarily based on agile methodologies; discussing how to integrate design activities into agile-based environments is essential [30]. The available web applications/RIA/RiWAs design solutions do not provide this facility and, therefore, can be seen as less adoptable. If a design methodology offers a process with rules and guidelines to incorporate the design activities into agile-SE, it can be considered an adoptable methodology.

### **1.2.3. Summary of the Literature Review**

A comprehensive literature review is given in Chapter 3. A literature review summary is provided in Table 1.1 to evidence the problem's existence among the available design solutions.

As given in Table 1.1, the available solutions are either general, do not support DC and related aspects, are limited to high-level/low-level designing, or do not offer development or integration support. If a language is based on UML, this thesis considers it more usable, given that UML is the de facto standard software modelling language, and the software engineers generally know UML. Grounded in this notion, UML-based DSMLs are believed to be more usable than new languages which are not based on UML.

Table 1.1 Summary of the literature review

Solution	Context	Comprehensiveness	Usability	Dev and Integration
<b>MDA with UML</b>	-General.	-Primarily low-level -No DC	-De facto standard	-None
<b>Arc<sup>42</sup></b>	-General	-Primarily low-level -High-level diagram -No DC	-UML-based -Primarily documentation	-None
<b>SAP's OOA/OOM</b>	-General	-TAM is high-level -Primarily low-level -No DC	-UML-based	-None
<b>ArchiMate</b>	-General	-No DC	-UM-like new lang.	-None
<b>C4</b>	-RiWAs	-Primarily high-level -No DC	-No formal syntax (box-and-line)	-None
<b>UWE/UWE-R</b>	-Web/RIA	-Low-level -No DC	-UML-based	-None
<b>IFML</b>	-RiWAs -Interaction flows	-Low-level -front-end -No DC	-UM-like new lang.	-None
<b>SysML</b>	-General	-Low-level -No DC	-UML-based	-Highly support agile-SE
<b>IAML</b>	-RIA/RiWA	-Primarily low-level	-UML-based	-None

### 1.3. Research Aim and Hypothesis

This research aims to introduce a **simple and adoptable design methodology for the RiWAs**, named *RiWAs Design Methodology (RiWAsDM)*, whose adoptability is demonstrated with use cases and evaluated with multiple methods to ensure its validity. The core element of the RiWAsDM is a novel UML-based DSML meta-model for the RiWAs called *RiWAs Modelling Language (RiWAsML)*, which is defined as a UML extension. Refer to section 1.5.2 for the scope of the RiWAsDM. The simple and adoptable attributes are generally explained in Section 2.2.4.2, and their context is set in Section 4.1 under requirements for the RiWAsML/RiWAsDM. These attributes are chosen to ensure the simplicity of the RiWAsML, and the RiWAsDM can be practically adopted into actual RiWA engineering instead of being limited to a conceptual solution.

The following hypotheses are established to assist in achieving the research aim. These hypotheses are confirmed using reasoning in Section 10.1.2, and the fulfilment of the research aim is discussed in Section 10.1.3.

#### Hypothesis 1 [H<sub>0</sub>1]

It is essential to realise the abstract and complete architectural formalism of the RiWAs since the architecture provides the blueprint for the system. The realisation required to introduce a comprehensive set of design models and model-elements for RiWAs can be gained via an abstract architectural style, which can realise all the general characteristics and essential features of the RiWAs. Further, the comprehensive realisation of such an architectural style can ensure the simplicity and adoptability of the models. Based on this argument, the **H<sub>0</sub>1** is set as follows.

**H<sub>01</sub>:** A comprehensive set of RiWAs design models and model-elements can be identified to design all the general characteristics and essential features of the RiWAs based on a solid abstract architectural style, maintaining higher simplicity and adoptability.

### **Hypothesis 2 [H<sub>02</sub>]**

Utilising the models and model-elements identified while proving the H<sub>01</sub>, a simple and adoptable DSML for RiWAs can be produced. This DSML should be based on UML for higher usability. UML's extension mechanism can be used to implement UML-based DSML, creating UML profiles for the new models and model-elements. Considering these facts, the **H<sub>02</sub>** is defined as follows.

**H<sub>02</sub>:** A simple and adoptable UML-based DSML for RiWAs can be implemented using the comprehensive set of models and model-elements identified while proving the H<sub>01</sub>.

### **Hypothesis 3 [H<sub>03</sub>]**

Based on the DSML introduced while proving the H<sub>02</sub>, a simple and adoptable design methodology can be implemented, which discusses the RiWAs design and engineering approaches and provides guidelines for adopting the RiWAs designing activities into agile SE projects. Built on this notion, the H<sub>03</sub> is set as follows.

**H<sub>03</sub>:** A simple and adoptable RiWA design methodology can be produced – utilising the UML-based DSML introduced while satisfying the H<sub>02</sub> – which provides RiWAs design and engineering approaches and guidelines for adopting RiWAs designing into agile-SE.

## **1.4. Research Objectives**

The following objectives were set in order to achieve the research aim while proving the hypotheses.

**Obj 1. Identify an abstract comprehensive architectural style for the RiWAs:** To introduce a novel design methodology, it is crucial to understand the abstract architectural formalism of the RiWAs, which can realise the general characteristics and essential features of RiWAs. The lack of realisation of the architectural formalism of the RiWAs can be seen as a reasonable circumstance for the available RiWAs design methods/methodologies to be highly diverse; hence, incomplete and not adoptable. Since architectures help to reduce complexity by increasing the realisation of the systems, it is practical to identify a solid abstract comprehensive architectural style and implement a DSML based on it to reduce the complexity and increase the adoptability. Considering this position, identifying an abstract comprehensive architectural style, which can provide a foundation for the

RiWAsML, is set as the first objective of the research. This objective is satisfied in section 3.1, helping to prove research hypothesis 1.

**Obj 2. Introduce the RiWAsML:** Introduce the proposed RiWAsML to design the general characteristics and essential features of the RiWAs based on the abstract comprehensive architectural style identified while achieving Obj 1, utilising: the knowledge gained via extensive literature survey, the study of the RiWAs development technologies and techniques, RiWAs development experience, and intensive brainstorming. This objective is satisfied in Chapters 4, 5, and 6 while proving the research hypotheses 1 and 2.

**Obj 3. Introduce the RiWAsDM:** This objective is set to introduce the proposed simple and adoptable design methodology for the RiWAs based on the RiWAsML implemented by Obj 2. Chapter 7 satisfies this objective, also proving the research hypothesis 3.

**Obj 4. Demonstrate the utilisation of the RiWAsDM through use cases:** introducing a conceptual designing methodology does not practically provide adequate assistance to actual RiWAs engineering unless it is demonstrated and discussed how the methodology is practically utilised. Therefore, it is crucial to demonstrate the utilisation of the introduced RiWAsDM through use cases, and this objective is set to accomplish that. The use cases are given in Chapter 8, as proof of concept of the RiWAsDM's adoptability.

**Obj 5. Evaluate the introduced RiWAsDM:** The final objective is set to finalize the research aim by evaluating the produced RiWAsDM using multiple methods stated in Section 1.5.5 to ensure its simplicity and adoptability. The evaluation is discussed in Chapter 9.

## 1.5. Methodology

The literature survey found no methods or methodologies for introducing a new design methodology. This research's methodology is primarily based on the methods utilised to produce the RiWAArch style [12], which was inspired by Fielding's work that introduced the REST architectural style [19].

### 1.5.1. Research Type and Framework

Two main aspects governed the selection of research methods: the research type and the research framework, which are discussed in the following sections.

#### Research Type

Many aspects of the research are related to the conceptual and qualitative aspects, as follows. This research aims to produce a design methodology with a design language, which is a conceptual artefact. Software designing is creative work, and designing a modelling language can also be seen as creative; consequently, conceptual and qualitative. It is not very practical to gather requirements from the general users of software models, who are the RiWAs designers, as they may not have

enough academic-level experience and conceptual understanding of the abstraction required to produce a DSML; finding experts with strong academic knowledge and engineering experience is not much feasible. The outcomes of the research are a design model and its notations, which are conceptual and cannot be executed, and gather results for analysis; therefore, the validation of the outputs heavily depends on intensive brainstorming and reasoning rather than on empirical evidence. The research attempts to satisfy quality attributes like simplicity, learnability, readability, and understandability, which cannot be quantitatively measured, and the effects of these attributes are qualitatively analysed. Considering these characteristics, this research can be seen as qualitative, where quantitative research methods are not applicable. These aspects make this research a conceptual and qualitative work.

A few empirical activities took place, such as (1) studying the available solutions and gathering details related to their models and model-element and (2) collecting data from domain experts to evaluate the RiWAsDM. However, these data were analysed employing reasoning instead of statistical methods. Therefore, the type of this research is reflected as a qualitative and conceptual study, which draws arguments based on qualitative/conceptual facts rather than data or empirical study; hence, suitable qualitative/conceptual methods are utilised over empirical methods.

### Research Framework

The framework provided by the *Software Engineering Research Methodology* (SERM) [31] is selected to govern the research methodology, and SERM's rating categories are utilised to rate the research. SERM realises major research paradigms such as post-positivism and provides a framework with three pillars: conceptual, formal, and development, as shown in Figure 1.1.

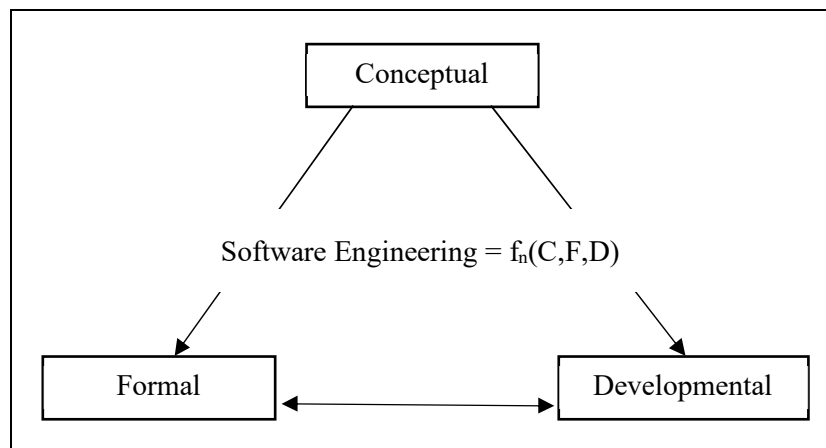


Figure 1.1 SERM framework [31]

The SERM defines SE research as a combination of Conceptual (C), Formal (F), and Development (D) dimensions as expressed by the function *Software Engineering = f<sub>n</sub>(C, F, D)*. Based on this framework, SERM also provides a mechanism to categorize and rate the SE research, as explained in Table 1.2.

Table 1.2 Rating categories of SERM [31]

Rating	Research Dimensions		
	Conceptual	Formal	Developmental
<b>High</b>	Major extensions or generalizations of an existing concept or a new concept	Defined in math and logic terms; formal definition or proofs; mathematical description.	Prototype or model with validation and verification
<b>Medium</b>	Incremental extension and/or generalization of an existing concept	Definitional without the math and/or logic proofs; establishes correctness criteria.	Prototype or model with limited functionality
<b>Low</b>	Existing concept with limited extensions	Descriptive details and conjectures	Discussion of program requirements
<b>None</b>	No new concept	No formal definitions	No implementation described

Based on the SERM rating mechanism, this research can be recognized as follows.

**Conceptual:** This thesis produces a novel design methodology with a new DSML for RiWAs. The RiWAsML is implemented as a UML extension; however, the UML extension mechanism is a tool to implement new DSML; therefore, the RiWAsML can be considered a new concept as well as the RiWAsDM. According to the SERM, this research's rating of conceptual dimension can be recognized as high.

**Formal:** Since this is conceptual and qualitative research, despite using quantitative mathematical methods, this thesis uses suitable qualitative methods based on logical reasoning for the RiWAsDM implementation and evaluation. A process is formulated for implementing the RiWAsDM (see Section 1.5.3), and the RiWAsML is based on UML, which is a formal modelling language. Multiple formal methods are utilised to evaluate the introduced methodology (refer to section 1.5.5). Considering these aspects, this research's rating of formal dimension can be acclaimed as medium.

**Developmental:** This research identifies the requirements for a design methodology and implements it with a new DSML, which includes a comprehensive set of models to design RiWAs and also provides rules and guidelines to map the design to development. The adoptability of the design methodology is verified with real-world use cases while discussing the development of the designs. The design methodology is further validated by evaluating it with multiple methods. These artefacts can set the research's developmental dimension's rating as high.

According to the SERM, the research can be ranked high, considering its conceptual, formal, and development ratings.

### 1.5.2. Research Scope

This section specifies the scope of the research in the direction of setting the context for the target systems. The research narrows the scope and focuses on the core aspects of the RiWAs and RiWAs designing with the notion that if the core aspects can be successfully satisfied, then future work can efficiently extend the research (see Section 10.5 for future work). The following scope limitations are applied to the research.

**The size of the RiWAs:** the 2-tier client-server architecture is the core formalism of the web-based systems. However, the RiWAs process data and use databases for persistence; therefore, the 3-tier architecture can be considered the core formalism of the RiWAs. If a SE concept can realise the RiWAs' core formalism, it can be extended to address the expanded forms of the target systems. Based on this notion, this research limits the size of the target RiWAs to the 3-tier architecture. Once a DSML is introduced for the 3-tier RiWAs, the DSML would be extended to larger n-tier RiWAs later.

**Type of the RiWAs:** as web-based systems, browser-based systems are the main type of the RiWAs. Hence, this research primarily studies browser-based RiWAs. Still, the research considers RiWAs with mobile-based and IoT-based clients and web services where necessary and discuss some related details.

**RiWAs implementation/deployment:** the implementation/deployment technologies are limited to the 3-tier systems, aligning with the RiWAs' size scope, avoiding the below.

- Service Oriented Architecture (SOA) or micro-services.
- Cloud computing and integration, and edge computing.
- Cloud services for deployment.

**Feature of the RiWAs:** this thesis narrows the scope of the target RiWAs to the systems that perform CRUD operations on the database, eliminating the following.

- Use of third-party services to implement features like authentication, email, SMS, or payments.
- Push notifications and/or real-time updates.
- Multimedia uploading, processing, and streaming.
- Data Science (DS), Artificial Intelligence (AI), or Machine Learning (ML) related features such as analytics and chatbots. These are functions/features of a system and can be implemented as modules/services extending a RiWA; therefore, they can be seen as future work (refer to Section 10.5 for future work).



**RiWAs design:** this research's primary focus on the design-related aspects lies in the following.

- **RiWAArch style-based high-level designing:** The RiWAArch style was selected since it firmly realises the general characteristics and essential features of the RiWAs, as well as high-level elements and their configurations (refer to Section 3.1.3).
- **Structural designing:** A strong understanding of a system's structural aspects is required for behavioural designing; thus, attention is given to realising structural formalism.
- **Top-down designing:** Since the RiWAArch style is selected as the base formalism, the high-level designing is accompanied by the style, which can aid the low-level design.
- **UML-based design:** UML is selected since it is the de facto standard software design language (refer to Section 3.2.3.1) and provides the extension mechanism for implementing DSMLs (refer to Section 2.2.4.5).

**RiWAs Design Methodology:** The RiWAsDM's primary focus is on implementing the RiWAsML and discussing the integration of RiWAsML-based designing into RiWAs engineering by introducing a design methodology. Therefore, some other aspects, like implementing CASE tools or support for other phases like requirements engineering or testing, are kept out of the scope.

### 1.5.3. Implementation of the Proposed Design Methodology

A challenging task of this research was the implementation of the proposed RiWAsDM since no formal methods or methodologies for implementing new methods/methodologies were found in the literature. Several accessible publications which produce approaches, methods, and methodologies were referred to [22] [32] [33] [34] [35] [36] [37]; it was noted that they mainly present their new approach/method/methodology and do not underpin the research methodology of producing their results.

Some research publications that produce UML profiles for target domains were examined, and the following points were noted, which provided some insight.

- The paper titled *UML Profile for Aspect-Oriented Software Development* (2003) briefly states some requirements for their new UML profile [38].
- The publication *A real-time profile for UML* (2006) [39] analyses the target systems, identifies the characteristics, and defines the context for a new UML profile. Also, a running example is discussed, and the target system characteristics are examined profoundly.
- *Towards a UML profile for data intensive applications* (2016) [40] uses an approach with 4 steps: (1) produce conceptual models for each abstraction level, (2) identify concepts for quality assessments, (3) define the proposed profile, and (4) assess the new profile using case studies.
- The authors of the paper *An UML profile for representing real-time design patterns* (2018) [41] have examined the target systems and specified the context for the target domain. A case study

example is also given to depict the implementation of the proposed profile and evaluate its effects. The authors further evaluate their new profile using a comparative analysis by comparing the profile with similar solutions using a context comprised of the properties *Variability*, *Expressivity*, *Consistency*, *Completeness*, *Traceability*, and *Composition* with a scale of 3 values: *Partially verified*, *Not verified*, and *Verified*.

More systematic approaches to implementing DSMLs were found in the following work.

- The research report *Outline of a Method for Designing Domain-Specific Modelling Languages* (2010) by Ulrich Frank [17] presents a process with 8 steps to implement DSML, given in Figure 1.2.

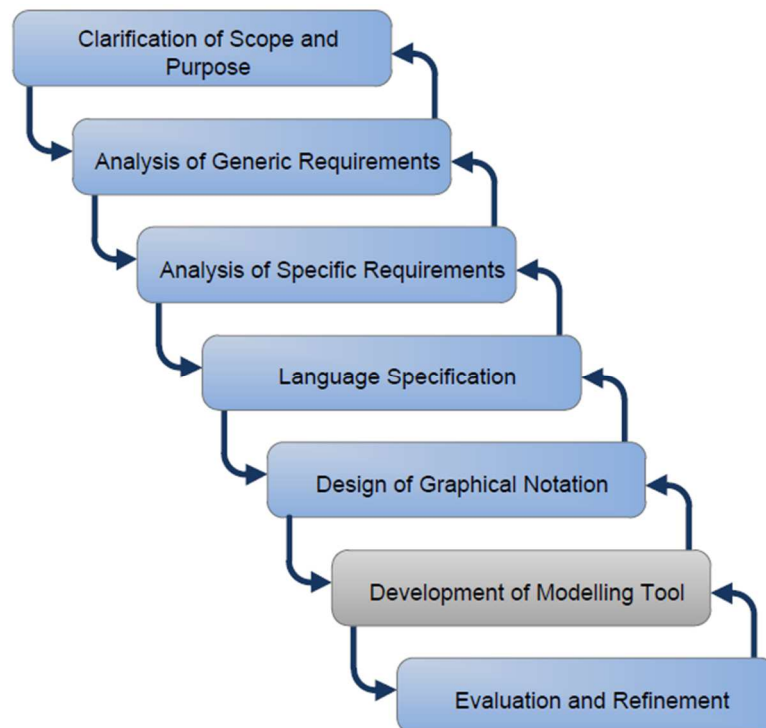


Figure 1.2 DSML implementation process by Ulrich Frank [17]

These steps are executed similarly to the iterative waterfall method, allowing feedback loops. Each step consists of an iterative micro-process to improve the process's simplicity and support the execution. However, 8 steps with micro-processes in each make it a complex process.

- The PhD thesis, *A Modelling Language for Rich Internet Applications* (2011) by Wright [42], produces a new DSML for RIAs named IAML. The research uses a process shown in Figure 1.3 to introduce the IAML.

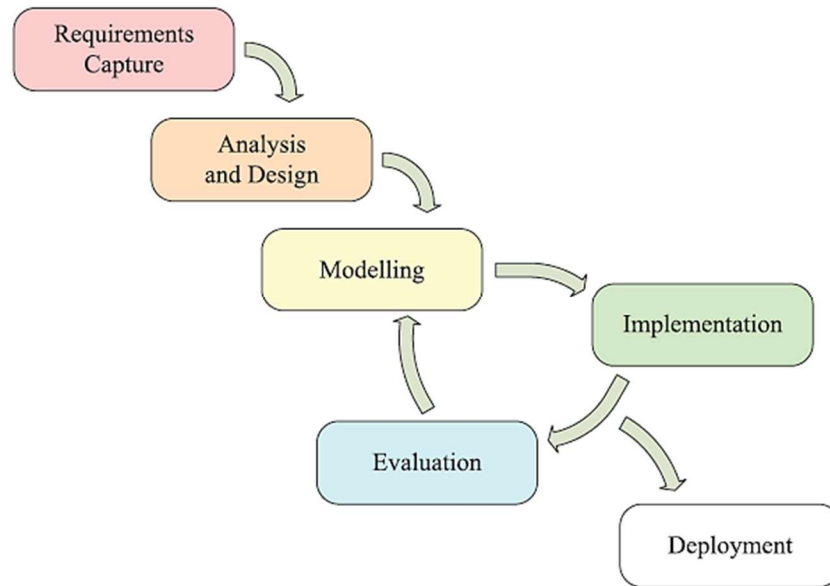


Figure 1.3 The hybrid modelling language process model used in the development of IAML [42]

Overall, this is a linear sequential process with a loop in the middle for modelling, implementation, and evaluation steps to iterate evolutionarily. The development loop is allocated to develop a CASE tool for the IAML. Not revisiting the requirements and analysis to update them may immensely limit the scope to enhance the quality of the results.

- Brambilla et al. (2012) [8] give a process with the 3 steps below for developing a DSML meta-model.
  - **Step 1. Modelling domain analysis:** set the context by identifying the target domain's purpose, realisation, and content.
  - **Step 2. Modelling language design:** define the formal syntax and models for the meta-model.
  - **Step 3. Modelling language validation:** demonstrate with use cases to validate the completeness and correctness.

This process is simple yet robust and focuses on all the aspects of introducing a DSML.

When looking at the literature which produces DSML, it can be understood that setting the context for a DSML by specifying the characteristics of the target type of systems is mandatory. Also, demonstrating the use of the produced DSML using examples is essential as a proof of concept and to explain the practical usage and value of the introduced DSML.

Inspired by the DSML implementing methods in literature and based on the methods utilised to produce the RiWAArch style [12], a design methodology implementation process is formed to implement the RiWAsDM, including the steps required to implement the RiWAsML, illustrated in Figure 1.4. This process contains 3 main steps that intend to assist in achieving research objectives 1, 2, and 3.

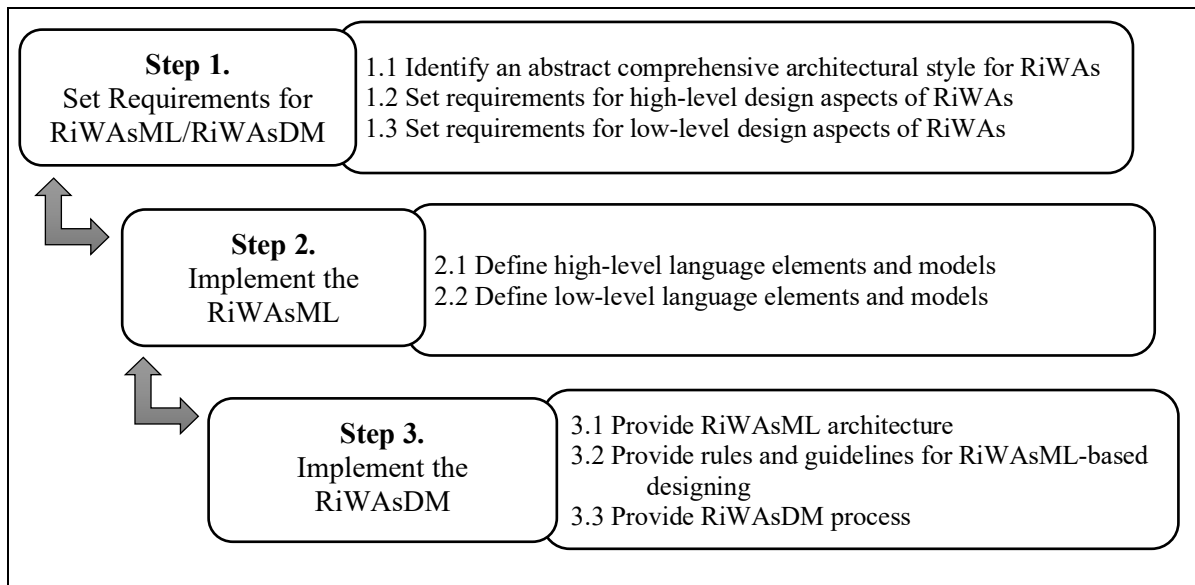


Figure 1.4 RiWAsDM implementation process

### 1.5.3.1. Step 1. Set Requirements for RiWAsML/RiWAsDM

Identifying and setting the context for a DSML is mandatory [26] [17] [8], thus, also for a design methodology. This step is allocated to set the context for the RiWAsML/RiWAsDM by identifying the requirements for a DSML within the research scope stated in Section 1.5.2 while covering research objectives 1 and 2.

Unlike the requirements for a software system, it is not straightforward to identify the requirements for a design methodology. The users of a design methodology are software engineers, and they might not exactly know what to expect from a comprehensive design methodology for RiWAs in general since their knowledge and experience can be limited to a specific type of RiWAs. First, it was decided to conduct a data survey to collect some data from the engineers in the domain of RiWAs and utilise the results to understand the requirements. However, after some discussion with three domain experts, it was noted that the knowledge and experience of the engineers are narrowed down to the work they are engaged with; therefore, a comprehensive overall understanding would not be gained. After that, to formalise the requirements for the RiWAsDM, it was decided to study the RiWAs designing related concepts through an extensive literature survey and combine the acquired knowledge with the experience of the author of this thesis, gained by researching in the domain over a decade. The requirements were defined step by step, as discussed below.

#### Step 1.1. Identify an Abstract Comprehensive Architectural Style for RiWAs

As the first step of this phase, it was required to identify the general characteristics and essential features of RiWAs to be addressed by the RiWAsML. The general characteristics and essential features of RiWAs are already understood [21] [12] (refer to Section 2.3.5), and learning the RiWAs' general formalism, which can realise these general characteristics and essential features, is required.

It was decided to identify an architectural style that can realise the RiWAs' general characteristics and essential features, relying on the benefits gained from an architectural style to realise the systems. Section 3.1 reviews the available styles for RiWAs and selects the RiWAArch style [12] as the foundation formalism for the RiWAsML/RiWAsDM. This step fulfils the research objective 1 (refer to Section 1.4).

### **Step 1.2. Set Requirements for High-level Design Aspects of RiWAs**

Using the RiWAArch style as a framework, the requirements for high-level design models and model-elements were identified and set in this step. These requirements explain the architectural details to be included in the RiWAs' high-level designs and propose the models and model-elements for them. Section 4.3 presents the requirements for high-level design aspects of RiWAs. This step partially achieves research objective 2, which is completed by step 1.3.

### **Step 1.3. Set Requirements for Low-level Design Aspects of RiWAs**

Based on the RiWAArch style and the requirements set for high-level design aspects in step 1.2, the requirements for low-level design aspects of RiWAs are identified and set in Section 4.4. The outcomes of this step, combined with the results of step 1.2, fulfil research objective 2.

#### **1.5.3.2. Step 2. Implement the RiWAsML**

This step is dedicated to implementing the RiWAsML in the direction of achieving research objective 2. The RiWAsML is UML-based; therefore, the models and model-elements are introduced as a UML extension.

#### **Step 2.1. Define High-level Language elements and Models**

RiWAsML's primary focus is the high-level structural modelling of the RiWAs. This step produces high-level structural design models, model-elements, and UML profiles to satisfy the requirements set in step 1.2 (refer to Chapter 5).

#### **Step 2.2. Define Low-level Language elements and Models**

This step implements low-level models, model-elements, and UML profiles for the RiWAsML to satisfy the requirements set in step 1.3, aligning with the high-level models and model-elements implemented in step 2.1. This step also produces some behaviour models for RiWAs to support the development of RiWAs. Chapter 6 discusses the outputs of this step.

### **1.5.3.3. Step 3. Implement the RiWAsDM**

This step focuses on implementing the RiWAsDM based on the RiWAsML produced in step 2. The implementation of the RiWAsDM completes research object 3.

#### **Step 3.1. Provide RiWAsML Architecture**

Understanding the architecture of a modelling language is crucial to gaining the maximum benefit from it. RiWAsDM discusses the RiWAsML architecture towards assisting in understanding the mapping between RiWAs high-level and low-level designs and development (refer to Section 7.2.1)

#### **Step 3.2. Provide Rules and Guidelines for RiWAsML-based Designing**

Rules and guidelines are essential to understand the proper utilisation of a modelling language to design the RiWAs accurately in the direction of supporting the development. Section 7.2.3 provides a set of rules and guidelines to be followed by the RiWAsML-based RiWAs designing.

#### **Step 3.3. Provide RiWAsDM Process**

Implementing a DSML would be advantageous only if its features can be integrated into actual RiWAs engineering. The RiWAsDM process is defined in this step to fill this gap and assist in integrating RiWAsML-based design activities into the agile SE environments. This process is given in Section 7.3, which discusses a design and engineering approach for RiWAsML and guidelines for adopting RiWAsML-based designing activities into agile SE methodologies.

### **1.5.3.4. Execution of the Process**

This process only focuses on implementing the RiWAsML/RiWAsDM, and the demonstration and evaluation methods are discussed separately. Even though the RiWAsDM implementation process is presented as a set of sequential steps, it was executed evolutionarily and iteratively, similar to the iterative waterfall model. The requirements were amended throughout the research, and the RiWAsML was upgraded accordingly. Even while working with the use cases, some more requirements were identified, and all the related sections of the thesis were appropriately updated.

### **1.5.4. Demonstrating the Adoption of the RiWAsML/RiWAsDM**

Introducing a conceptual design methodology would not benefit RiWA engineering unless the utilisation of the introduced methodology is demonstrated and discussed with examples. Literature producing the DSML utilises use cases as proof of concepts to evaluate the new DSML and demonstrate its use. However, these use cases are hypothetical, and scenarios are mostly limited in scope to demonstrating a particular model or a set of related models instead of using a comprehensive scenario to demonstrate the complete DSML [43] [44] [45] [46] [47]. IAML [42] provides a complete demonstration of system designing based on a hypothetical scenario.

This thesis uses real-world use cases to demonstrate the adoptability of the RiWAsML/RiWAsDM in Chapter 8, achieving research objective 4. Two actual use cases are employed to discuss the RiWAsML’s attributes based on the high-level models and model-elements. Then, a comprehensive real-world use case is utilised to demonstrate all the features of the RiWAsML/RiWAsDM, discuss how the designed functions are developed in reality, and discuss the adoption of design activities into an agile environment.

### 1.5.5. Evaluation

Evaluating a design methodology is highly challenging as it is a conceptual artefact and straightforward evaluation methods for design methodologies are unavailable. DSML implementing literature evaluates the DSML primarily with use cases as proof of concept.

Frank provides a process with 6 steps to evaluate DSML, given in Figure 1.5 [17]. This process is supposed to be executed by the participants: Domain Expert, User, Language Designer, and Tool Expert. This process is likely to be exploited to evaluate a design methodology. Nevertheless, this process only focuses on the use-case-based evaluation and may not produce convincing results without comparing the new DSML against the available ones.

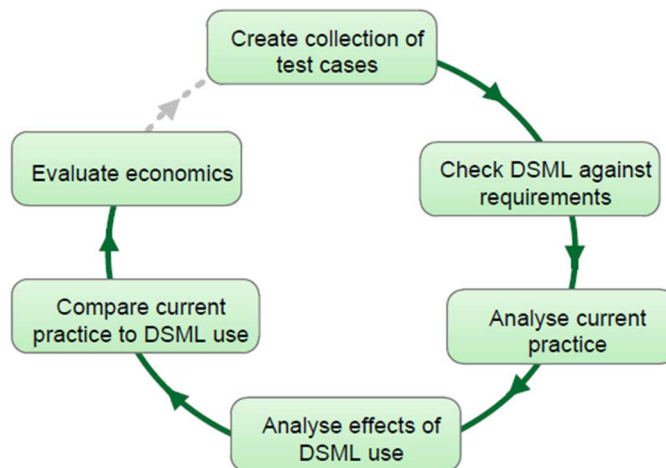


Figure 1.5 Micro Process ‘Evaluation and Revision’ of DSML [17]

Wright [42] uses metrics to compare the IAML against the available DSMLs for RIAs. The use of metrics to evaluate a DSML is a controversial topic. A DSML is a conceptual artefact, and its application as a design tool can be highly subjective. Even though metrics can be used to measure some aspects of a DSML, the use of metrics to compare DSML is questionable because of the following reasons. A new DSML is introduced since the available DSMLs do not cater to the target systems. If the available DSMLs are identified as not being helpful in designing the target systems, why use metrics to measure their inability to value the new DSML? Further, it is difficult to obtain a proper idea of the level of strengths or weaknesses by examining the quantitative values of the

metrics. Due to these aspects, it isn't easy to justify using metrics to evaluate the DSMLs or a design methodology.

Since evaluating a conceptual implementation like a design methodology highlighting its strengths is not straightforward, this thesis exploits multiple qualitative evaluation methods to identify the strengths and weaknesses of the RiWAsDM based on the evaluation of the RiWAArch style [12], inspired by the work of Fielding [19]. A self-evaluation, contextualised comparisons, and expert evaluation were performed, and the results were triangulated by reasoning to draw the ultimate outcomes. The evaluation of the RiWAsDM is given in Chapter 9, achieving research objective 5.

#### 1.5.5.1. Self-Evaluation

A qualitative self-evaluation is used to analyse and identify the strengths and weaknesses of the RiWAsDM against the requirements. Also, the satisfaction level of the requirements within the context is assessed. In the self-evaluation, all the features of the RiWAsDM are evaluated against the requirements one by one, by discussing their contributions to satisfying the requirements based on the use cases. The effectiveness of a particular feature against a specific requirement is indicated using the scale given in Table 1.3.

*Table 1.3 Self-evaluation scale*

Symbol	Interpretation
++	Very high effect
+	High effect
-+	Moderate effect
-	Less effect
--	Very low or no effect
NA	Not/None Applicable

The ultimate results are drawn by considering the cumulative effect of the RiWAsDM's features under each requirement. Section 9.1 contains the results of the self-evaluation.

#### 1.5.5.2. Contextualised Comparisons

Contextualised comparison is a comparative analysis [48] [49], which is a qualitative evaluation method, and it has been utilised by conceptual SE research working with qualitative attributes [19] [41] [12] [50]. This thesis performs contextualised comparisons to compare the RiWAsDM against the available solutions to reflect its advancements and limitations. The context is based on the requirements set for the RiWAsDM. The degree of satisfaction of the requirements identified in the self-evaluation is exploited in the contextualised comparison when judging the RiWAsDM against the available methods/methodologies. The same scale given in Table 1.3 is used to indicate the



comparative effect of the solutions, including the RiWAsDM. The results of the contextualised comparison are discussed in Section 9.2.

### 1.5.5.3. Expert Evaluation

Renmans and Pleguezuelo [51] show that qualitative methods are the most commonly used in realist evaluations, and 97% of all realist evaluations use interviews. Frank [17] suggests that experts should participate in evaluating the DSMLs. Domain experts can contribute to assessing a concept with their knowledge and experience gained while working in the target domain. A qualitative expert evaluation has been conducted to get the opinions of the domain experts based on their knowledge and experience, and the results are given in Section 9.3. Some aspects related to the expert evaluation are discussed below.

#### Expert Evaluation Medium

This is a document-based evaluation. A document was created, including all the related details for the experts to refer to and an evaluation form to fill out. A document-based method is selected to provide the experts with all the relevant information, allow them to go through everything at their own pace, study the RiWAsDM before attempting to fill out the evaluation form, and justify their evaluation based on their expertise [52].

A pilot evaluation was conducted with some students to identify the concerns of the document and finalise it. These students are 2<sup>nd</sup>-year undergraduate students who completed a UML-based system designing module. They were given the document by the semester's end after completing the said module. They found that the RiWAsML is easy to use, and the main concern was not having enough details of the RiWAsML to refer to when required. The RiWAsML language reference (refer to Section 7.2.2) was improved based on their feedback and included in the expert evaluation form. The finalized expert evaluation document is given in Appendix D.1.

The evaluation document provides a use case and asks for the experts' feedback for each requirement. They have to examine the use of RiWAsML/RiWAsDM in the use case and rate the satisfaction of the requirements using the scale in Table 1.4.

*Table 1.4 Expert evaluation feedback scale*

Symbol	Interpretation
5	Very high effect
4	High effect
3	Moderate effect
2	Less effect
1	Very low or no effect

Since this rating can be subjective and biased, the experts are asked to justify it using natural language to gain insights from their expertise.

The main issue of the document, which was identified while executing the evaluation, is that the context and its scope were not mentioned. It allowed the experts to think outside of the scope and feedback. However, it helped validate this research's identified limitations and future work, which are discussed in Sections 10.4 and 10.5.

### **Selection of Experts**

The experts were selected from the following two categories.

1. Industry experts with strong hands-on experience in engineering the RiWAs.
2. Academic experts with a deep understanding of the RiWAs design and development concepts.

Multiple experts from the use case projects and universities were contacted with the evaluation document, and the ones who agreed to take part in the evaluation were identified. It was decided to initiate the expert evaluation with two experts from each category. If there were disputes or conflicting ideas between the feedback on a particular criterion, the evaluation should be extended to more experts until the researcher is saturated [53] [54]. Since there were no disputes or conflicts, the expert evaluation was finalised with feedback from two experts from each category.

### **Execution of the Expert Evaluation**

The evaluation document was mailed to the selected experts, who were assigned 2 weeks to complete the evaluation. They were expected to study the materials by themselves and attempt the initial review to control the assessment of the quality attributes: learnability, readability, and understandability. They were asked to contact the researcher when they required assistance understanding the materials.

Once they submitted the evaluation document with the filled form, the researcher carefully examined the form to identify unclear or inconsistent points. The points with issues were commented on with details and sent back to the expert to answer with clarifications. This step was repeated until both the researcher and the expert were satisfied and agreed with the results.

Once feedback from all the experts was gathered and verified, the analysis started. Under each requirement of the RiWAsDM, all the participating experts' feedback was examined, and the effects were recorded using the same scale given in Table 1.3 in Section 1.5.5.1. The cumulative effect on a particular requirement was decided by calculating the averages of the ranks under each requirement and analysing the feedback of all the experts under that requirement. This was repeated for all the requirements to obtain the ultimate results as given in Section 9.3.

## 1.6. The Solution and Research Contributions

This research introduces a novel design methodology for the RiWAs, named the RiWAsDM, to fulfil the research aim. The contributions of the thesis to the domain of RiWAs designing and UML-based designing are listed below, **which are detailed in section 10.2.**

1. A design methodology implementation process.
2. A new RiWAs modelling language named **RiWAsML** offers unique features such as controller and DB-bus modelling.
3. A new UML extension for the RiWAsML.
4. A new design methodology named **RiWAsDM** for the RiWAs.
5. Demonstration of the use of the RiWAsML/RiWAsDM through a real-world use case.

## 1.7. Structure of the Thesis

An overview of the thesis structure is given below to help understand the flow of the discussions. Additionally, at the beginning of each chapter, an introduction is given, explaining the chapter's intention and the main points covered; at the end of each chapter, a summary is presented, highlighting the essential outputs of the chapter.

**Chapter 2. Background:** This chapter discusses the background concepts to provide a foundation for the rest of the thesis, highlighting the position of systems designing within software engineering and RiWAs.

**Chapter 3. Review of Available Related Solutions:** This chapter reviews the available solutions for RiWAs designing under 4 sections: architectural style for RiWAs, high-level design methods/methodologies, low-level design methods/methodologies, and research work related to RiWAs designing.

**Chapter 4. Requirements for RiWAsML and RiWAsDM:** This chapter sets the requirements for the RiWAsML/RiWAsDM based on the RiWAArch style, which realises the general characteristics and essential features of RiWAs.

**Chapter 5. RiWAsML: High-level Modelling Language:** This chapter introduces RiWAsML's high-level models, model-elements, and UML profiles to design the RiWAs' architectural aspects.

**Chapter 6. RiWAsML: Low-level Modelling Language:** This chapter introduces RiWAsML's low-level models, model-elements, and UML profiles to design the RiWAs' development-level aspects.

**Chapter 7. Rich Web-based Applications Design Methodology:** Based on the RiWAsML introduced in chapters 5 and 6, this chapter implements the RiWAsDM.

**Chapter 8. Use Case:** The adoption of the RiWAsML/RiWAsDM into practical RiWA engineering is demonstrated in this chapter using real-world use cases as proof of concept.

**Chapter 9. Evaluation:** This chapter evaluates the RiWAsDM using multiple methods and triangulating their results.

**Chapter 10. Conclusion:** This chapter concludes the thesis by showing the achievements of the research objectives, proving the hypotheses, and discussing the fulfilment of the research aim. Also, the research contributions, challenges, limitations, and future work are discussed.

## Chapter 2. Background

---

This chapter gathers knowledge related to the primary concepts of this research – Software Engineering and Software Design, and Rich Web-based Applications (RiWAs) – and details them towards building a solid framework for the rest of the thesis. While conducting the literature survey, it was noted that the terminologies and the definitions of the related concepts are not strongly specified, and experts discuss and interpret them differently; also, the relationships between these concepts are not clearly defined. Therefore, it is not easy to understand the position of these concepts and utilise them in discussions. This chapter sets the terminology for the rest of the thesis, defining them and highlighting the relationships among them in the direction of straightforward referencing throughout the thesis.

### 2.1. Related Software Engineering Concepts

The use of the term *Software Engineering* (SE) runs back to the 1960s [55] [56], which represents the discipline of engineering software systems, covering all the related artefacts such as SE methods and guidelines. Since then, over time, the domain of SE has grown rapidly, introducing numerous concepts, including methodologies, standards, protocols, frameworks, etc., which are continuously evolving. The interrelationships of these SE concepts are highly complex, where the terminologies used to discuss them are not firmly defined (or defined in different ways by many) and, thus, unclear.

The section's intention is not to discuss what SE means; instead, it discusses the concepts related to this research within the domain of SE towards setting a firm foundation for indicating the place for the software design process within it while specifying a taxonomic arrangement of the related concepts like SE methodologies and approaches, for easier reference within the scope of this research. This section first discusses the terms *method* and *methodology*, setting definitions to form a context for this thesis. Then, SE approaches are discussed towards appreciating the use of software designing activities to motivate the title of this thesis.

The terms *method*, *model*, and *methodology* are controversial and often interchangeably exploited by many. Since this thesis introduces a new design methodology, a strong notion of these terms is essential. This thesis defines them as follows in the direction of framing the concept of methodology to set a context for the rest of the discussions.

#### 2.1.1. Software Engineering Methods

SE method can be defined as “*a systematic procedure or technique of doing work in software engineering in order to reach a certain goal and/or produce a defined set of software artefacts*” [2] (page 414), which is dedicated to a particular type of activity like development or testing. A method may specify a particular tool, like a model, framework, or environment/platform, which is used to

accomplish the target activity. There are many methods associated with different phases of SE; the following can be given as some examples in two distinct levels.

- Global methods, which apply to the entire engineering project
  - Software Development Life Cycle (SDLC) models
  - Documentation standards
  - Project management methods
- Phase specific methods
  - **Requirements phase:** requirement-gathering methods like surveys and observations
  - **Design phase:** architectural styles, design patterns, structured design, and Object-Oriented design
  - **Development phase:** Frameworks/languages and libraries
  - **Testing phase:** black-box-testing and white-box-testing
  - **Implementation phase:** virtualization and cloud services
  - Any other phase-specific methods like tools, standards, techniques, rules, practices, policies, etc.

Based on these aspects, this thesis defines the SE method as follows.

SE method is a concept or tool that supports a specific task at the global or particular phase level.

### 2.1.2. Software Engineering Methodology

The software engineering methodologies cover all the aspects of the SE projects, including the SDLC model, diverse methods for different phases and activities, rules and guidelines, and standards. Some of the available definitions are given below.

**Software Development Glossary: 88 Essential Terms** by Anna Peck, a senior SEO specialist at *Clutch.co* [57], defines methodology as a “*technique that enables the design and development of software to be implemented.*”

**Mnkandla** [58] defines the software methodology by surveying multiple definitions as “*a group of methodologies used in the development of applications.*” Mnkandla extends that the “*methodologies will give details of what should be done in each phase of the software development process. You will notice that the methodologies do not necessarily specify how things should be done. That level of detail is usually left to the organization to tune the methodology to its environment by for example developing templates, and other documents that spell out how things should be done*” [58]. Disagreeing with stipulating the methodology as a framework, Mnkandla states that “*the major difference between a methodology and a framework is that a framework should be used at a more abstract level, which means you will need a methodology or more in order to implement a*

*framework*” [58]. Mnkandla’s notion of the methodology is more like a method where the author uses the term framework in the level of methodology; however, the idea is unclear.

**Laplante** [18] says that a methodology “*identifies how to perform activities for each period, how to represent the activities and products, and how to generate products.*”

**Pressman** [59] explains a methodology is a process with a collection of techniques for accomplishing a specific software development goal, including a process model.

This thesis defines the SE methodology at a global level of engineering projects as follows, in the context of this research, by considering the available definitions and explanations.

Software engineering methodology at the global level is a framework that governs SE projects by defining a set of methods selected based on some specific criteria. These methods systematically explain how tasks in different phases are executed within an SE project. The following are some decent examples of SE methodologies: rapid development methodologies such as *Scrum* [60] [61], *DevOps* [3], and *Continuous Integration - Continuous Delivery* (CI-CD) [4].

There are phase/task-specific methodologies as well, which specify a collection of methods to be utilised within the phase/task. For example, *PRINCE2 Methodology* [62] is a structured project management methodology focusing only on project management and related aspects. Likewise, *software design methodologies* provide a dedicated set of methods that pay attention to the design phase activities within the SDLC, which emerged in the 1960s as an independent scientific discipline [63]. This research focuses on the concept of design methodology, which is further discussed in section 2.2, and related design methods/methodologies are reviewed in chapter 3.

### 2.1.3. Software Engineering Approaches

This thesis exploits the SE approaches to classify the SE methodologies based on their significance for the design activities since this research concentrates on the software designing aspects. The SE methodologies can be classified into two main significant contemporary approaches as stated below, appreciating the attention given to software designing activities.

1. *Model-Driven Software Engineering (MDSE)*:– also called *Model-based Software Engineering* (MBSE), or just *Model-Driven Engineering* (MDE) or *Model-based Engineering* (MBE).
2. *Agile Software Engineering (ASE)*.

The MDSE is based on software design aspects, proposing to spend considerable time and effort on system designing. The strength of the MDSE has been identified, and it has served well for decades [18] [8] [64]; therefore, in-depth discussions are avoided in this thesis. Opposed to MDSE, the ASE is based on the *Agile Manifesto* [5], which stresses delivering “*working software over comprehensive*

documentation” [5], where this documentation includes the design documents. ASE is compared with the MDSE, and its advantages are discussed by many academic publications as well as online articles [65] [66] [67] [68] [69]; hence, this thesis avoids discussing them further.

Different types of MDSE approaches are available, and they are separated based on the attention given to different design aspects of software systems, as stated below [64].

1. *Basic Model Driven Architecture (MDA)*: this emphasizes the importance of extensive designing of the entire system.
2. *Feature-Oriented Model-Driven Development (FOMDD)*: this attempts to create models with predefined parts to compose the complete system.
3. *Agile Model-Driven Development (AMDD)*: focuses on modelling just enough to continue with the development. This approach inherits features from both MDA and ASE.

The classification discussed above is illustrated in Figure 2.2.

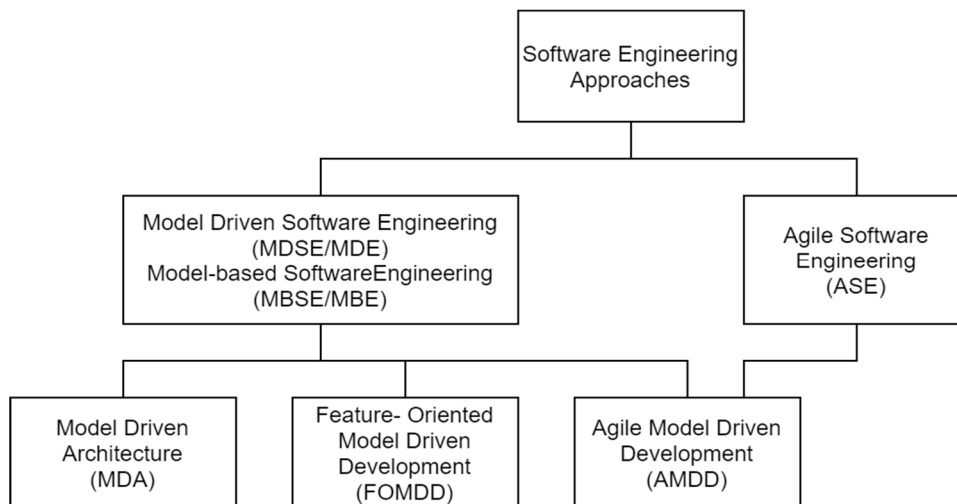


Figure 2.1 Software engineering approaches

When conversing about the ASE, there is an impression that the design aspects are entirely ignored, and the engineering should focus only on the development. The agile manifesto never restricts modelling; furthermore, the principles behind the agile manifesto also specify that “*the best architectures, requirements, and designs emerge from self-organizing teams*” [5] [70]; therefore, we can argue that agile methodologies can also benefit from software modelling. The domain experts also state the necessity of software modelling, even with the ASE methodologies [71] [72]. Based on these thoughts, this thesis emphasizes and concludes that, regardless of the SE approach, software designing/modelling still plays an important role; thus, the software design methods and methodologies are significantly needful even in agile SE environments [30]. The AMDD is an approach inheriting features from both the MDSE and ASE, which is the focus of this thesis.



## 2.2. Software Design Methodologies and Methods

This section discusses the concepts related to **software designing**, which is also called **software modelling**, in the direction of providing the necessary knowledge to the targeted concept, the RiWAs design. According to the IEEE definitions [6], software designing is “*the process of defining the architecture, components, interfaces, and other characteristics of a system.*” Software designing plays a vital role in SE, providing a blueprint for implementing software systems [73] [74], regardless of the SE approach, as discussed in section 2.1.3. The following sections discuss the design methodologies, design process, design approaches, modelling languages, and associated concepts. The knowledge delivered by this section is essential for discussing and introducing a design methodology for RiWAs.

### 2.2.1. Software Design Methodologies

A **software design method** can be considered a concept or tool that assists in performing a software design activity, such as modelling a particular aspect or function of a software system and documenting the designs. Modelling languages, CASE tools, and design approaches are design methods that offer knowledge and resources to the designers to perform design activities. Designing is a creative process; hence, design methods cannot provide exact instructions for designing for a specific problem [20]; instead, they offer guidelines to follow and discuss the utilisation of the methods.

Dedicated methodologies that overlook the software design aspects are called **software design methodologies**. The designing activities are controlled by two main aspects: (1) the SDLC model, which is under the supervision of the SE methodology and approach, and (2) the software design methodology. The software design phase within the SLDC typically follows the requirements phase and followed by the development phase. Once the requirements are gathered, the software system can be designed using an appropriate design methodology and then developed based on the design.

Selecting a proper design methodology is vital to improve productivity and accuracy. The number of iterations and the amount of work in each iteration depend on the SDLC model used by the SE methodology and approach. The linear SDLC models are usually associated with Model-Driven Engineering, where the entire system is first designed and then developed. Contrarily, the iterative SDLC models are exploited mainly by the agile SE approaches, in which a system is developed part by part, where each piece is designed and developed in separate iterations, allowing incremental system development. However, as concluded in section 2.1.3, regardless of the SE approach, SE methodology, or the SDLC model used, software designing has marked its place as necessary. Therefore, identifying a proper design methodology is highly useful to get the maximum out of software modelling since an appropriate software design methodology can guide the designing process towards improving the efficiency and productivity of the SE project.

Naturally, the software design and development concepts, such as design patterns and algorithms, are discussed together, as the development activities are highly associated with and directly affected by the software design. This thesis identifies two major paradigms to discuss software design and development methods. The earlier one is the *Structured System Analysis and Design (SSAD)*, which is based on the procedural/structural development disciplines, and the latter one is the *Object-Oriented Design and Development (OODD)* paradigm, based on the Object-Oriented concepts. Since the OODD has become the de facto standard in SE, this thesis limits the research to focus on the OODD-based design methods/methodologies and techniques.

Identified design methods/methodologies for web engineering are reviewed in Chapter 3; hence, they are not discussed in this section. Instead, the following sections discuss needful related concepts to gain the knowledge required when introducing the proposed RiWAsDM.

### 2.2.2. Software Design Process

The software design process can basically be divided into the **preliminary design (high-level design)** and **detailed design (low-level design)**, which are discussed in detail in the sub-sections below, also stating the related artefacts. The proposed RiWAsML is discussed based on these levels in chapters 5 and 6.

#### 2.2.2.1. High-Level Design (Preliminary Design / Software Architectural Design)

Preliminary design is “*the process of analyzing design alternatives and defining the architecture, components, interfaces, and timing and sizing estimates for a system or component*”, where architectural designing is “*the process of defining a collection of hardware and software components and their interfaces to establish the framework for the development of a system*” [6]. Based on this notion, this thesis considers the preliminary high-level design as the **software architectural design**. Some literature refers to software designing as software architecting in general; for example, model-driven architecture (MDA) reflects that in systems architecting, “*models can represent systems at any level of abstraction or from different viewpoints*” [75]. In contrast to referring to software designing as software architecting, this thesis uses the term *architectural design* only to denote the high-level design.

#### Software Architecture

The software architecture (or technically the architectural design) provides an overall abstract picture of the architectural elements and their relationships within a system at its run time, assisting in realising the system [19]. Architecture is the foundation of any software system, and the support gained from a carefully designed sound architecture is significant throughout all the phases of SE projects. The increased realisation helps reduce complexity [76] since the meaning of software complexity encloses the difficulties in understanding [27].

### Software Architectural- Elements

Fielding [19] explains that the software architecture is defined by a **Configuration** of the elements: **Components**, **Connectors**, and **Data**. The **Constraints** on the relationships between these elements help to achieve the desired set of **Architectural Properties**. His definitions of these elements are as follows: “A **Component** is an abstract unit of software instructions and internal state that provides a transformation of data via its interface”; “A **Connector** is an abstract mechanism that mediates communication, coordination, or cooperation among components”; and “A **Datum** is an element of information that is transferred from a component, or received by a component, via a connector.” Fielding defines **Configuration** as “the structure of architectural relationships among components, connectors, and data during a period of system run-time.” Fielding explains architectural properties as “the set of **Architectural Properties** of a software architecture includes all properties that derive from the selection and arrangement of components, connectors, and data within the system. Properties are induced by the set of constraints within an architecture.” Referring to Ghezzi *et al.* Fielding’s notion about constraints is that “**Constraints** are often motivated by the application of a software engineering principle to an aspect of the architectural elements.”

### Architectural Styles

Architectural styles offer a framework for designing system architectures [77]. As per David Garlan, “an architectural style typically specifies a design vocabulary, constraints on how that vocabulary is used, and semantic assumptions about that vocabulary” [78]. This thesis attempts to identify an architectural style for RiWAs (refer to research objective 1 in Section 1.4) and introduce the DSML for RiWAs, named the RiWAsML, based on the identified style (refer to research objective 2 in Section 1.4).

### Architectural Description Language

A design methodology should include tools for designing system architectures as well. *Architectural Description Languages (ADLs)* provide notations and tools to design and draw software architectures. These ADLs can be categorised into three distinct approaches, as discussed below. The available ADLs are reviewed in Section 3.2.

1. **Informal box-and-line drawing**: this informal technique is an early practice of drawing software architectures using boxes and lines [78], where boxes denote the elements and lines are used to illustrate the communication channels between the boxes. Some available methods, like the C4 model [79] and Arc<sup>42</sup> [80], use this approach. This method is simple without proper syntax but incorporates considerable usability issues without formal language elements and semantics. Formal languages with models, model-elements, rules, and guidelines are introduced to overcome the problems of this method. Refer to Section 3.2.1 for the review of the box-and-line approach.

2. **Formal ADLs:** A set of more formal languages, such as AADL [70], are available, which are generally referred to as ADLs in this thesis. These formal ADLs follow standards like *ISO/IEC/IEEE 42010:2011, Systems and software engineering — Architecture description* [81]. Formal ADLs use a coding approach to generate the system architectures. Refer to Section 3.2.2 for the review of the available ADLs.
3. **Semi-formal graphical languages:** Graphical languages provide models and model-elements to design system architectures. UML is a graphical GPML for software design, and some UML-based languages, such as TAM [82], have been introduced to cater to the specificity of designing architectures. Refer to Section 3.2.3 for the review of the available high-level UML-based methods/methodologies.

#### 2.2.2.2. Low-Level Design: Detailed Design of System Components

Low-level design (or detailed design) is defined as: *“the process of refining and expanding the preliminary design of a system or component to the extent that the design is sufficiently complete to be implemented”* [6]. The low-level design of a software system describes the internal elements of the components, connectors, and even offers designs of data structures, providing fine-grained details to develop these elements. Different aspects, such as algorithms, components, databases, and networks, use distinct modelling methods. A design methodology needs to provide enough tools to assist with low-level design aspects of a system. This thesis considers the lowest-level design as the **development-level**, which helps map the design elements to their development by providing enough development-supportive details (refer to Section 7.2.1).

#### 2.2.3. Software Design Approaches

There are two approaches to designing software, which are briefly discussed below.

1. **The top-down approach:** the top-down approach starts with the high-level architectural design, *“and then a process of decomposition begins to work downward toward more detail. The starting point is the highest level of abstraction. As decomposition progresses, the design becomes more detailed until the component level is reached”* [83].
2. **The bottom-up approach:** the bottom-up approach begins with the low-level design components *“that are needed for the solution, and then the design works upward into higher levels of abstraction. Various components can then be used together, like building blocks, to create other components and, eventually, larger structures. The process continues until all the requirements have been met”* [83].

Further details – including the pros and cons of these approaches and the methods and tools used – are not discussed here, considering they are out of the scope of this study.

#### 2.2.4. Software Modelling Languages

In software designing, various methods are utilised to model different aspects of the software systems; for example, flowcharts are used to design algorithms, entity-relationship (ER) diagrams are used to model databases, and UML use case diagrams are used to model requirements. These methods use distinct modelling languages to design the target aspects. *General-Purpose Modelling Language (GPML)* is a language that provides modelling elements to software systems in general, without specifically focusing on a type of system such as web applications or mobile applications [8]. The models and model-elements of the GPMLs may provide tools to design some general aspects of any system; for example, UML's class diagram can be used to design the domain logic of many types of software systems, including desktop, web, and mobile apps. However, the GPMLs fail to address the specificity of different types of software systems; for example, UML does not provide tools to design GUI-related aspects, which are essential for many modern apps, including web and mobile apps. To design unique features specific to a domain, engineers may use a *Domain-Specific Modelling Language (DSML)*, which offers enough tools to design the essential features of the systems in that particular domain [8]. Identified DSMLs for the domain of web-based applications are reviewed in Chapter 3.

##### 2.2.4.1. Modelling Language- Elements

A **model** is an abstract presentation of a particular aspect of a software system made of **model-elements** [8]. The modelling languages provide models (structure) and model-elements for them, with **syntax** (notations with rules) and **semantics** (meaning) to design different aspects of a software system [8]. A single model expresses a selected set of features or a particular view of a software system, and a modelling language may provide multiple models and their model-elements to represent different aspects of a software.

A system is designed by drawing **diagrams** based on the models using the given syntax and rules of the model's elements, ensuring that the requirements are expressed semantically. Some languages like UML also refer to the design diagrams as models; nonetheless, this thesis refers to the user designs based on language models as diagrams. A design diagram may contain textual descriptions as well. Design models and diagrams are paramount for understanding, documenting, and sharing knowledge about complex software systems [8].

##### 2.2.4.2. Modelling Language Attributes

There are attributes to express the validity of modelling languages, such as simplicity, consistency, completeness, extensibility, scalability, readability, understanding, correctness, generality, and power of integration [17] [8]. The attributes selected for the RiWAsML/RiWAsDM – the simplicity and adoptability – are generally discussed below, and their context is set in Section 4.1 under the requirements for RiWAsML/RiWAsDM.

1. **Simplicity** – A modelling language should be less complex to assist in understanding the systems designed using it straightforwardly [18] [20]. Simplicity can be preserved by incorporating the *separation of concerns* principle, which appreciates decomposing a system and identifying and separating the modules for greater realisation, thus, management [19]. A DSML must identify enough models and model-elements to design all the general characteristics of the target systems, maintaining a higher simplicity [17].
2. **Adoptability** – If a modelling language provides enough assistance in designing the target systems, it will be highly adopted into engineering as a quality tool without being limited to a theoretical solution. The following attributes are selected to validate the adoptability, which looks into various dimensions of assistance a language provides to improve the adoptability.
  - 2.1. **Comprehensiveness** – A modelling language has to provide enough elements to construct expressive yet intuitive models [84]. A modelling language can be considered comprehensive when it gives the following: 1.) models and model-elements to design all the general aspects of target systems, 2.) rules and guidelines for designing the target systems using the language and mapping the designs into development, and 3.) design and engineering approaches to follow and guidelines to adopt into agile SE environments [17].
  - 2.2. **Usability** (Learnability and Readability/Understandability) – Usability is an essential attribute for the new DSMLs [17]. The models and model-elements provided by modelling language must be expressive yet intuitive and easy to learn [84]. UML is a well-known design language, and if a DSML is based on UML, it can be considered easy for engineers with UML knowledge to learn and use. Also, suppose the low-level models and model-elements of a DSML are more related to the actual development of the target systems; in that case, the engineers can quickly learn the language as they may have the development experience and knowledge. When a language syntax contains many development-oriented details, and the models provide much relevant information, the designs become more readable and understandable. A highly readable, understandable, and learnable language can be seen as a significantly usable and, thus, adoptable tool.
  - 2.3. **Development support** – The primary need of a modelling language is to design a system to support its development. Even though a design is conceptually good but cannot be converted into actual development, the conceptual support provided by the language is not of much value. A language must give enough assistance to map the designs into development to consider it a worthy tool to adopt into engineering.
  - 2.4. **Integrability** – performing design activities in an agile SE environment is challenging as agile methodologies prioritize development over design and documentation. If a design language/methodology guides how to integrate the design activities into agile SE methodologies, resulting in AMDD, it will adequately assist in adopting the language/methodology into engineering.

### 2.2.4.3. OMG Meta-modelling Process

To define a new modelling language, *Object Management Group (OMG)* provides the *Meta Object Facility (MOF)* specification with the meta-modelling process [85], which the language specification is referred to as a **meta-model**. This process is based on a 4 layered meta-model hierarchy shown in Figure 2.3 [85].

- **Layer M3 (the meta-meta-model):** The M3 layer defines the language that can be used to describe new modelling languages. The M3 language can be seen as a language of modelling languages for specifying the M2 meta-models. The MOF is the language used in the OMG's meta-modelling process M3 [85].

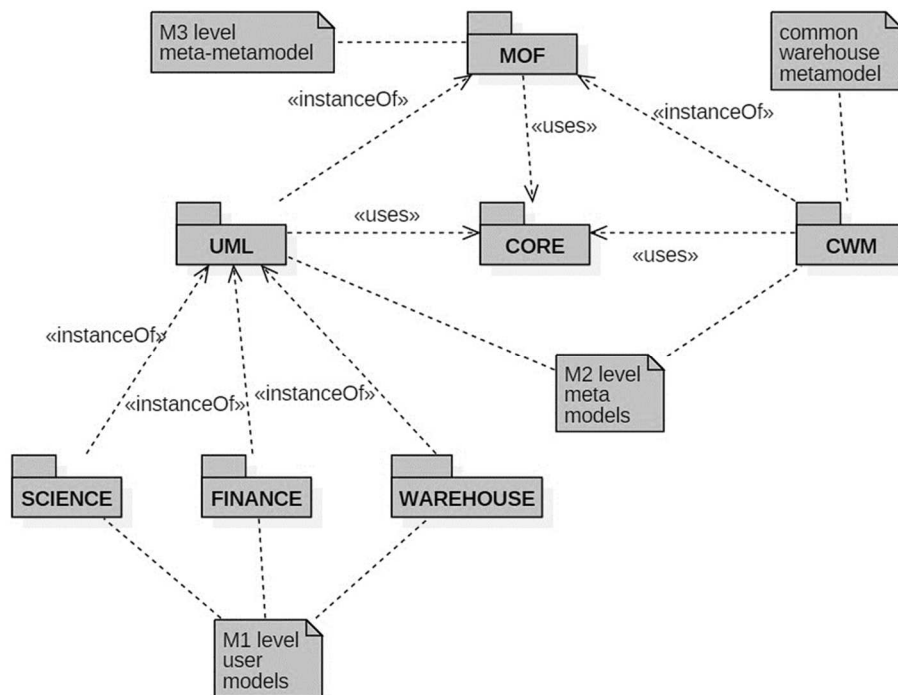


Figure 2.2 OMG's meta-model hierarchy [86]

- **Layer M2 (the meta-model):** This layer defines the modelling languages which provide meta-models with syntax for designing software. UML is an example of the M2 meta-model defined by the MOF (refer to Section 3.2.3.1). DSMLs, defined as UML extensions, are also considered M2 meta-models.
- **Layer M1 (user models):** The elements of this layer are the user models constructed based on an M2 meta-model. This thesis refers to these models as **diagrams**.
- **Layer M0 (instance models):** This layer explains the running system, where its elements are the actual runtime instances within the system. The M1 user models define a system's particular elements and their configuration at the M0 level.

In order to support integrating the modelling languages into the CASE tools, OMG offers an XML-based format named *XML Metadata Interchange (XMI)* [87] [88]. Further, OMG provides a process

to submit proposals for new specifications, which the new DSML can get approved by the OMG as a standard [89].

#### 2.2.4.4. Define Domain-Specific Modelling Language

A GPML like UML cannot assist in designing the domain-specific features of a software system and requires DSML to cater to the specificity of the target system's context [9]. If a DSML is unavailable for any specific domain, one can introduce a new DSML. There are two main approaches to defining a DSML, following the OMG as the international standards provider [90].

1. One approach to defining a DSML is using OMG's MOF [85] and introducing a new meta-model like UML.
2. The other approach is to use the UML's **extension mechanism** to introduce a UML-based meta-model as a set of new UML profiles.

The advantages of implementing a DSML as a UML extension – over introducing a new meta-model using the MOF – have been recognized towards higher adoptability [9] [91]. UML profiles respect the UML semantics and, therefore, are intuitive for designers, providing a lower learning curve, and the available UML tools can highly support UML profiles. These facts highlight the benefits of using UML profiles and undoubtedly outweigh if there are any limitations [9] [91] [8]. Considering these benefits, this research opts for the UML extension over a new MOF-based DSML to introduce the RiWAsML.

#### 2.2.4.5. UML Extensions and Profiles

As mentioned above, new DSMLs can be defined as UML profiles using UML's lightweight extension mechanism. This section elaborates on the elements of the UML's extension mechanism.

- A **metaclass** is a class/component/element in a **meta-model** like UML, which can be extended by one or more **stereotypes**.
- The **Stereotype** is a class that defines how an existing **metaclass** is extended as a part of a **profile** [92].
- **Constraints** are associated with the stereotypes, imposing restrictions on the corresponding meta-model-elements – such as pre-and post-conditions of operations, invariants, derivation rules for attributes and associations, the body of query operations, etc. – which can be expressed using either a natural language or the formal *Object Constraint Language* (OCL) [9].
- Properties of a stereotype are referred to as **tag definitions** or **metaproperties**, where the values of these tag definitions are referred to as **tagged values** when a stereotype is applied to a model-element [92].



- A **Profile** is defined as a package [9] [93] [8], which is stereotyped as `<<profile>>`, that can extend either a meta-model like UML or another profile by respecting the original semantics [9] [94].
- **Profile diagram** is a structured diagram that utilises the UML’s lightweight extension mechanisms’ **Stereotypes**, **Tagged Values**, and **Constraints** for specializing the model-elements to define a customized extension to the generic UML as a DSML to address a specific domain [9] [95]. An example of a **Profile diagram** is illustrated in Figure 2.4. All the customizations are grouped into a single package called a **profile** [9].

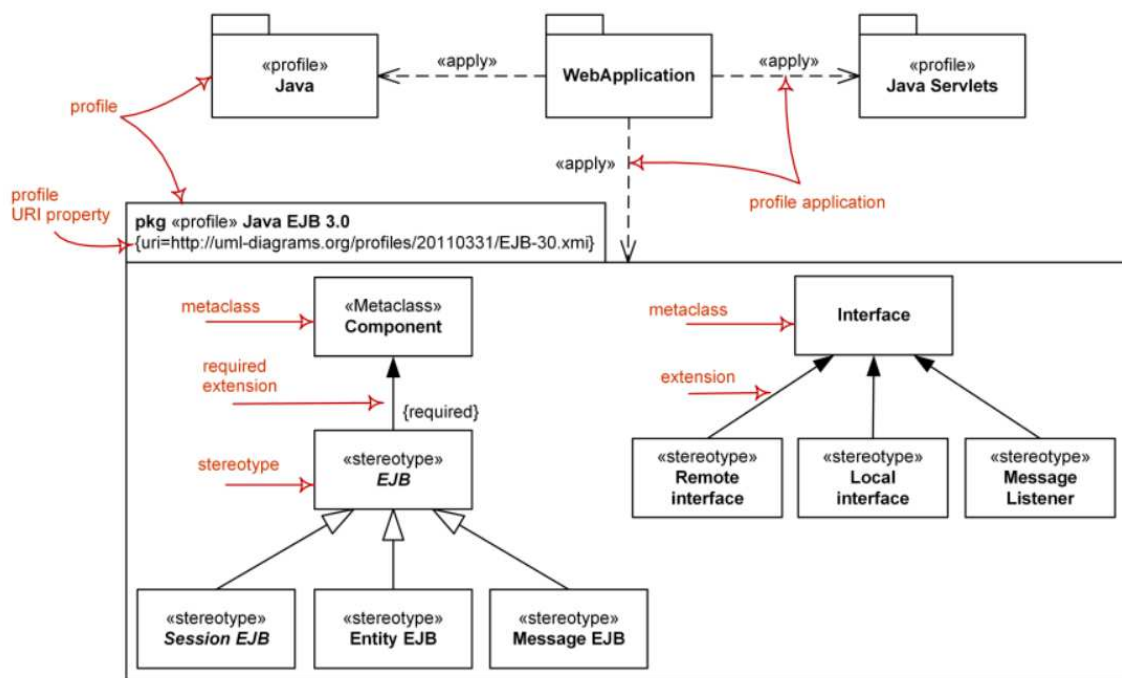


Figure 2.3 Example of core elements of UML profile diagram [95]

As per the [uml-diagrams.org](http://uml-diagrams.org), “the **Profiles** mechanism is not a first-class extension mechanism. It does not allow to modify existing meta-models or to create a new meta-model as MOF does. A **Profile** only allows adaptation or customization of an existing meta-model with constructs that are specific to a particular domain, platform, or method. It is not possible to take away any of the **Constraints** that apply to a meta-model, but it is possible to add new **Constraints** that are specific to the **profile**” [95].

### 2.3. Rich Web-based Applications

*Rich Web-based Applications* are a type of web-based distributed system. This section discusses *distributed systems* in general in the direction of introducing web-based systems, then distinguishes between *web-based applications*, *Rich Internet Applications (RIAs)*, and *Rich Web-based Applications (RiWAs)*. After that, this section talks about *Delta-Communication (DC)* and the types of RiWAs. Then, the general characteristics and essential features of the RiWAs are discussed in the

direction of setting the context of the RiWAs for the RiWAsML and RiWAsDM. Finally, some aspects of RiWAs engineering are discussed.

### 2.3.1. Standalone Systems vs Distributed Systems

Standalone systems are a type of system whose elements are executed within the same device/platform, such as a desktop computer or laptop. Opposed to the standalone systems are distributed systems, in which the elements are distributed across many devices/platforms, as defined below.

**Distributed systems** are comprised of elements distributed across multiple devices/platforms in different tiers (at least between the client and the server), where these elements communicate over a data network [96].

The distribution systems offer many advantages, such as resource sharing, while improving distribution transparency, extensibility, and scalability [96]. Note that this thesis's notion of the concept of distributed systems is not in the context of distributed computing and only focuses on the distribution of the elements across tiers and platforms, which communicates using network protocols such as HTTP.

After the introduction of the *Hyper Text Transfer Protocol (HTTP)*, the *World Wide Web (WWW)* [97] emerged; then, the engineering of distributed systems and their use in the WWW became popular, where the demand for web-based systems [98] increased rapidly. These web-based systems swiftly evolved into much more advanced systems, which are discussed in the next section.

### 2.3.2. Web-based Applications vs Rich Internet Applications vs Rich Web-based Applications

The distributed systems that utilise the service of the web are considered **web-based applications**, which are defined as follows.

**Web-based applications** are a type of distributed system with elements on the client-side that communicate(s) with elements in a web server for processing data and producing information based on the service of the web with HTTP, client-server architecture, request-response model, and other related techniques and technologies [98].

Standard web applications are thin client applications where the data captured from the clients are sent to server elements for processing, which implements the business logic. They use simple GUIs over the traditional page sequential paradigm, resulting in slow responses and, thus, a lack of user experience. Addressing the web-based applications' lower user experience, which is caused by the poor GUIs and slower responses, the *Rich Internet Applications (RIAs)* marked a paradigm shift,

introducing thick client applications in the era of web2. Widening the coverage of the concept of the RIAs, the term *Rich Web-based Applications (RiWAs)* [10] acts as an umbrella term to address the types of web-based systems which provide higher user experience via their rich GUIs, which are capable of partial page rendering enabled by the client-side event handling, client-side model, and advanced communication model named *Delta-Communication (DC)* [11]. The RiWAs are defined as follows.

**Rich web-based application** is a type of web-based application in which the client-side elements contain rich graphical user interfaces, which are capable of partial page rendering using the advanced processing capabilities enabled by client-side events handling and a client-side model. Other than standard HTTP, Delta-Communication techniques and technologies are used for faster communication, which can be implemented in synchronous or asynchronous mode. Rich graphical user interfaces of the rich web-based applications, together with faster Delta-Communication, provide an enhanced and rich user experience [10].

Modern web applications, including *Facebook*, *Google apps*, and *Microsoft apps*, are RiWAs. The online version of Microsoft's *Word*, *Excel*, and *PowerPoint* applications includes the same features as their desktop versions, maintaining the look and feel. The table below compares the standard web applications against the RiWAs, which are detailed in Section 2.3.5.

Table 2.1 Web applications vs. RiWAs

Web applications	RiWAs
Poor GUIs using page sequence paradigm and/or page refresh	Rich GUIs, which can perform partial page rendering
Thin client - Browser-based client (no client-side events handling and model)	Thick client with events handling and model - Browser-based, mobile app, desktop app, IoT app
Standard web server-side application	web server-side applications and web services
HTTP	HTTP and DC

### 2.3.3. Delta-Communication

The advanced communication model of the RiWAs is generally known as *Asynchronous Communication*, and the popular technique to develop this communication model is the *Asynchronous Javascript And XML (AJAX)* [99], which is based on the technology named *XmlHttpRequest (XHR)* object [100]. However, the capabilities of this advanced communication model go beyond the notion of asynchronous communication, and also a variety of technologies and techniques have been introduced after AJAX to develop diverse aspects of this communication. The umbrella term *Delta-Communication (DC)* is introduced in our previous work [11] to address all

these features and different development techniques/technologies. The term DC covers various development techniques and technologies, which are denoted by the taxonomy in Figure 2.5 [10], proving a general concept to address them all. Since the DC techniques/technologies in this taxonomy are already discussed in the literature [10], they are not explained again in this thesis. The DC is defined as follows.

**Delta-Communication** is the rich communication model used by the RiWAs for client-elements to communicate with the server-elements, to exchange only the needful dataset – for a particular feature executed at the time – which is smaller, compared to the size of the request/response of standard HTTP communication. Since the size of the data set communicated is smaller, the communication completes faster, eliminating the work-wait pattern. The processing of the response is done by the client-components in the background; therefore, the page refreshes are eliminated and replaced by partial page rendering to update the content of the current GUI with the results of the response [11].

The DC can be implemented in both pull and push communication modes. In pull-communication, the data is explicitly requested by the client and pulled from the server; in push-communication, the server pushes data to the client(s) when required, even without a request [11]. The RiWAs' communication technologies and techniques are depicted in Figure 2.5.

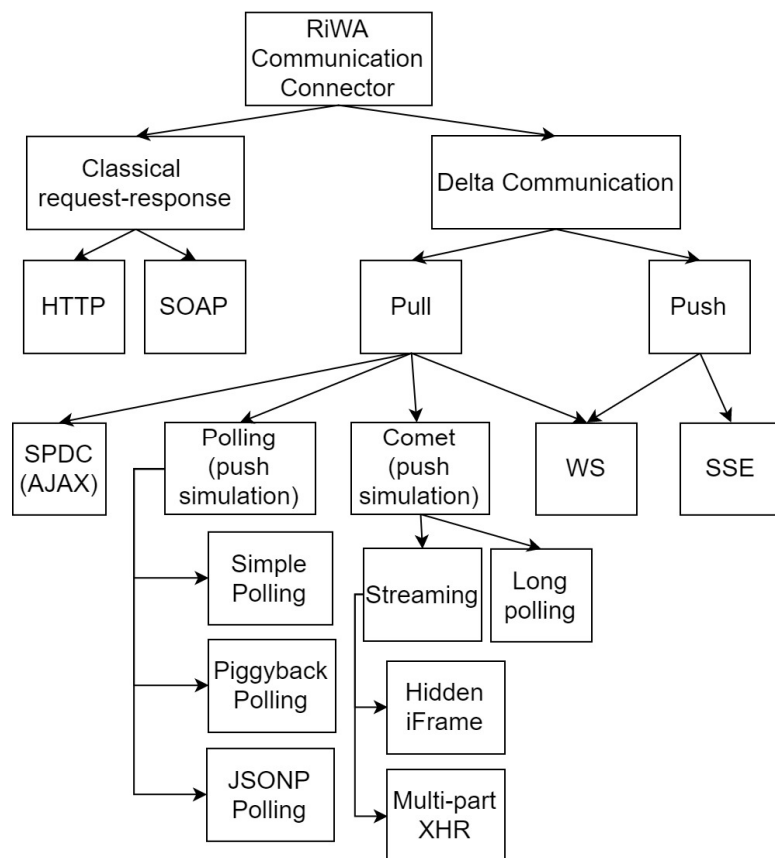


Figure 2.4 Taxonomy for the development techniques and technologies of DC [10]

Since in-depth details of the DC have already been discussed in different forums [11], only the highlights of the DC are listed in this thesis, which are given below.

- DC can be implemented in both pull and push communication modes.
- DC can be implemented in either synchronous or asynchronous mode.
- DC is processed behind the GUI, unnoticed by the user.
- DC enables GUI partial rendering, eliminating full page refresh.
- The amount of the data communicated by the DC is smaller than a full page refresh; therefore, the data are communicated faster.
- By utilising the above features, DC helps improve the user experience.

### 2.3.4. Types of Rich Web-based Applications

The types of RiWAs can be categorized according to the nature of the client-component, as shown in Figure 2.6.

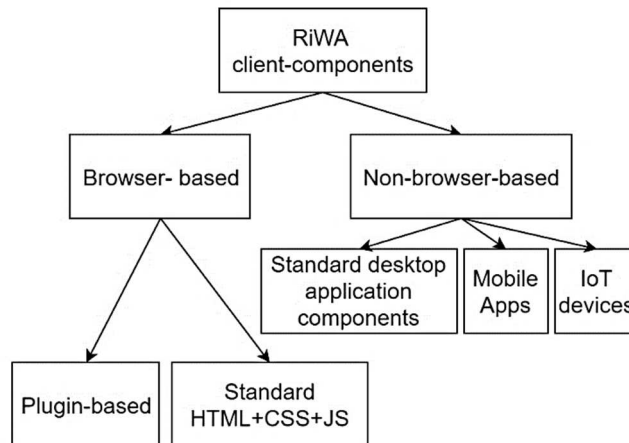


Figure 2.5 Taxonomy for the client-component(s) of RiWAs [11]

In the case of browser-based RiWAs, the proprietary plug-in-based technologies such as *Adobe Flash* [101] or *Java Applets* are not used anymore; instead, the standard JavaScript (JS) based techniques like AJAX [99] have become the de facto standard [102]. For the non-browser-based category, a client-component can be developed to run without a browser, either as a standard desktop app, mobile app, or even as an embedded application in an *Internet of Things* (IoT) enabled device [11].

### 2.3.5. General Characteristics and Essential Features of RiWAs

A large number of technologies/techniques like AJAX, Comet, WebSocket (WS) [103], and frameworks/libraries/plugins have been introduced throughout the last two decades to develop the RiWAs [102]. Regardless of the complexity, the diversity of the types of RiWAs, and the variety of technologies/techniques used to develop them, some general characteristics and essential features of RiWAs are identified [21], which are realised by the RiWAArch style [12]. This section discusses

the general characteristics and essential features of RiWAs towards setting the context for the RiWAsML and RiWAsDM, achieving the research's objective 2 (refer to Section 1.4) by aligning with the step 1.1 of the RiWAsDM implementation process (refer to Section 1.5.3.1). Further examples and elaborations of these characteristics and features are given in Chapters 4, 5, and 6 while introducing the RiWAsML.

### 2.3.5.1. General Characteristics of RiWAs

1. **Rich GUIs:** Rich GUIs are the key advantage of the RiWAs, which improve the user experience by providing rich content similar to desktop applications. Traditional web applications use multiple web pages to implement a function based on the page sequence paradigm, where the RiWAs may implement the same function on a single GUI. For example, consider an email client application; a traditional web application may show the folders on a page, asking the user to select a folder to navigate to the next page, which contains the emails in the selected folder. Then, the user may select an email on that page to navigate to another page to open the selected email. A RiWA may implement this on a single page with three columns, where the left column shows the folders, the middle column shows the emails on the selected folder, and the right column shows the content of the selected email.

More importantly, even though the traditional web application may implement a GUI similar to a RiWA, as a thin client, it still engages page refreshes on each and every click of the user, slowing down the system and impairing the user experience. A RiWA can replace the page refreshes with partial page rendering – which is a vital feature of the rich GUIs – with the help of client-side events handling, client-model, and DC, which improves the user experience compared to the traditional web application.

2. **A collection of applications:** A traditional web application acts as a single browser-based application, and a RiWA can be seen as a collection of *Application* elements (as defined in Section 4.3.1.1) running together, which communicate with each other at runtime. Any RiWA contains at least two *Application* elements, one on the client-side and the other on the server-side. These *Application* elements communicate using HTTP and/or DC. The client of a traditional web application is always browser-based; however, the client-side *Application* element of a RiWA can be a browser-based app, mobile app, desktop app, or even a program running on an IoT-based device. Further, unlike the traditional web applications, the server-side *Application* element of a RiWA can be a standard web server-side application running in a web server, a web service, a micro-service, a cloud-based service, or a process capable of handling HTTP or DC communication.

A RiWA may consist of multiple *Application* elements of different types that run on distinct platforms in different tiers. For example, the use case of a learning management system in

Chapter 8 (refer to Figure 8.11 in Section 8.3.1) includes two client-side *Application* elements (a browser-based app and a mobile app) and two server-side *Application* elements (a web app and a web service).

3. **Client-side events handling:** Traditional web applications – as thin clients – implement event handlers on the server-side, and when a user triggers an event on the client, a request is sent to the server to handle the event. This process always engages page navigation or refresh, slowing down the system and decreasing the user experience. In RiWAs, users may trigger events – like a button click, text type, or drag and drop – on the rich GUIs while interacting with them; with the help of advancements in client-side development technologies/techniques like JavaScript, the event handlers can be implemented on the client-side – to handle the events on the client with the help of client-model – instead of sending the event data to the server for processing. The client-side event handling eliminates the page navigation or refreshes in the direction of improving the user experience by responding to the events faster. If server-side components are required for processing, then the client-side event handlers use DC to communicate with the server components and get the results faster than traditional HTTP-based communication, minimizing the response time and thus improving the user experience. The client-side event handlers are capable of updating the current GUI with the results by partially rendering the necessary GUI sections, eliminating the page refreshes.
4. **Split business logic between client and server:** With advanced client-side technologies, it is possible to develop processing components on the client-side, making the RiWAs thick client applications. In such a setting, business logic can be implemented on both the server and client sides by splitting the model, and the client-side event handlers can utilise the client-model as required without consulting the server-model, making the RiWAs more responsive and scalable than traditional web applications.
5. **Use of DC:** The RiWAs' *Application* elements primarily use DC to communicate with each other, which is faster, thus improving the user experience. Refer to section 2.3.3 for the features of the DC.

### 2.3.5.2. General Essential Features for RiWAs

1. **MVC-based modularization:** Modularization splits the features of a system into components based on their functionality, satisfying the *separation of concern* principle [97] towards increasing simplicity and modifiability [104]. MVC has served well as an architectural style and a design pattern to separate the development aspects into modules based on presentation, event handling, and business logic. Considering the general characteristics of the RiWAs, it is essential to use MVC to improve the simplicity of RiWAs' formalism to assist in development by using the MVC to separate the rich GUIs, client-side event handlers, and split business logic into client-

model and server-model [105]. The RiWAArch style [12] realises MVC based on a version named *Balanced Abstract Web MVC* (BAW-MVC) [105], which is specialized for RiWAs.

2. **DC handling connectors:** Since DC is RiWAs' primary communication type, proper elements for DC handling and management in both client and server are essential. These elements can support separating the DC handling logic from the business logic and selecting proper technologies/techniques to develop DC and related features. The RiWAArch style realises the DC and provides a comprehensive DC connector with the DC-engine on the client-side and the DC-bus on the server-side [10].
3. **Database:** Modern RiWAs necessarily process data and save it for persistence; therefore, databases are used. Considering this viewpoint, 3-tier architecture can be seen as the core formalism of the RiWAs. Hence, contemplating database and related functions is essential for RiWAs.

### 2.3.6. Rich Web-based Applications Engineering

Traditionally, standard SE methodologies like MDA [75] have been used for developing standalone applications. However, after the introduction of the *Hyper Text Transfer Protocol* (HTTP), the *World Wide Web* (WWW) [97] emerged, and the use of web-based systems became popular, and the demand for them increased rapidly. In this setting, *web engineering* was recognized as a dedicated discipline in the late 1990s, introducing new methodologies, tools, processes, etc., to assist web-based systems engineering [106].

Since the RiWAs is a type of web-based application, the RiWAs engineering aspects – including the designing – are accompanied by the web engineering discipline. This section denotes the position for the RiWAs designing within web engineering, which is a sub-discipline of the software engineering domain. A few styles, design methods, and methodologies have been introduced in an attempt to realise the RiWAs' complexity; the related ones are reviewed in Chapter 3. However, RiWAs still lack strong, dedicated design methods/methodologies to cater to their specificities throughout engineering.

#### High-level Designing of Rich Web-based Applications

Some architectural styles have been presented to support the high-level architectural modelling of RiWAs, which are reviewed in section 3.1. Among them, the *RiWAArch style* [12] can be marked as a comprehensive style, which can realise the general characteristics and essential features of RiWAs, discussed in section 2.3.5. The RiWAArch style is shown in Figure 3.1 and reviewed in Section 3.1.3.

The completeness of the RiWAArch style can assist in exploiting it as the standard architecture for RiWAs in general. This thesis utilises the benefits of the RiWAArch style by employing it as the foundation formalism when introducing the RiWAsML.



### Low-level Designing of Rich Web-based Applications

The UML is a GPML; therefore, it fails to address the specificity of web-based applications. Different UML extensions have been introduced to support web application designing, such as *UML-based Web Engineering* (UWE) [23] and *WebML* [107]. However, since the RiWAs contain unique features on top of the traditional web applications – like rich GUIs, client-side events handling, DC, and distributed logic – the web application design tools are not strong enough to support the specificity of RiWAs’ designing. Some UML extensions and new design methods have been introduced for RIAs/RiWAs, which are reviewed in Sections 3.3 and 3.4. Based on the reviews, it can be understood that none of these available design methods provides a complete set of tools to design every aspect of the RiWAs. This problem is elaborated on in section 1.2. As a solution, this thesis introduces a UML-based design methodology named RiWAsDM to cater to the specificity of the RiWAs (refer to Chapters 4 to 7).

## 2.4. Chapter Summary

The *Agile Model Driven Development* (AMDD) approach merges the benefits of *Model Driven Software Engineering* (MDSE) and *Agile Software Engineering* (ASE) by adopting the DMSE’s design activities into ASE. Software design methodology governs the design activities by providing enough design methods, including **modelling language** for **high-level** and **low-level** design, rules and guidelines, CASE tools, etc. The **high-level design** produces **architecture**, which can realise the system’s high-level architectural elements and their configuration, and **low-level design** details the high-level design elements, assisting in mapping the designs into development. Software design may use either the **top-down approach** or the **bottom-up approach**.

Software **modelling languages** offer **models**, **model-elements**, and rules and guidelines to design software systems. **UML** is the de facto standard *General-Purpose Modelling Language* (GPML), and *Domain-Specific Modelling Languages* (DSMLs) can be introduced using UML’s extension mechanism. **Attributes** like simplicity, adoptability, learnability, readability, and understandability express the quality of a modelling language.

RiWAs’ **general characteristics and essential features** are rich GUIs, a collection of *Application* elements, client-side events handling, split business logic between client and server, use of DC and DC connectors, MVC-based modularizing, and use of databases, which are realised by the **RiWAArch style**. A DSML and a design methodology for RiWAs should cover them all.

## Chapter 3. Review of Available Related Solutions

---

The available related solutions are reviewed under four categories: architectural styles, high-level design methods/methodologies, low-level design methods/methodologies, and research works. The reviewing criteria are given at the beginning of each section. Only an overall review of the available related work is presented in this chapter, and in-depth model and model-elements level reviews are given where necessary in Chapters 4 to 7 while implementing the RiWAsML/RiWAsDM.

### 3.1. Architectural Styles for RiWAs

This section reviews the architectural styles available for RiWAs towards selecting a strong style to be exploited as a framework for the proposed design methodology, aligning with Step 1.1 of the RiWAsML implementation process (refer to Section 1.5.3.1). The styles available for RiWAs are reviewed using the following criteria, which are defined towards understanding their simplicity and adoptability.

1. Specificity for RiWAs.
2. Realise general characteristics and essential features of RiWAs (refer to Section 2.3.5) towards comprehensiveness.
3. Abstract: independent from development technologies/techniques, hence, portable.
4. Based on known available style(s), and thus, it is easy to learn/understand.
5. Adoption is discussed; therefore, it supports development.

#### 3.1.1. SPIAR: A Component and Push-based Architectural Style for AJAX Applications (2008) [108]

In this research, Mesbah and Deursen introduce an architectural style named *Single Page Internet Application architectural style* (SPIAR style) for single-paged browser-based RiWAs, which focuses more on the front-end implementation. SPIAR style tries to “*minimize user-perceived latency and network usage, and improve data coherence and ultimately user experience*” [108]. Since SPIAR only focuses on single-page browser-based systems, it does not realise other types of RiWAs, like systems with mobile client apps.

As the name of the research paper denotes, its context is only AJAX applications; hence, SPIAR cannot be considered abstract. Furthermore, the SPIAR style is based on the characteristics of some available frameworks: Echo2, GWT (a web framework offered by Google), Backbase (a commercial package delivered by Backbase), cometd (a server-side push framework), and Dojo (a client-side framework to work with cometd). Because of these technological constraints, the SPIAR does not provide an abstract solution.

The DC-related aspects are addressed by the SPIAR as follows. The requests from the *AJAX Engine*, which is on the client-side, are handled by an element named *Decoder* on the server-side, and the responses are sent back to the AJAX engine by an element named *Encoder*. The *Ajax Engine*, together with the *Encoder* and *Decoder*, create a connector for pull-DC. A connector with a *Push-Server* component in the server and a *Push-Client* component in the client is used to handle push-DC. These connectors improve the visibility and simplicity of the SPIAR style. However, the SPIAR style does not depict how non-DC communication is integrated. Since the SPIAR style targets single-page applications, standard HTTP communication is eliminated after the application's first time loading; thus, it does not require non-DC for page navigation. SPIAR uses a novel modularization unrelated to MVC, and the implementation of business logic or controllers is not explicitly discussed, which lowers the simplicity more.

SPIAR does not incorporate available styles; only the AJAX engine is taken from the AJAX general architecture. The other new elements and the configuration need to be learned in order to adopt the SPIAR. As a novel approach with some new components, SPIAR incorporates a high initial learning curve for anyone without experience with the frameworks mentioned above. How the SPIAR style is adopted in development is not discussed, even for the mentioned frameworks, limiting it to a conceptual solution with low usability and, thus, low adoptability.

### **3.1.2. jAGA: jQuery-based Ajax General Interactive Architecture (2012) [109]**

This research introduces an architecture for AJAX-based RiWA, named jAGA. jAGA focuses on browser-based RiWAs; thus, support for other types of RiWAs, like mobile apps, cannot be gained. Since jAGA is based on jQuery and AJAX, it is not abstract, and as aforesaid, it only targets browser-based RiWAs, limiting the scope of the platform and technologies/techniques. Cumulatively, as a result, it has lower portability and adoptability.

The simplicity of jAGA is mainly based on its AJAX sub-operations, and a strong MVC-like module separation is not applied. jAGA does not explicitly discuss how the DC is separated from non-DC; nevertheless, the AJAX sub-operations may help separate the AJAX-based DC from the other elements, which is a jAGA's strength. jAGA uses a novel approach to realise the AJAX-based RiWAs, consisting of several modules and sub-modules based on some new concepts, without inheriting characteristics from available architectural styles; hence, the usability of jAGA can be marked low. Further, jAGA does not discuss development-related details, further lowering the adoptability.

### **3.1.3. RiWAArch Style: Rich Web-based Applications Architectural Style (2020) [12]**

RiWAArch style is a comprehensive style for RiWAs since it realises the general characteristics and essential features of the RiWAs (refer to section 2.3.5) with a higher simplicity. The RiWAArch style

does not depend on any technology/technique or platform, making it an abstract, portable, and, hence, adoptable solution for RiWAs engineering.

Modularization of the RiWAArch style is firmly based on MVC, improving simplicity by separating business logic from presentation and event handling. Simplicity in RiWAArch style is enhanced by unscrambling the DC handling entirely from the MVC modules. The RiWAArch style centralizes DC handling by using a DC connector, separating DC from non-DC, further enhancing simplicity, and also improving visibility, modifiability, and performance properties.

As a hybrid of well-known styles like client-server, MVC, and message-bus styles, the RiWAArch is easy to learn/understand and, hence, highly usable. Also, adoption is demonstrated via a comprehensive use case, explaining every aspect of the style.

All these aspects make the RiWAArch style a highly adoptable solution for RiWAs engineering. The RiWAArch style is shown in Figure 3.1.

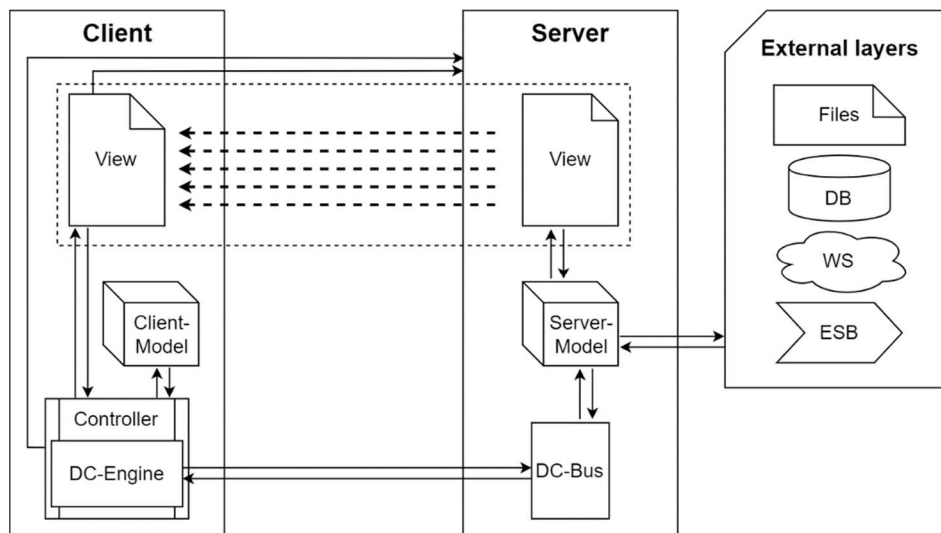


Figure 3.1 The Rich Web-based Applications Architectural style (RiWAArch style) [12]

### 3.1.4. Summary of the Available Architectural Styles Review

Table 3.1 summarises the review of the available architectural styles, aligning with the review criteria.

- **C1:** Specificity for RiWAs.
- **C2:** Realise general characteristics and essential features of RiWAs (refer to section 2.3.5) towards comprehensiveness.
- **C3:** Abstract – independent from development technologies/techniques, hence, portable.
- **C4:** Based on known available style(s), and thus, usable by being easy to learn/understand.
- **C5:** Adoption is discussed; therefore, it supports development.

Table 3.1 Summary of the architectural styles review

Style\ Criteria	C1	C2	C3	C4	C5
<b>SPIAR</b>	Single paged	Partial	Based on some frameworks	Lower usability	Lower adoptability
<b>jAGA</b>	Browser-based	Partial	jQuery and AJAX	Lower usability	Lower adoptability
<b>RiWAArch</b>	RiWAs in general	Comprehensive	Abstract	Higher usability	Higher adoptability

Based on the review, considering the high satisfaction of the review criteria, this research values the RiWAArch style as an abstract, comprehensive style that can realise the general characteristics and essential features of the RiWAs, which is highly adoptable to RiWAs engineering. Therefore, the RiWAArch style is selected to provide the foundation for the RiWAsML, fulfilling the research objective 1 (refer to Section 1.4) by following Step 1.1 of the RiWAsML implementing process (refer to Section 1.5.3.1). Chapters 4 to 7 discuss how the RiWAArch style is utilised to implement the RiWAsML.

## 3.2. High-level Design Methods/Methodologies

There are three main approaches for designing software architectures: informal box-and-line drawing, formal *Architectural Description Languages* (ADLs), and UML-based languages (refer to sub-section *Architectural Description Language* in Section 2.2.2.1). These approaches and the methods/methodologies used in them are reviewed based on the criteria below.

1. **Usability:** easy to learn and read/understand employing syntax and semantics
2. **Adoptability:** adoptable to RiWAs engineering by means of usability and development support.

### 3.2.1. Informal Box-and-Line Drawing

This section reviews the box-and-line approach in the direction of highlighting the importance of formal design languages.

The box-and-line approach is the earliest and probably the most used technique to draw software architectures. In this approach, formal notations are not provided, and the architect has the freedom to define syntax and semantics for them. The elements/entities are denoted using boxes, and the communication or any other relationship between the elements is depicted using lines. This technique is an easy way of drawing diagrams without restrictions where anything can be flexibly included and illustrate architectures completely. Therefore, all the aspects related to RiWA architectures – specified by the review criteria – can be depicted using boxes and lines.

The main disadvantage of the box-and-line approach is when the boxes are used to illustrate the entities and lines are used to depict relationships or communication channels, the semantics of the elements are omitted from the design [110]. As per Garlan [78], there are many disadvantages related

to the box-and-line technique, such as “*the meaning of the design may not be clear since the graphical conventions will likely not have a well-defined semantics. Informal descriptions cannot be formally analysed for consistency, completeness, or correctness. Architectural constraints assumed in the initial design are not enforced as a system evolves.*” Since formal syntax and semantics are missing in the box-and-line approach, different diagrams may use diverse meanings for boxes and lines, and the comparison of diagrams is problematic due to inconsistencies. Also, an initial effort and a learning curve are engaged with each diagram to learn the syntax and semantics used for the diagram.

Formal design languages are introduced to address the issues of the box-and-line approach, which are reviewed in the following sections. OMG’s MDA [75] explains the advantages of having a formal modelling language in the direction of improved communication, as listed below.

- Provides well-defined terms and notations that assist in a common understanding of the context.
- Provides a foundation for models as semantic data to be managed, versioned, and communicated.
- Provides libraries of reusable (asset) models and model-elements as a standard vocabulary with rules, reusable processes, business object models, or design patterns.
- Models and model-elements become part of the “corporate memory” for designing within an organization.

Considering all these concerns, the box-and-line approach can be seen as lacking usability and, thus, adoptability.

### 3.2.2. Formal Architectural Description Languages

RiWAsML/RiWAsM focuses on the UML-based design, and *Architectural Description Languages* (ADLs) are considered out of the scope; however, they are reviewed in this section to show their lack of usability/adoptability.

Formal ADLs use programming-like languages to define software architectures whose code can generate visual diagrams. Each formal ADL uses a dedicated set of syntax and semantics; hence, they always incorporate a higher initial learning curve compared to the methods/methodologies using the semi-formal UML-based approach.

Formal ADLs have been introduced since the early 90s and are mentioned in the literature, for example, *AADL*, *ADML*, *Darwin*, *Koala*, *Rapide*, *SBC-ADL*, *UniCon*, and *Wright*. However, it appears that the ADLs had not been much accepted due to their higher learning curve and inflexibility; hence, they were not maintained over time, and their development has ended. Therefore, these early ADLs are not reviewed in this section.

After surveying the ADLs, Ozkaya and Kloukinas [111] conclude that the lack of interest shown for the ADLs can be the “*consequence of three main problems that no ADL has managed to solve at the*

same time: (i) lack of support for formal analysis of architectures, (ii) notations that sometimes make specifying large and complex system architectures harder than it should be, and (iii) potential unrealizability of system architectures.” The requirements of web-based systems were rapidly evolving, and the ADLs had not been updated to cater to these requirements, which can be seen as another reason for ADLs to become out of favour.

The formal ADLs identified during the literature survey as still maintained are reviewed below within the context of this thesis.

### **3.2.2.1. xADL (version 3.0, 2002) [112]**

The latest related publication was in 2002 [113], and the official website [112] has no other timeline details related to the releases, and it looks obsolete. xADL is defined as a set of XML schemas that provide unprecedented extensibility and flexibility, which are the strengths of the xADL. xADL allows engineers to “write new XML schemas extending xADL to add your new modelling constructs” [112]. xADL’s official website does not provide many details about the current schema and how it can be used. However, based on the extensibility of the xADL, we can think that it can provide the expected support to design the RiWAs’ architectural elements and ways to show additional details on the architecture. The disadvantage of this extensibility is once the xADL is extended for a project, the extensions are less formalized and may exhibit all the weaknesses of informal notations like the box-and-line approach. This problem can be overcome by introducing formal extensions, focusing on particular types of systems.

### **3.2.2.2. Acme (2011) [114]**

Acme is a generic ADL for software systems, which is built on a core ontology of seven types of entities for architectural representation: *components*, *connectors*, *systems*, *ports*, *roles*, *representations*, and *rep-maps*. As a general ADL, it does not address the specificity of the web-based systems and, thus, RiWAs. Even though Acme has syntax to represent the elements *system*, *components*, and *connectors*, entities/elements like *tiers* and *platforms* cannot be specified.

Since Acme is a generic ADL, DC communication and related aspects are not given. Nevertheless, Acme can include details with the communication channels, which can be helpful in denoting DC-related information to specify the DC channels.

Acme has no exact ways to define additional entities like databases. Nevertheless, Acme supports including additional properties in the architecture description, which can be exploited to specify different entities and include more details like metadata. However, this method is complex and makes Acme less usable.

### 3.2.2.3. xAcme with xArch (2001) [115]

xAcme is an extension of Acme that describes Acme using xArch. xArch is an XML-based standard that specifies the architectural structure of software systems, which defines the *ComponentInstance*, *ConnectorInstance*, *LinkInstance*, *GroupInstance*, and *ArchInstance*. Yet, xArch needs more details like *properties*, *families*, and *constraints*, which are in Acme. xAcme tries to add these details using XML to the xArch to describe all the features provided by Acme. xAcme extends xArch to support the Acme extensions: *properties*, *acmeProperties*, *families*, *constraints*, and *acmeConstraints*.

However, since xAcme is a general early approach, it lacks features that serve the RiWAs. xAcme is an approach to satisfy Acme through xArch; hence, xAcme includes Acme's shortcomings in the context of RiWAs. The entities/elements above the components level, such as tiers and platforms, cannot be defined in xAcme and the communication-related aspects are weakly addressed. There are no explicit ways to include additional entities or details to the architecture description; however, the properties and constraints might be exploited to add these details.

### 3.2.2.4. AADL: Architecture Analysis and Design Language (2022) [116] [117]

AADL is a maintained ADL; its official website was updated in 2022 [117]. AADL offers standards for defining guidelines for design and analysis through a DSML. As per the official website, “*AADL captures large designs through high-level architectural concepts built after domain expertise: component categories that describe key building blocks, such as processor, devices, threads, and rules to assemble them. Through careful abstractions, complex designs can be captured as smaller models amenable to inspection and analysis*” [117]. Regardless of all the benefits offered by the AADL, as an ADL, it still incorporates complexity, lowering the usability.

### 3.2.2.5. Summary of the Review of ADLs

Table 3.2 summarises the ADLs review based on the criteria below.

- **C1: Context** – Is the ADL generic, or can it be extended for domain-specific features?
- **C2: Status** – is the ADL maintained or obsoleted?

ADLs can be seen as an outdated approach to designing software systems. However, this review identified that AADL is a living and maintained ADL, and xADL can also be marked as a potential solution that can be extended towards providing features to design RiWAs architectures. Still, they may have adoptability issues because of their complex nature.



Table 3.2 Summary of the ADLs review

ADL \ Criteria	C1	C2
<b>xADL</b>	Generic but can be extended	Obsoleted
<b>Acme</b>	Generic	Obsoleted
<b>xAcme xArch</b>	Generic	Obsoleted
<b>AADL</b>	Generic. Can integrate with languages like SysML	Maintained

### 3.2.3. UML-based/UML-like Design Methods/Methodologies

This section primarily reviews solutions available for high-level designing, which are based on UML or similar to UML. Since some of these methods can be deemed as methodologies, the review of identified methodologies is merged with the review of high-level methods. This section only provides an overall review, and the model-element-level details are further reviewed in Chapters 5, 6, and 7 while introducing the proposed RiWAsML and RiWAsDM.

The review criteria are based on the requirements set for the high-level design aspects of RiWAs in Section 4.3.

1. **Context:** offer generic or specific features towards web-based applications or RiWAs.
2. **The simplicity of elements:** realise the architectural elements and offer syntax for them – tiers, platforms (hardware, OS, and application-level virtualization), applications, components (model and controller), and connectors.
3. **The simplicity of communication channels:** realise the different types of communication channels and provide notations for them – HTTP, DC, data reading, and method calls.
4. **Additional architectural elements:** supply high-level elements like users, data sources, web services, and networks.
5. **Additional details:** assist in showing further information on the designs using text-based notes.

#### 3.2.3.1. MDA (version 2, 2014) [75] with UML (version 2.5.1, 2017) [7]

OMG's *Model Driven Architecture* (MDA) [75] can be seen as a software design methodology, which is based on the MDSE (refer to section 2.1.3). MDA suggests starting with platform-independent models and then converting to platform-specific models to assist domain-specific implementation. The basic concepts of MDA are *System*, *Model*, *Modeling Language*, *Architecture*, *View* and *Viewpoint*, *Abstraction*, *Architectural Layers*, *Transformation*, *Separation of Concerns*, and *Platform*. MDA also discusses MDA Model Transformation and Execution, System Lifecycle Support in MDA, and Set of MDA Standards. The discussions on SDLC support generally state

MDA's impact on the SDLC; however, integrating MDA into other SE methodologies, especially the Agile SE, is not discussed.

UML is MDA's standard modelling language, and the fundamental details of the UML as an MOF meta-model are discussed in Section 2.2.4. Among software modelling languages, the UML [7] – introduced and maintained by the OMG [118] – is probably the widely accepted and used modelling language for low-level software designing under the OODD paradigm [9]; therefore, can be seen as the de facto standard software modelling language. The UML is a GPML enriched with a set of models to cater to the general requirements of OO software systems designing. When going through the literature, it is evident that UML is lasting compared to ADLs; therefore, the UML-based approach can be marked as a better approach to software designing, including architecture.

Since UML is a GPML, the UML models and model-elements do not provide enough features to design the software architectures [78] [111]. Models like the *Component diagram* and *Deployment diagram* allow designing some aspects related to the high-level architecture; however, these models are unable to capture all the architectural aspects since the UML notations are “*limited by their lack of support for formal analysis and their lack of expressiveness for some architecturally relevant concepts*” [78]. Hence, some solutions extend UML to cater to the specificity of architectural designing.

UML's diagram list is given in Figure 3.2 under two main categories: structure and behaviour.

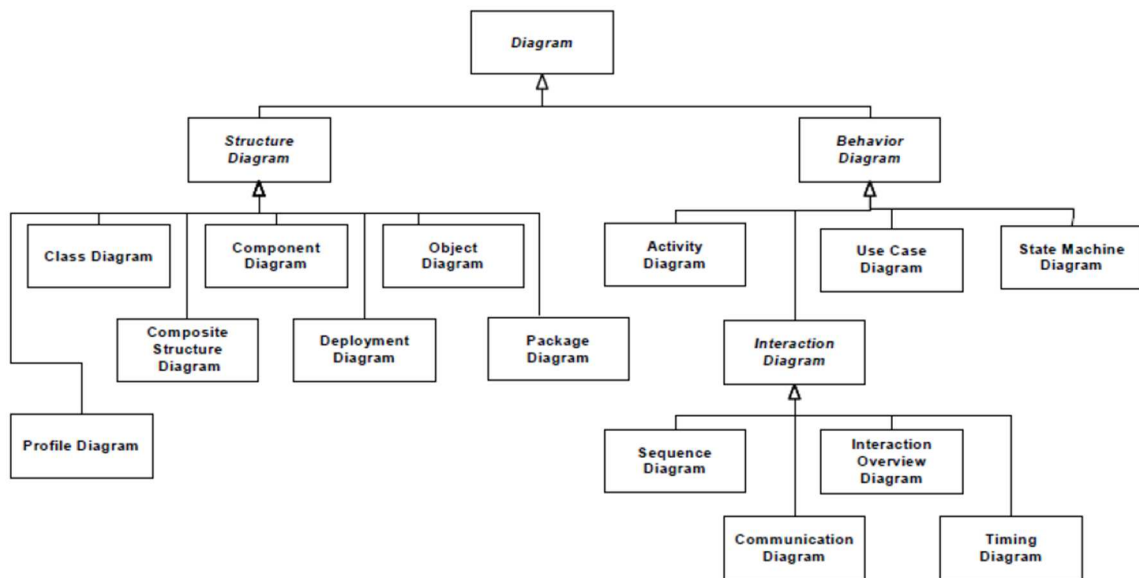


Figure 3.2 The list of UML diagrams [119]

The generic characteristics of the UML within its broad scope do not cater to the specificity of some software systems, such as web-based applications. For example, UML does not include tools to model the presentation, DC, and related aspects of the RiWAs. UML provides an extension mechanism to define DSMLs by extending the UML's abstract meta-model (refer to Section 2.2.4.5).

### 3.2.3.2. Arc<sup>42</sup> (Version 8, 2022) [80] [120]

Arc<sup>42</sup> can be considered a methodology that “*provides a template for documentation and communication of software and system architectures*” using the following 12 template sections with tips, which include designs [80].

1. **Introduction and goals:** Requirements, stakeholder, (top) quality goals (24 tips)
2. **Constraints:** Technical and organizational constraints, conventions (5 tips)
3. **Context and scope:** Business and technical context, external interfaces (19 tips)
4. **Solution strategy:** Fundamental solution decisions and ideas (6 tips)
5. **Building block view:** Abstractions of source code, black-/whiteboxes (28 tips)
6. **Runtime view:** Runtime scenarios: How do building blocks interact (11 tips)
7. **Deployment view:** Hardware and technical infrastructure, deployment (10 tips)
8. **Crosscutting concepts:** Recurring solution approaches and patterns (11 tips)
9. **Architecture decisions:** Important decisions (10 tips)
10. **Quality:** Quality tree and quality scenarios (8 tips)
11. **Risks and technical debt:** Known problems, risks and technical debt (6 tips)
12. **Glossary:** Definitions of important business and technical terms (6 tips)

Arc<sup>42</sup>'s primary focus is to document the software systems, and “*since a complete package of templates is given, the efforts to document architectural details are minimized*” [80]. The following sections in the Arc<sup>42</sup> document template include designs.

- **Building block view:** a high-level architectural view with 4 levels, starting with the *scope and context* and then decomposing the elements down into *Level 1*, *Level 2*, and *Level 3*.
- **Runtime view:** describes concrete behaviour and interactions of the system's building blocks in the form of scenarios. The runtime view uses sequence diagrams and activity diagrams.
- **Deployment view:** captures the technical infrastructure and the mapping of (software) building blocks to that infrastructure elements. The deployment and component diagrams are utilised in this view.

Arch<sup>42</sup> does not provide syntax to design architectures; instead, it provides some guidelines for the building block view, which is similar to the C4 model [79] (refer to Section 3.2.3.5). The low-level designing under the runtime and deployment views are based on the UML.

### 3.2.3.3. SAP's OOA (Version: 16.7.07, 2023) [45] with TAM [82]

SAP provides a tool called *PowerDesigner* to design systems using *Object-Oriented Modelling* (OOM) concepts, and the complete process is called *Object-Oriented Architecture* (OOA). OOA discusses aspects like web services designing, generating and reverse engineering OO source files, generating models from an OOM, checking an OOM, and using language-related details on models such as Java, VB, C++, and C#.

SAP's OOA is mainly based on the standard UML and additionally offers a standard for architecture modelling named *Technical Architecture Modeling (TAM)*, which provides as few models as possible but as many as necessary to maintain simplicity as the primary focus, defining and describing the following.

- Which diagram types are allowed to model technical architecture at SAP?
- What elements in a particular diagram type are allowed, optional, or prohibited?
- Which extensions of the UML meta-model have been made for specific diagram types?
- What are the semantics of newly added elements in diagram types, and how can those elements be used?

Even though the name says *architectural modelling*, TAM discusses other aspects, such as requirements modelling using use case diagrams and low-level designing. The documentation mentions that “*TAM does not contain nor refer to any profiles*” [82], and it is unclear how TAM differs from standard UML.

As per TAM's documentation, TAM introduces a new diagram named the *Component/Block diagram* [82], which is based on the UML component diagram. The *component/block diagram* contains some syntaxes useful for RiWAs' architectural modelling, and it shows “*the compositional structure of any system that processes information and illustrates how agents access data in storages and communicate over channels*” using the syntax below [45].

- **Elements:** Boundary Line, Protocol Boundary, Agents, Human Agents, Storages, Common Feature Areas, and Multiple Dots
- **Communication:** Request/Response, Unidirectional, and Bidirectional Communication Channels
- **Access:** Read, Write, and Modify Accesses

These syntaxes do not include tiers, platforms, and DC-related elements. Also, other than the high-level *component/block diagram*, the low-level design models for views, controllers, and DC connectors are not provided.

#### 3.2.3.4. ArchiMate (version 3.2, 2022) [121]

ArchiMate provides “*an integrated architectural approach that describes and visualizes the different business domains and their relations*” [122]. The ArchiMate is an independent language – which is inspired by UML but not based on the UML – with a core framework with 5 layers (business, application, technology, physical, and Strategy) and 3 aspects (passive structure, behaviour, and active structure) to classify the elements of the language [121]. The model-elements in each layer are listed below.

- **Business Layer:** Business Actor, Business Role, Business Collaboration, Business Process, Business Function, Business Interaction, Business Event, Business Service, Business Interface, Business Object, Product, Contract, and Representation
- **Physical Layer:** Equipment, Facility, Distribution Network, and Material
- **Technology Layer:** Node, Device, System Software, Technology Interface, Technology Function, Technology Service, Technology Collaboration, Technology Interaction, Technology Event, Technology Process, Artifact, Communication Path, and Network
- **Application Layer:** Application Collaboration, Application Component, Application Service, Application Function, Application Interaction, Application Interface, Application Process, Application Event, and Data Objects
- **Strategy Layer:** Resource, Capability, and Course of Action

Further, some other notations are given under the categories below.

- **Motivation Elements:** Stakeholder, Driver, Assessment, Goal, Outcome, Principle, Requirement, Constraint, Meaning, and Value
- **Implementation & Migration Extension:** Plateau, Gap, Deliverable, Work Package, and Implementation Event
- **Relationships:** Composition, Aggregation, Assignment, Realisation, Serving, Access, Influence, Triggering, Flow, Specialization, Association, and Junction

ArchiMate has a large set of notations, also including colour codes. The examples of the ArchiMate show that the tiers can be separated using a dashed line, and the type of tier can be written on the dashed line. ArchiMate includes notations for *node*, *device*, *system software*, *product*, *component*, and *interface*. The *device* and *system software* can define the platform; however, the *system software* can also denote environments like an application server, a database system, or a workflow engine. The *product* can depict an application, and the components can be defined using the *component* notation, and the connectors can be defined using the *interface* syntax. Notations to include many additional entities/elements to the architecture, like *users*, *roles*, *databases*, *networks*, and *events*, are provided by ArchiMate, with more notations to add additional details like *meaning* and *value*. With a massive amount of syntax, the simplicity of ArchiMate is improved; however, in the context of RiWAs, these elements do not offer sufficient abstract realisation.

Even though ArchiMate is inspired by UML and has some similar syntax, the new framework and notations enable a higher learning curve for UML users, lowering its usability. Further, similar to UML, as a generic language, ArchiMate does not include syntax to depict RiWAs-specific DC and related aspects. Therefore, the ArchiMate cannot be considered an adoptable solution for RiWAs engineering.

### 3.2.3.5. C4 model [79]

The C4 model can be considered a design methodology which looks at a system at four abstract levels. It uses an approach similar to *Data Flow Diagrams* (DFDs), where the overall context is designed in top-level diagrams, and the inner content of the elements is detailed in the lower level, as stated below.

- **Context level:** The system **context** is the topmost layer made of containers of applications and data stores.
- **Container level:** A **container** comprises components, which are logical units that interact within the container.
- **Component level:** A **component** encompasses code-level elements like classes and algorithms, which require lower-level designing.
- **Code level:** A component's **code**-level design model can be considered a class diagram.

The C4 model is inspired by UML; however, it is primarily based on box-and-line notation. Following the concept “*abstraction first, notation second*”, C4 uses minimal notations to depict *system context, container, database, mobile app, browser, person, and relationship* [79]. The designer has the freedom to introduce and use new notations; in that case, the C4 model suggests providing a legend of the notations used. This nature makes the C4 model more like an approach than a method or a methodology with formal syntax and semantics.

The C4 model does not explicitly realise the separation of tiers and depicts the platform as limited to mobile apps and browsers. When illustrating the communication channels, the C4 model appreciates using unidirectional lines with annotations. Since any other details related to the communication lines in the C4 model are not strongly defined, the communication channels can be vague without stating the proper type or mode. Further, as a general approach, the C4 model does not provide formal guidelines to explicitly realise DC and related aspects. However, annotations can be exploited to include more details informally. The users and DBs can be included in designs; still, the flexibility of having more notations can be exploited to add more entities/elements to the design. The C4 model highly appreciates including more textual details to describe all design aspects sufficiently.

Large systems may benefit from the C4 model's approach of designing the system in different layers when it is complex to illustrate everything in a single overall architectural diagram. With the flexibility of including new design elements in modelling, the C4 model could cater to the specificity of the RiWAs. However, the lack of syntax and rules in the C4 model engages all the issues related to the informal box-and-line approach, as discussed in Section 3.2.1.

### 3.2.3.6. Summary of UML-based/UML-like Design Methods/Methodologies

Table 3.3 summarises the UML-based methods/methodologies review based on the following criteria in the context of architectural design.

- C1 – Context
- C2 – The simplicity of elements
- C3 – The simplicity of communication channels
- C4 – Additional architectural elements

Table 3.3 Summary of the UML-based methods/methodologies review

Solution\Criteria	C1	C2	C3	C4
<b>MDA with UML</b>	- Methodology - General - Primarily low-level designing	- <i>Package</i> - can be exploited for tier - <i>Node</i> =platform - Component - Connector=interface (no DC support)	- No DC	- Available
<b>Arc<sup>42</sup></b>	- A general methodology focuses on documenting the software - No syntax is provided, and mainly UML-based - Some guidelines are given for the <i>building block view</i> , which is for architectural modelling and similar to the C4 model			
<b>SAP's OOA/OOM with TAM</b>	- Process + tool - General - UML-based	- <i>Protocol boundary</i> line to denote tiers like separation - <i>Agent</i> is likely to be exploited for Application element - Components - Connectors are similar to UML interfaces	- Direction and some types are available - No DC	- Storages and agents
<b>ArchiMate</b>	- General - A framework and a large collection of notations - Not UML-based	- <i>Node</i> =platform - <i>product</i> is similar to <i>application</i>	- No DC	- Provides many notations
<b>C4</b>	- RiWAs - Box-and-line	- Mobile and browser platforms - Containers are similar to applications - Components	- Unidirectional lines with annotations. - No DC.	- Informally can add more elements - Text-based details can be included

The available solutions for architectural designing are mostly generic and unable to cater to the specificity of the RiWAs. They commonly lack notations depicting tiers, *Application* elements, views, and DC-related aspects. TAM's *component/clock diagram* is likely to be specialized for RiWAs by providing the lacking elements. C4 is a potential approach that can be improved with formal syntax to cater to the specificity of RiWAs architecture modelling. These design methods/methodologies are compared against the RiWAsML/RiWAsDM in Section 9.2 in the Evaluation chapter.

### 3.2.4. Summary of the Available High-level Design Methods/Methodologies Review

Table 3.4 summarises reviews of high-level design approaches, indicating their pros and cons by analyzing them in the context of completeness and usability as follows.

- **C1:** Comprehensiveness – support RiWAs designing by providing high-level processing and communication-related elements, communication channels and related aspects, additional entities, and additional textual details
- **C2:** Usability – UML-based and able to utilise for RiWAs architecture designing, with a lower learning curve

Table 3.4 Summary of reviews of high-level design methods/methodologies for RiWAs

Method \ Criteria	C1	C2
<b>Box-and-line</b>	All the aspects can be designed.	Can be easily used for RiWAs. No formal syntax; hence, it lacks usability.
<b>ADLs</b>	Some aspects can be designed.	Not specialized for RiWAs. Complex, incorporates a higher learning curve, and is less usable.
<b>UML-like</b>	Some aspects can be designed.	Not specialized for RiWAs. Usable (UML-based solutions are highly usable).

When the box-and-line approach is used, all the aspects of the RiWAs architectures can be included in the design. However, this approach introduces many issues due to the informal syntax and semantics, which stresses the requirement for more formal approaches.

Available and maintained ADLs are capable of depicting most of the entities/elements and details needed for RiWAs architecture. As a type of programming like languages, ADLs always incorporate a higher learning curve. None of the available ADLs is specialized for RiWAs; hence, it cannot be adopted to RiWAs architectural designing without extensions. For example, xADL can be extended to support all the requirements, which needs extra effort. However, the complexity of the ADLs makes them a less usable approach.

The available UML-like/UML-based methods can depict most entities/elements and details of the RiWAs, yet they primarily do not support DC and related aspects. Therefore, none of the available UML-based/UML-like methods can completely design RiWAs architectures, and they cannot be directly adopted for RiWAs architecture designing. However, since they are based on UML or similar to UML, the learning curve is lower than that of ADLs; hence, they are more usable.



### 3.3. Low-level Design Methods/Methodologies

Web engineering design methods/methodologies have been introduced since the beginning of the web era in the late 1990s. The emergence of RIAs started in the early 2000s; since then, some web engineering methods have been extended, and new methods/methodologies have been introduced to support RIAs/RiWAs engineering. The RIAs/RiWAs design methods/methodologies generally focus on the low-level modelling aspects; the early methods/methodologies have been reviewed [16].

This thesis selects available relevant design methods/methodologies and briefly reviews them based on the following criteria:

1. **Context:** if method/methodology is specifically for RiWAs.
2. **Comprehensiveness:** if the method/methodology supports low-level designing of the general characteristics and essential features of the RiWAs, discussed in Section 2.3.5.
3. **Simplicity (of elements):** if the method/methodology realises RiWAs' low-level views, controllers, models, and connectors.
4. **Simplicity (of communication types):** if the method/methodology realises RiWAs' low-level communication types.
5. **Development and integration support:** if the method/methodology discusses developing the low-level elements and integrating RiWAs designing into RiWAs engineering.

This review provides an overview, and more detailed information is looked into where required in the relevant sections of chapters 4, 5, and 6. Since UML is a GPML and has already been reviewed in Section 3.2.3, it's not included in this section.

#### 3.3.1. UWE (version 3.0, 2016) [23] and UWE-R (2009) [123]

*UML-based Web Engineering* (UWE) is a lightweight extension of the UML [23], which extends the standard UML and provides the following models and model-elements for web application designing.

- **Requirements model**
  - **Use case model:** browsing, processing, and webUseCase.
  - **Activity model:** userAction, systemAction, displayAction, navigationAction, displayPin, and interactionPin.
- **Content model:** similar to UML class diagram.
- **Navigation model:** navigationClass, menu, index, query, guidedTour, processClass, and externalNode.
- **Presentation model:** presentationGroup, presentationPage, text, textInput, anchor, fileUpload, button, image, inputForm, customComponent, presentationAlternatives, and selection.

- **Process Model**

- **Process Structure Model** – similar to the class diagram, derived from the navigation diagram.
- **Process Flow Model** – an activity diagram with userAction and systemAction elements.

Due to these additional artefacts on top of UML, UWE incorporates a higher learning curve. The UWE official website provides examples of various use cases to learn designing with UWE to improve its usability. The latest UWE profile is version 3.0, which was released on 18/06/2016, and after that, the official website [23] stopped releasing updates on 10/08/2016, stating, “*This website won’t be updated anymore. Last update: 2016-08-10*” on the footer, abandoning it to question the suitability for modern RiWAs.

UWE generally supports web applications and does not address the characteristics and features of the RiWAs. UWE-R [123] is a lightweight extension for the UWE to realise the RIAs, which contain new model-elements that inherit structure and behaviour from the UWE elements [15]. UWE-R does not modify UWE metaclass; instead, it provides an extension to conceive as add-ons to UWE to express RIA concepts [123]. The following extensions for UWE with model-elements are provided by UWE-R [123].

- **Navigations extensions:** Dialogue, RichNavigationClass, and RichNavigationLink.
- **Presentation extensions:** Canvas, Panel, TreePanel, TabbedPanel, AccordionPanel, Audio, Video, and DialogueWindow.
- **Process extensions:** AutonomousAction, ClientProcessClass, ServerProcessClass, and ControlMessage.

The UWE-R attempts to cover some aspects of the DC and the distributed logic of the RIAs/RiWAs and, thus, can be seen as a potential solution. The use of the extensions is demonstrated using Google Maps as a use case; however, the actual development of the designs is not discussed, limiting the UWE-R to a conceptual solution.

Even though UWE-R is a lightweight extension for UWE, UWE-R inherits the complexity of UWE and further increases it through additions. The UWE-R’s extensions do not significantly add to UWE, catering to the modern RiWAs, especially in client-side controllers, model, and DC handling. UWE-R has not developed beyond the initial research paper, which makes it an unusable solution; thus, its contributions are not taken into account in this thesis; instead, UWE is further reviewed and discussed where necessary when introducing new design elements and models. Even though model-elements and diagrams are presented, neither UWE nor UWE-R provides rules and guidelines to make them a complete methodology, limiting their scope to a modelling language.

### 3.3.2. WebML [107] and IFML (version 1.0, 2015) [124]

WebML can be used to design “a data-intensive Website amounts to specifying its characteristics in terms of various orthogonal abstractions, each captured by a distinct model” [107], and now it is converged to *Interaction Flow Modeling Language* (IFML), and the WebML official website is not further available. The IFML supports a wide range of application types, including desktop applications, mobile applications, traditional client-server applications, and RIAs; it has been adopted by OMG as a new standard and published in 2015 [124]. However, OMG has noted that IFML does not cover the modelling of the presentation issues [124], and its primary focus is on user interaction and control behaviour of the front end of software applications.

The IFML uses a new meta-model instead of extending the UML; hence, it incorporates new notations and models. Therefore, the learning curve can be considered higher; however, it comprises a small set of model-elements, which makes it not highly complex or challenging to learn. Since the IFML focuses on the interaction flows, it does not cover the split business logic and the DC aspects of the RIAs; hence, IFML can be considered incomplete for RiWAs designing. IFML’s official website [124] indicates that it was copyrighted from 1997 - 2018, indicating that it was not updated after 2018. Also, OMG’s IFML specification web page [125] states that the latest update of IFML is in 2018. Looking at the updates, the validity of IFML for modern RiWAs is questionable.

### 3.3.3. SysML (Version 2.0 Beta 1, 2023) [126] [127] [128]

SysML is a GPML for hardware and software systems, which is defined as a UML 2 Profile [126]. SysML provides the following models and model-elements.

- **Requirement diagram [new]:** performanceRequirement, interfaceRequirement, designConstraint, and designConstraint.
- **Structure Diagrams**
  - **Block Definition Diagram** (similar to UML class diagram) – a block can be a software, hardware, mechanical, or wetware (persons, organizations, facilities) component. The Block Definition Diagram specifies the system’s static structures to be used for Control Objects, Data Objects, and Interface Objects.
  - **Internal Block Diagram** (similar to UML composite structure diagram): Parts, Properties, Connectors, Ports, and Interfaces.
  - **Parametric Diagram [new]:** Constraints, Value Properties, and Parameters.
  - **Package diagram** (similar to UML package diagram): models, views, model libraries, and frameworks.

- **Behavior Diagrams**

- **Activity diagram** (similar to UML activity diagram)
- **Sequence diagram** (similar to UML sequence diagram)
- **State Machine diagram** (similar to UML state machine diagram)
- **Use Case diagram** (similar to UML use case diagram)

SysML features a comprehensive hypothetical *Griffin Space Vehicle Project* example to demonstrate the utilisation of the language. SysML is aligned with *Model-based Software Engineering* (MBSE) and *Model-based System Development* (MBSD); further, it discusses integrating with Agile SE best practices, resulting in an approach named *Agile MBSE*.

Even though the official website states that SysML “*has emerged as the de facto standard system architecture modelling language*” [126], it still lacks the essentials to model high-level architectural aspects of software systems. In the web context, the details related to tiers and platforms cannot be explicitly specified, and the communication-related elements are not much improved than standard UML; thus, the DC-related concepts cannot be depicted. As a GPML, SysML cannot be seen as a handy solution for RiWAs engineering.

### 3.3.4. Summary of the Low-level Design Methods Review

The review of low-level design solutions is analysed according to the following criteria, which are detailed at the beginning of the section.

- **C1:** Context
- **C2:** Comprehensiveness and simplicity of elements
- **C3:** Comprehensiveness and simplicity of communication types
- **C4:** Development and integration support

Table 3.5 Summary of low-level design methods review

Method \ Criteria	C1	C2	C3	C4
<b>UWE/UWE-R</b>	Web/RIA	Presentation, navigation, process	UWE-R tries to capture asynchronous communication	Presentation, class, sequence
<b>IFML</b>	GPML Interaction flows	front-end user interaction	Navigation and data flow	Presentation, events, and actions
<b>SysML</b>	GPML	Does not focus on web-related features		Integration with agile SE is highly discussed

As a GPML, SysML fails to cater to the specificity of the RiWAs. IFML’s scope is limited to front-end interactions and lacks the tools to support designing RiWAs. UWE is a good solution for web applications, and the UWE-R is a potential solution for RiWAs; nevertheless, they do not strongly realise the granularity of the distribution of the domain logic, client-side controller, and DC

connectors; hence, they lack simplicity and comprehensiveness. These design methods/methodologies are compared against the RiWAsML/RiWAsDM in Section 9.2 in the Evaluation chapter.

### **3.4. Research Work Related to RiWAs Designing**

Many research papers, books, and articles have been published since the beginning of the web era, discussing web-based application designing activities and related concerns. This section briefly reviews the UML-based solutions identified in the literature survey, focusing on these solutions' context, comprehensiveness, usability, and development support.

#### **Modeling Web Application Architectures with UML (1999) [129]**

This is an old publication, and even though the paper title says “*architecture*”, it mainly discusses some basic web page presentation-related design concerns. This paper is reviewed in this thesis to evidence that web-based systems designing has been looked into since the beginning of the web era. Web applications have evolved immensely; however, the design activities remain unsupported as the evolution of web-based applications and their development technologies have been much faster than web application modelling research.

#### **Systematic Design of Web Applications with UML (2001) [130]**

This is a book section that offers a systematic design method for web applications that takes into account the navigation space and presentational aspects of the web applications. The introduced method extends the previous approaches of Baumeister et al. (1999) and Hennicker & Koch (2000), extensively trying to model navigational and presentation aspects by providing multiple models and model-elements given below.

- **Navigation space model:** Navigational Class and Direct Navigability.
- **Navigation structure model:** Index, Guided Tour, Query, and Menu.
- **Static presentation model:** Frameset, Presentational class, Text, Anchor, Button, Image, audio video, Form, Collection, and Anchored collection.
- **Dynamic presentation model:** Window.

The presentation aspects are mainly Frameset-based, which is outdated and unuseful. In the case of navigational elements, how to map the navigational models to actual development is not discussed.

#### **Modelling the User Interface of Web Applications with UML (2001) [131]**

This work is based on the UWE's [23] design process and focuses on navigational and presentation aspects, offering the following models and model-elements.

- **Navigational model:** menu-based navigation modelling using index and menu elements.

- **Storyboarding process:** to identify user interfaces.
- **Presentation model:** decide where the navigation objects and access elements will be presented to the user. Elements used are target, window, frameset, frame, displays, links, and targets.

The authors discuss the method of producing these models under each model with some examples.

#### **A UML Profile for Modeling Framework-based Web Information Systems (2007) [132]**

This study produces a development-framework-based web information systems design method, enabling “*designers to produce diagrams that represent framework concepts, and developers (maybe, in the future, CASE tools) to quickly and directly translate these diagrams to code*”. It provides the following models.

- **Domain Model:** “*UML class diagram that represents classes of objects from the problem domain and their Object/Relational (OR) mappings*” [132].
- **Persistence Model:** a UML class diagram that represents the *Data Access Object* classes responsible for the persistence of domain objects.
- **Navigation Model:** “*UML class diagram that represents the different components that form the Presentation Logic tier, such as Web pages, HTML forms and action classes*” with stereotypes for the elements: page, template, form, and binary. A standard class is proposed to represent a “*class to which the Front Controller framework delegates the action execution*” [132].
- **Application Model:** “*UML class diagram that represents service classes, which are responsible for the implementation of use cases*”, which is more like an MVC-based model to implement business logic [132].

This paper does not provide examples or discuss how to use these models in actual development, limiting it to a conceptual solution.

#### **Designing Interaction Spaces for Rich Internet Applications with UML (2007) [133]**

This paper presents a design process for RIAs with 5 models and an activity.

- **Data Model:** describes the domain classes.
- **Use Case Model:** provides the context.
- **Task Model:** refines each use case by describing the activities performed during the use case, utilising a UML statechart diagram.
- **Interaction Space Model:** is a refinement of each task model. It describes the structural details of corresponding task flows where a user interaction is needed.
- **Guide Model:** is a refinement of the task model. It provides details on navigation and synchronization of user interaction from a software behaviour point of view.

- **Mapping to Implementation:** maps the design abstractions to the appropriate implementation according to the UML principles employing – tagged values, side effect actions, and transformations to the running system.

These models use some new notations; however, the models are based on UML class and statechart diagrams. The paper does not discuss the models and model-elements in detail and only demonstrates their utilisation using a simple use case.

#### **RUX-model (2007) [22]**

RUX-model has explicitly been introduced for the RIAs, and it also incorporates the adaptation of the legacy systems to RIAs. However, the RUX-model only focuses on the GUI expectations, eliminating the other aspects like the DC and business logic models of the RIAs, thus, making it incomplete. However, it can be combined with other methods like UWE [23] for better results [24]. From the learning curve perspective, the RUX-model provides an entirely novel approach and, therefore, incorporates a higher learning curve. Since the RUX-model lacks specifications, documentation, or tutorials, it cannot be considered a practical solution, and hence, it is not further examined in this thesis.

#### **OOH (2003) [134] and OOH4RIA (2008) [35]**

*Object-Oriented Hypermedia* (OOH) [134] extends the UML to support traditional web application designing, providing the following models and model-elements.

- **Navigation access diagram:** navigation classes, navigation targets, navigation links, and collections.
- Two-fold presentation layer
  - **Abstract presentation diagram**
  - **Composite layout diagram**

OOH4RIA [35] extends the OOH by introducing additional models, as stated below.

- **Conceptual Model**
- **Presentation Model:** to design GUIs
- **Orchestration Model:** an extension of “*UML state diagram which captures interaction patterns from presentation widgets as well as the navigation between screenshots of a RIA*” [35].
- **Transformation Model:** for model-to-model transformations and “*model-to-text transformations which establish a mapping from the models to the implementation*” [35].

The OOH4RIA authors discuss an “*OOH4RIA model-driven development process*” to assist in developing RIAs. The OOH4RIA-based development depends on the *Google Web Toolkit* (GWT); therefore, it is not abstract. Since OOH and OOH4RIA introduce many new models and model-

elements, they incorporate a higher complexity and learning curve. The OOH4RIA does not address the DC model or the distributed logic of the RIAs; thus, it can be seen as incomplete. The concept of the OOH4RIA is presented in a conference paper, and there is no implementation beyond the article discussing the application into real engineering to make it a more practical solution.

#### **UML-based Web Engineering - an Approach Based on Standards (book chapter, 2008) [84]**

This chapter discusses using UML for web application designing based on UWE [23]. It provides a comprehensive use case demonstrating the use of UWE models to design an online movie database. This book chapter does not add to the UWE; it's more of a tutorial that discusses the use of UWE in web engineering. Since UWE cannot cater to the modern RiWAs, the content of this book chapter is less valuable in contemporary RiWAs engineering.

#### **A Profile Approach to Using UML Models for Rich Form Generation (2010) [135]**

This paper proposes a UML profile to generate GUI forms using an *Object Relational Mapping* (ORM) profile based on the *Java Persistence API* (JPA) and supports user input validation. The authors produce a UML profile for JPA-based ORM and a profile for Hibernate input validator, which are mainly based on Java annotations. A GUI form builder profile, which can generate GUI forms, is introduced based on the new UML profiles. The main focus of this work is to support the design and development of the GUI forms and related CRUD operations on databases, and navigational, business logic, and DC aspects are not looked into.

#### **IAML – A Modelling Language for Rich Internet Applications (thesis, 2011) [42]**

This doctoral thesis presents *Internet App Modelling Language* (IAML), which attempts to capture client-side events and user interactions. This research also offers a CASE tool to improve the adoption of the IAML. The use of IAML is demonstrated using a hypothetical yet comprehensive use case named *Ticket 2.0*. The IAML has much more potential than the other available solutions and provides the following models and model-elements for RiWAs designing.

- **Logic model:** Function, Complex term, Parameter value, and Value.
- **Function model:** Function, Predicate, Boolean property, and Bulatin property.
- **Event-Condition-Action model:** Event, Condition, Action, and Operation.
- **Wires model:** Wire, Wire source, Wire destination, and Wireable.
- **Constructs model:** Changeable.

A complete specification of the modelling language is given in this work; however, the models and model-elements are briefly explained with fewer examples. And even though the use case demonstration provides a set of figures containing the IAML diagrams, they are not elaborate enough to gain a clear understanding. These limitations make the usability of IAML low.



The IAML mainly focuses on low-level presentation and processing, ignoring high-level modelling. The concept of the *Event-Condition-Action model* looks promising towards capturing client-side events handling; still, without enough discussions, it's not very usable. Also, the communication handling concerns are poorly addressed.

Looking at these weak points of the IAML, its usefulness in actual RiWAs engineering is questionable despite its conceptual strengths in low-level designing.

### **A UML 2.0 Profile Web Design Framework for Modeling Complex Web Applications (2014) [136]**

This paper proposes the following UML-based web application design models under three aspects.

- **Conceptual model**
  - **Information model:** provides a general structure of the web applications.
  - **Dynamic model:** identifies dependent processes to be extensively designed using activity and sequence diagrams.
- **Navigational model**
  - **Navigation Structure:** designs the nodes and objects that can be visited by navigating through the web applications.
  - **Navigation Space:** designs how the nodes and objects are reached while navigating.
  - **Navigation UML interaction:** captures how navigation classes are reached and appeared on the screen.
- **User interface model**
  - **Abstract Interface Model:** designs structural organization of the web applications.
  - **Graphical User Interface Model:** designs GUI-related aspects.

This paper does not provide a use case or discuss the development aspects, leaving it to be a theoretical solution.

### **Summary of the Review of the Related Research Work**

The available research publications mainly focus on the presentation and navigation concerns of web-based systems, ignoring the client-side processing and DC aspects. However, in the navigational and presentation aspects, they are not trying to capture the RiWAs' abilities to implement multiple related features on the same view and enable them to be accessed through different paths for different actors.

Among the identified related research work, IAML (refer to section 3.4.11) is a conceptually advanced solution; nevertheless, adopting it into real RiWAs engineering is uncertain without enough discussions and use cases to demonstrate its design and development support. Further, without high-

level design support, IAML is incomplete. Still, since the context of IAML is RIAs/RiWAs, it is further reviewed where necessary when introducing the RiWAsML in Chapters 5 and 6.

### 3.5. Chapter Summary: The Analysis of the Literature Review

The box-and-line approach has many usability issues due to its lack of formal syntax. ADLs are complex and less prevalent in the community of software engineers. UML-based methods/methodologies are popular and can address both high-level and low-level design aspects; therefore, they can be seen as a better option for a RiWAs design methodology. New DSMLs have been introduced to cater to the specificity of the RIAs/RiWAs, where most are extensions to the standard UML. Table 3.6 contains the summary of most related UML-like/UML-based solutions.

*Table 3.6 Summary of the review of related solutions*

<b>Solution</b>	<b>Context</b>	<b>Comprehensiveness</b>	<b>Usability</b>	<b>Dev and Integration</b>
<b>MDA with UML</b>	-General.	-Primarily low-level -No DC	-De facto standard	-None
<b>Arc<sup>42</sup></b>	-General	-Primarily low-level -High-level diagram -No DC	-UML-based -Primarily documentation	-None
<b>SAP's OOA/OOM</b>	-General	-TAM is high-level -Primarily low-level -No DC	-UML-based	-None
<b>ArchiMate</b>	-General	-No DC	-UM-like new lang.	-None
<b>C4</b>	-RiWAs	-Primarily high-level -No DC	-No formal syntax (box-and-line)	-None
<b>UWE/UWE-R</b>	-Web/RIA	-Low-level -No DC	-UML-based	-None
<b>IFML</b>	-RiWAs -Interaction flows	-Low-level -front-end -No DC	-UM-like new lang.	-None
<b>SysML</b>	-General	-Low-level -No DC	-UML-based	-Highly support agile-SE
<b>IAML</b>	-RIA/RiWA	-Primarily low-level	-UML-based	-None

Most available solutions are either general, do not support DC and related aspects, are limited to high-level/low-level designing, or do not offer development or integration support. Therefore, none of them can be considered comprehensive as they do not address all the general characteristics and essential features of the RiWAs.

## Chapter 4. Requirements for RiWAsML and RiWAsDM

---

This chapter sets the requirements for the proposed RiWAsML and RiWAsDM to realise the general characteristics and essential features of the RiWAs (refer to Section 2.3.5) based on the RiWAArch style (refer to Section 3.1.3). This chapter contains the results of Steps 1.2 and 1.3 of the RiWAsDM implementation process (refer to Figure 1.4 in Section 1.5.3) while achieving research objective 2 (refer to Section 1.4).

### 4.1. Attributes Required to be Satisfied by the RiWAsDM

This research aims to introduce a **simple** and **adoptable** design methodology (refer to Section 1.3); these attributes are selected to evaluate the practicality of the RiWAsDM and are generally explained in Section 2.2.4.2. The benefits of these attributes are discussed under motivation in Section 1.2.2. The simple and adoptable attributes are expected to be satisfied by the RiWAsML/RiWAsDM as requirements, and this section sets the context for them.

#### 4.1.1. Attr 1. Simplicity

Simplicity refers to the less complex nature of a modelling language that is improved by employing the *separation of concerns* principle, which appreciates decomposing a system and identifying and separating the modules for greater realisation, thus, management [18] [19]. A design methodology's DSML benefits from simplicity towards improving the understanding of the system elements and can be considered an essential attribute to enhance the methodology's quality [17] [8] [75]. A design language must separate and identify the models and model-elements to provide a profound realisation of the target systems. The RiWAsML is expected to answer the questions below, aligning with the RiWAArch style to maintain greater simplicity.

1. What are the **high-level** *Application* elements in a RiWA and their platforms in different tiers, and how do they communicate with each other?
2. What are the **high-level** elements inside the *Application* elements, and how do they communicate with each other and with the internal high-level elements of the other *Application* elements in the system?
3. What are the **low-level/development-level** elements of the high-level elements inside the *Application* elements, how do they communicate with other low-level/development-level elements, and what technologies/techniques can be utilised to develop them?

RiWAArch style is recognised as a style with high simplicity (refer to Section 3.1.3), and the RiWAsML is expected to be based on it towards ensuring high simplicity.

### 4.1.2. Attr 2. Adoptability

The following attributes are opted to evaluate the adoptability of RiWAsML/RiWAsDM into real-world RiWAs engineering.

#### 4.1.2.1. Attr 2.1. Comprehensiveness

Considering the assistance expected by a design methodology, the RiWAsDM is chosen to require the following features to reflect it as a comprehensive solution.

1. The RiWAsML should provide enough models and model-elements to design all the abstract high-level and low-level elements and their configurations aligning with the RiWAArch style (refer to Section 3.1.3). The comprehensiveness attribute is also related to the simplicity discussed in Section 4.1.1, and the expected model-elements and their configuration in different levels are stated below, based on the RiWAArch style to address the general characteristics and essential features of the RiWAs (refer to Section 2.3.5).
  - a. *Application* elements executed on dedicated platforms in different tiers and their configuration. The requirements R3, R4, and R6 in Section 4.3 identify the related models and model-elements.
  - b. Additional high-level elements, like databases and external services, which are utilised by the *Application* elements. The requirements R5 and R6 in Section 4.3 discuss the related models and model-elements.
  - c. For each *Application* element, the internal elements, their configuration, and their interactions with the internal elements of the other *Application* elements. The requirements R3, R4, and R6 in Section 4.3 discuss the related models and model-elements.
  - d. The low-level constructs of the *Application* elements' internal elements and their communication with the other low-level elements. The requirements R7 – R10 in Section 4.4 discuss the related models and model-elements.
  - e. The RiWAArch style realises the high-level elements and helps identify the development-level elements (refer to Section 3.1.3); therefore, the RiWAsML's high-level models and model-elements should be discovered based on the RiWAArch style.
2. Provide rules and guidelines for designing RiWAs using the RiWAsML and mapping the RiWAsML models to development.
3. Provide design and engineering approaches for RiWAsML-based designing.
4. Provide guidelines for adopting the RiWAsML-based designing into agile SE environments.

Suppose a single design methodology provides all these features; in that case, it will be considered a comprehensive solution by this thesis that offers sufficient assistance for RiWAs' agile model-driven development.

#### **4.1.2.2. Attr 2.2. Usability (Learnability and Readability/Understandability)**

If a solution is more usable, the adoptability of the solution can be considered high. The usability attribute tries to ensure the adoptability of the RiWAsML/RiWAsDM through the following two aspects.

##### **Learnability**

The RiWAsDM is required to assist in learning the RiWAsML to increase its usability and, hence, adoptability by offering the following features.

1. UML is the de facto standard modelling language, and the RiWAsML should be based on UML; hence, the RiWAs engineers with UML knowledge can quickly learn and use the RiWAsML.
2. The RiWAsML must deliver models and model-elements based on the RiWAArch style, and then the RiWAs engineers, with the knowledge of the RiWAArch style, can effortlessly learn the RiWAsML.
3. The RiWAsML should provide textual details to identify model-elements and their types instead of having many different graphical notations (refer to requirement R1 in Section 4.2.1), which assists in RiWAsML learning without referring to many new visual symbols. Also, the textual details must be self-explanatory to minimise the learning efforts.
4. The low-level elements of the RiWAsML should be more development-related, and the engineers with RiWAs development experience can quickly learn and apply the RiWAsML.
5. Provide rules and guidelines to learn how to use the RiWAsML, develop the RiWAsML-based designs, and integrate RiWAs designing activities into the RiWAs engineering.

##### **Readability/Understandability**

If the designs are more readable/understandable, they are much more usable and can quickly be adopted into engineering. To improve the readability/understandability, the RiWAsML should incorporate the following.

1. Include more textual details on the diagrams so the engineers can easily read and understand them instead of referring to graphical notations. The RiWAsML must attempt to use fewer graphical elements and more text-based information to assist in development (refer to requirement R1 in Section 4.2.1).
2. The low-level models and model-elements should be more development-oriented; therefore, engineers with RiWAs development experience can read and understand them easily.

#### **4.1.2.3. Attr 2.3. Development Support**

A design language is more adoptable if it supports the actual development without being limited to a conceptual solution. The RiWAsML/RiWAsDM should support RiWAs development via the following features.

1. The RiWAsML models and model-element should not be based on a particular development technology/technique and should be abstract enough to be adopted into RiWAs development. To ensure this, the RiWAsML is supposed to realise the general characteristics and essential features of the RiWAs based on the RiWAArch style, which is identified as an abstract formalism (refer to Section 3.1.3).
2. The low-level designs should include development-related details and be flexible to contain development tools-based specifics that match the language, frameworks, and libraries selected for the RiWAs development; thus, the designs can be easily converted into code.
3. The RiWAsML should provide rules and guidelines for designing RiWAs and map the designs into actual development.

#### **4.1.2.4. Attr 2.4. Integrability**

Contemporary software engineering projects largely follow agile practices, which tend to minimize design activities but would still benefit design thinking [30] (refer to Section 2.1.3). The RiWAsDM should provide guidelines by discussing integrating RiWAsML-based design activities into RiWAs engineering projects by enabling AMDD.

## **4.2. Requirements for Common Model-elements of RiWAs**

Before setting the requirements for high-level and low-level language elements, requirements for common elements' syntaxes should be identified and addressed. This section specifies requirements for the label element and communication channels, which are common for both high-level and low-level design.

### **4.2.1. R1 – Model-elements Naming Label**

Visual modelling languages use labels to set names for the model-elements in designs for identification purposes. RiWAsML exploits the naming label to improve the usability of the language. This section discusses the requirement for the RiWAsML's label under the following two aspects towards maintaining consistency and making the language less complex and highly usable.

#### **4.2.1.1. Symbol vs Text**

Modelling languages offer many model-elements to improve their simplicity and comprehensiveness. The general practice is to use different visual notations to identify the model-elements distinctly. For example, UML uses various shapes for distinct elements such as nodes, components, classes, and activities; UML-based languages introduce more notations using graphical notations like shapes, icons, and colours to specify the model-elements.

The problem with using visually different graphical notations is that when there are many model-elements, it is not easy to provide noticeably dissimilar visual elements to represent all the elements;

the notations become more similar and complex to distinguish between them, reducing the readability/understandability. Also, when there are many notations, the language users must remember them all and the concepts they represent, which negatively affects the language's learnability. Lower learnability and readability/understandability reduce the language's usability and decline the adoptability. Therefore, a better technique is required to effectively assist in RiWAsML's learnability and readability/understandability.

This thesis proposes using a text-based technique to address the said problem in the direction of reducing the number of distinct graphical notations yet providing sufficient details to read, understand, and use the elements correctly. Text is powerful in adequately interpreting the meaning, which helps in learning and understanding [25]; it is a matter of embracing a proper technique. Instead of adding more textual details to the designs using additional means like notes, RiWAsML is proposed to exploit the naming label by providing a new naming format for the model-elements. The related aspects are discussed in the following section.

#### 4.2.1.2. Name vs Three-segment Label

UML and UML-based languages generally use a single name to label the model-elements. Usually, meta-model-element classes are identified by the visual symbol, and the label is only used to set a name for the element. The stereotypes use additional tags to denote the type of the element; for example, the UML node element's <<device>> stereotype denotes hardware devices [137]. Since the RiWAsML proposes to use the label to identify the element class, the label requires a specialized format to hold all the necessary details. A design language includes many model-elements, and a robust labelling convention is needed to maintain consistency and readability/understandability across the model-elements.

This thesis proposes a label format with 3 segments to provide the necessary details towards maintaining consistency for higher readability/understandability. Also, this format is expected to help in learning the language without referring to a specification of graphical notations or remembering the visual symbols.

- **Element:** The *Element* segment is required to specify the element's class, for example, if the element represents a tier, an *Application*, a view, a component, or a connector. The user should be able to learn or read/understand the element's class by reading the label's element segment.
- **Type:** The *Type* segment is required to state the technological details. The possible values for the *Type* should be specified for each element class.
- **Name:** The *Name* segment is needed to assign a custom name to the element for identification purposes based on the scenario.

Section 5.1.1 introduces the notation of the RiWAsML *Label* element.

### **4.2.2. R2 – Communication Channels in RiWAs**

The communication channels between the system elements help define the system's configuration, denoted by the design diagrams. The RiWAsML should satisfy the following requirements to realise the communication channels with high simplicity and assist in readability/understandability.

#### **4.2.2.1. Line vs Arrow**

The first concern to address is the symbol of the communication channel. The candidates are line and arrow. The web communication between two elements at a given time is directional; hence, the arrow is more suitable. However, the core model of web communication is the request-response model; thus, it is always bi-directional. In that case, the use of a bi-directional arrow is questionable as the bi-directional communication can also be depicted using a line. RiWAsML appreciates the consistency of the syntax for improved usability and requires the employment of the same notations at the top-to-bottom levels of design. At the high-level, a single communication channel can denote the request-response communication pair using a single bi-directional channel; nevertheless, in the low-level design of the same elements, the same request-response pair might be taking place between different development-level elements and in such a case, two uni-directional channels are required to depict that.

For example, a single bi-directional communication channel is sufficient to denote the communication between high-level view and controller elements. Yet, at the low-level design, consider that the request is from the view's button to the controller's event handler, and the response is from the controller's different method to the view's output element. In this example, 2 directional arrows are required to model the low-level communication. Accordingly, to maintain consistency, the RiWAsML is required to use arrows to denote all the communication channels.

#### **4.2.2.2. Single Arrow Type With a Label vs Multiple Arrow Types**

There are many types of communication between the elements in RiWAs, like HTTP and DC, and they require different syntax to distinguish between. UML uses different types of lines, arrows, and also stereotype labels like <<deploy>> to illustrate relationships between the elements. The candidate options for RiWAsML are single arrow style with stereotype labels and unique arrow style per communication channel type.

Even though the RiWAsML appreciates the text-based identification over graphical symbols, in the case of communication channels, it was experienced while working on the use cases that the stereotype labels make the design untidy and less readable. Thus, for communication channels, the RiWAsML requires the use of different arrow styles to distinguish between the communication channels, as discussed in the following section.



### 4.2.2.3. Required Communication Channels Types

Web-based applications' primary communication mechanism is the request-response model. The RiWAs are also mainly based on the request-response communication model; nevertheless, they use other mechanisms to exchange data between views, models, and controllers. The communication between the elements can be realised through the RiWAArch style, as shown in Figure 4.1.

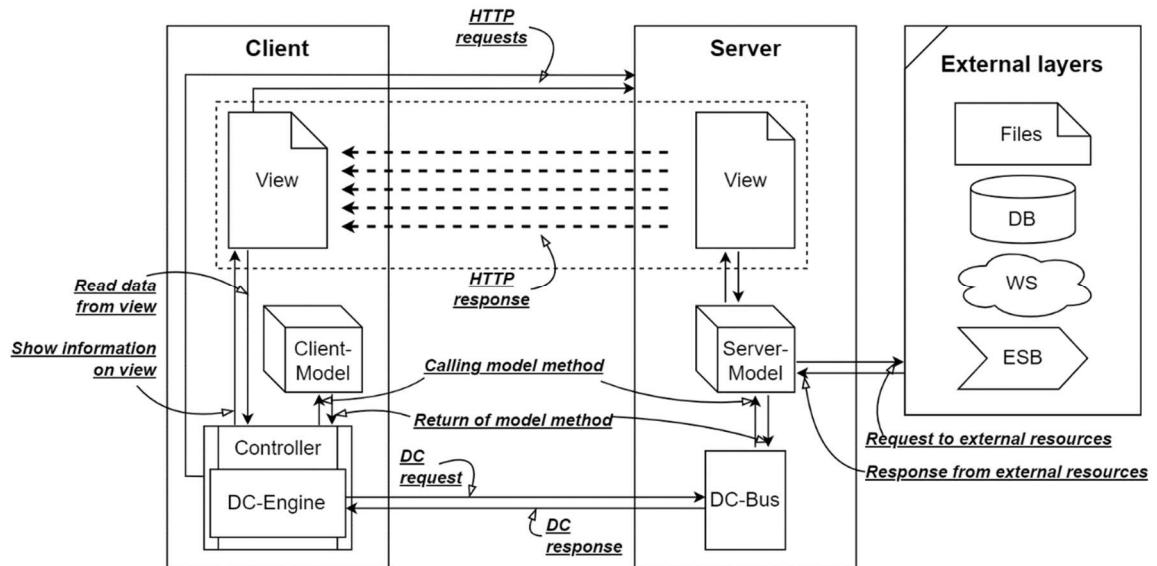


Figure 4.1 Communication channel types between elements in the RiWAArch style

According to the RiWAArch style, there are 5 main types of communication occurring between the elements in RiWAs as follows:

1. **Read data and show information:** the controller reads data from the view for processing and displays the information on the view. At the runtime, the view initiates the process by triggering an event on the GUI, which calls the relevant event handler on the view's controller, and the controller does what is needed and updates the view. This process can be realised by the request-response model as follows: the view requests from the controller to take an action, and the data read by the controller can be considered the data of the request; after processing the view's request, the controller updates the view with the results, which can be regarded as the response.
2. **Method calls and returns:** the controller and the DC-bus call the methods of the classes in the model and receive the results. This communication can be described by the request-response model, where the controller or DB-bus requests from the model by calling a method, and the model responds by returning the results. The parameters passed with the method call can be seen as the request's data.
3. **HTTP requests and responses:** the hyperlinks in views and redirecting commands of the controller send standard HTTP requests to the server. The server responds to the client by sending the next view and the related resources. Note that this is only for the RiWAs with browser-based clients.

4. **DC requests and responses:** the DC-engine requests from the DC-bus, and the DC-bus responds to the DC-engine using a compatible DC technology/technique like AJAX [100].
5. **Utilising external entities:** the RiWAArch style allows the model to consume external entities, and the communication between the model and an external entity can be realised with the request-response model, where the model is requesting data or service from an external entity and the external entity is responding with results. Even a case of model writing data to a file or database can be explained as follows: the model is requesting to write data, and the file or database responds with the status of data writing once completed.

By analysing the above types of communication, the following factors are derived to consider regarding the communication channels in RiWAs.

- **Direction:** all the types of communication between the elements in RiWAs are bidirectional and can be realised with the request-response model.
- **Type:** the type of communication should be out of the five types discussed.

The RiWAsML should provide suitable arrow notations to understand the direction and the type of communication channels straightforwardly. Section 5.1.2 introduces the notations for the RiWAs' communication channels.

### 4.3. Requirements for High-level Design of RiWAs

This section completes step 1.2 of the RiWAsDM implementation process given in Figure 1.4 in Section 1.5.3 by setting the high-level requirements for the RiWAsML. Requirements are set and discussed under different aspects: processing elements, view elements, additional high-level elements like data stores, and finally, models that utilise these elements. These requirements must address the general characteristics and essential features of the RiWAs (refer to Section 2.3.5), aligning with the RiWAArch style (refer to Section 3.1.3).

#### 4.3.1. R3 – High-level Processing Elements of RiWAs

This section discusses the processing elements identified, based on the RiWAArch style, required to design RiWAs' architecture, which should be provided with dedicated notations to illustrate them uniquely within the design. The processing elements and the infrastructures needed are shown in Figure 4.2, according to their parent-child relationships. Note that the visual symbols in this Figure are only used to distinguish them from each other and are not related to the RiWAsML notations.

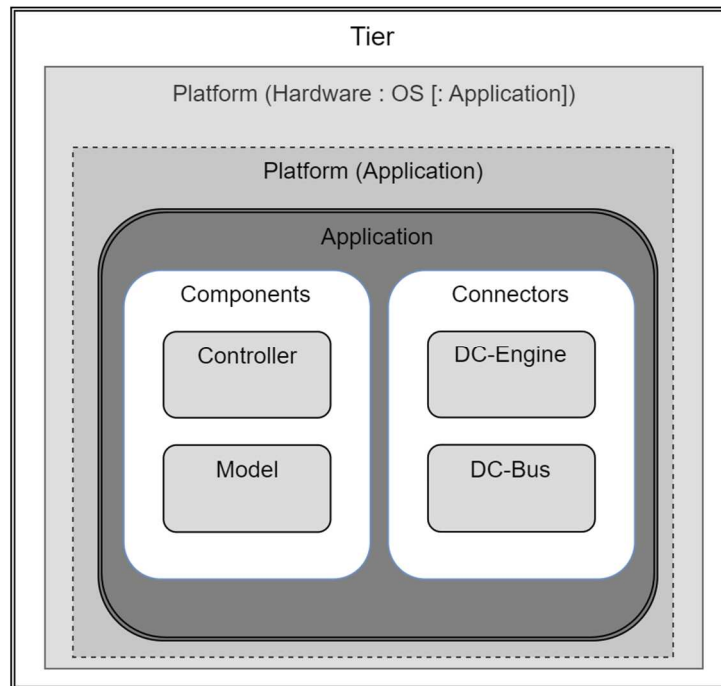


Figure 4.2 Processing elements in RiWAs

The requirements for the processing elements in Figure 4.2 are discussed in depth in the following sub-sections.

#### 4.3.1.1. Application Element

The general characteristics of RiWAs (refer to Section 2.3.5.1) realise that a RiWA is a collection of *Application* elements. This thesis defines the *Application* element as follows.

An *Application* element in a RiWA is an element that is comprised of processing elements such as components and connectors and is independently executable/runnable on a dedicated platform. An *Application* element communicates with the other *Application* elements in a RiWA to accomplish functions, and the role of the *Application* element depends on the tier to which it belongs. The client-side *Application* elements may contain views.

The *Application* element is the highest-level processing element in RiWAs and essentially requires a model-element. Some use cases to explain the *Application* element's behaviour are discussed below, emphasising the requirement for a dedicated model-element for the *Application* element.

- It is possible to use multiple *Application* elements of a RiWA on a single device. Consider a taxi booking RiWAs. A driver may install both the driver's and the passenger's apps on a mobile phone to use them when needed as a driver or a passenger.
- A user may work with different types of *Application* elements on multiple devices. For example, a user can utilise a RiWA by a browser-based app on a laptop and a mobile app on a mobile phone.

From the engineering point of view, the *Application* elements exhibit the following behaviours.

- It is possible for multiple *Application* elements to implement the same functions. For example, the user's browser app and mobile app are supposed to deliver the same set of functions; hence, the same logic should be implemented in both *Application* elements' components. However, since they should be executed on different platforms, they must be developed using different technologies; therefore, it is vital to show them as separate elements in the design.
- The concept of the *Application* element suggested by this thesis can be seen as a wrapper for a group of elements that are developed using dedicated technologies. Different *Application* elements may have different sets of internal elements, such as views, components, and connectors. It mainly depends on the type of the *Application* element (client-side, server-side, browser-based, mobile, micro-service, etc.)
- Internal elements of an *Application* element communicate with each other and with the internal elements of other *Application* elements.

The RiWAsML's syntax for the *Application* element should assist in modelling these behaviours. Section 5.2.3 introduces the RiWAsML's *Application* element,

#### 4.3.1.2. Tier Element

Layered architecture style improves the simplicity and visibility of a system by separating the elements into layers/tiers based on their roles [138]. Distributed systems like web-based systems highly benefit from layered styles like 2-tier client-server architecture, 3-tier, and n-tier architectures as the tiers also help understand the deployment of the elements and their communication technologies. As web-based applications, RiWAs' elements are at least separated from client and server tiers. Advanced RiWAs' elements are distributed across many tiers for various purposes like routing, load balancing, caching, and external service usage, and each *Application* element in a RiWA should belong to a particular tier. This thesis defines the tiers for RiWAs as follows.

The tier is the highest level of separation, which logically separates architectural elements by grouping similar elements. This separation is mainly based on the elements' role/purpose and distribution rather than technological aspects. The tiers help organise the elements, providing a clear understanding of the relationship between the containing elements, their role as a group within the tier, and their geographical or platform distribution.

The RiWA design probably benefits from denoting the tiers for improved simplicity, and it will also enhance the readability of the design. Refer to the use case in Section 8.1.1 for an example of tiers enhancing the readability of a design. Section 5.2.1 introduces the RiWAsML's *Tier* element syntax.

#### 4.3.1.3. Platform Element

Each *Application* element in a RiWA runs on a dedicated platform. Also, storage elements like databases require platforms for deployment. Hence, it is essential for RiWAs' high-level design to include the platform details to understand the deployment of the elements. The platform is defined by this thesis as follows.

The platform provides the environment for elements such as *Application* elements and databases to deploy and run.

The platform is a complex concept that involves the three levels below.

1. **Hardware:** a device like a computer, mobile phone, or any other physical device capable of running software systems such as the devices in the *Internet of Things* (IoT). In the case of IoT, the device can be even a TV, a vehicle, or any other custom device.
2. **Operating system (OS):** The OS is required to manage the hardware resources at the hardware platform level. Desktop or laptop computers use OSes like Windows or Linux, mobile phones use OSes like Android or IOS, and other hardware devices may even use custom OSes to hide the hardware layer's complexity and provide a platform for the runnable elements.
3. **Application-level virtualization:** Some *Application* elements require software – like web servers, DB servers, runtimes like JRE or .NET, or tools like browsers – to run within a device. These application software provide the virtualised environment the *Application* elements require to run.

The nature of the platform of an *Application* element depends on many factors, as discussed below.

- The device and OS are usually tightly coupled, and application-level virtualization is optional. For example, a mobile device with its OS provides the platform for a mobile app without requiring an application-level virtualization platform.
- A single device+OS can accommodate multiple application-level platforms. For instance, consider a computer running both a web server and a DB server, which are two different application-level platforms.
- In the case of having multiple application-level platforms within a device, they can be different tiers. Considering the previous example, the web server is in the application tier, and the DB server belongs to the data tier.
- When using resources like cloud-based services for deploying the *Application* elements or data stores, these services can be considered application-level virtualization environments, and their hardware and OS platform details may not be required.

The RiWAs' platform syntax should be able to realise these levels and factors. Section 5.2.2 introduces the notation of the RiWAsML's *Platform* element.

#### 4.3.1.4. High-level Components

As per Figure 4.2, an *Application* element comprises components and connectors. This thesis defines the components as follows.

A component is a processing element within an *Application* element that implements event handling and/or business logic.

Since the RiWAs implement client-side event handling and split business logic (refer to general characteristics in Section 2.3.5.1), all client-side and server-side *Application* elements contain components, and related models and model-elements are required to design the associated aspects. It is essential to realise the components using MVC (refer to the essential features in Section 2.3.5.2); hence, the components are discussed under the controller and model, aligning with the RiWAArch style. Section 5.2.4 introduces the RiWAsML's syntax for the *Component* elements.

#### High-level AppControllers Element

The controller is a controversial element in the MVC style, and many frameworks exploit it differently. For example, server-side frameworks like Laravel use the controller to handle incoming requests [139]; in such a case, the controller can be considered a connector instead of a component. Aligning to the original definition by the Smalltalk-80 [140] [141], the RiWAArch style uses the controller for events handling triggered on the views [12] [105], considering it as a component. Based on that viewpoint, the high-level model-element named *AppControllers* is defined as below.

The *AppControllers* is a high-level element in a client-side *Application* element that implements event handling for views.

Client-side event handling is a general characteristic of RiWAs (refer to Section 2.3.5.1); thus, a controller is an essential element in RiWAs' design. The *AppControllers* element must satisfy the following features.

- At the high level, the *AppControllers* element should realise all the controllers within a particular client-side *Application* element.
- The *AppControllers* elements implement the event handlers for the views; therefore, the *AppControllers* elements communicate with the high-level views element (refer to Section 4.3.2).
- *AppControllers* elements may use the client-model to process data based on business logic when handling events.
- *AppControllers* elements may communicate with the server-side *Application* elements via the DC-engine.

The RiWAsML is required to realise these high-level features of the *AppControllers* elements and should be able to model them precisely.

### High-level AppModel-element

The RiWAArch style contains two models by realising the RiWAs' split business logic general characteristic (refer to Section 2.3.51): the client-model and the server-model. Both implement the business logic and considered components, and this thesis defines the *AppModel* element as follows.

An *AppModel* is a component in a client-side or server-side *Application* element that implements business logic. All the *Application* elements in a RiWA contain a single dedicated *AppModel* element. The *AppModel* also considers the runtime data and data structures utilised by the logic.

The RiWAsML is supposed to offer notations for high-level *AppModel* components to design the following aspects aligning with the RiWAArch style.

- The *AppModel* element's type should be defined as client-model or server-model.
- *AppControllers* element can utilise the client-side *AppModel* element by calling its methods.
- DC-bus can utilise the server-side *AppModel* element by calling its methods while handling the DC requests.
- For browser-based RiWAs, the views may use the server-side *AppModel* element by calling its methods before being sent to the browser.
- Server-side *AppModel* elements can communicate with the databases to perform CRUD operations.

#### 4.3.1.5. High-level Connector Elements

Connectors are the architectural elements that realise a software system's communication and related aspects. RiWAs generally use DC, and it is essential for RiWAs to have DC handling connectors (refer to Section 2.3.5). The RiWAArch style uses the DC-engine on the client-side and the DC-bus on the server-side to implement a comprehensive DC connector pair [12]. Based on these aspects, this thesis defines the *Connector* element as follows.

A *Connector* element is a processing element dedicated to implementing communication logic. The DC-engine is a client-side *Connector* element that sends DC requests and accepts DC responses; the DC-bus is a server-side *Connector* element that accepts DC requests and sends DC responses.

RiWAsML must offer a high-level *Connector* element to assist in designing the following features.

- DC-engine is considered the *AppControllers* element's internal *Connector* element, utilised to send DC requests to a DC-bus and accept the DC responses.

- DC-bus implements the DC request handlers to accept the DC requests.
- DC-bus reads the data from the DC requests and passes it to its *Application* element's server-model for processing. Then, the result returned by the server-model is sent to the requested DC-engine as the response.

Section 5.2.5 introduces the notations for the RiWAsML's *Connector* element.

### 4.3.2. R4 – High-level Views Element

RiWAs generally use rich GUIs, which are realised as views by the RiWAArch style based on the MVC style (refer to Section 2.3.5) [12] [105]. Regarding MVC, the views are for the presentation and thus cannot be seen as components.

In the browser-based RiWAs implementation, the views are implemented as web pages; nevertheless, executable code can be written on the views. For example, on a web page – either HTML, PHP, ASP, or any other compatible type – some JS code can be written to be executed on-page-load to update the view with some visual effects. In such cases, it should be understood that these code is not a part of the view, even if the code is written on the same web page. In this example, the code should be executed upon an event, the on-page-load event, which is supposed to be a part of the controller that handles the events. Maybe the logic that generates the view's visual effect is based on some business logic, which should be implemented in the *AppModel* element that the *AppControllers* element invokes. Likewise, it is essential to understand that processing is necessarily a role of the components, and the views cannot be considered as components.

This thesis defines the *View* element for the RiWAsML as follows.

The *View* element is a client-side element that implements the rich GUIs for the users to interact with the RiWAs. The content on a *View* element can be dynamically generated or updated by its controller.

The high-level view element of the RiWAArch style represents all the views in the RiWA; therefore, the RiWAsML requires a high-level wrapper, which realises all the actual views on the high-level design. Note that this high-level element should be named “*Views*” in plural form. Based on the definition of the *Views* element, the high-level *Views* element is defined as below.

The high-level *Views* element is an abstract element which realises all the views within its *Application* element.

The RiWAsML has to assist in capturing the following views-related features.

- The *Views* element communicates with the *AppControllers* element by invoking the event handlers when the user triggers the events.



- The event handlers do the needful processing and dynamically generate content on the *Views* element or update the view by changing or removing content.

Section 5.2.6 introduces the RiWAsML's notation for the high-level *Views* element.

### 4.3.3. R5 – Additional High-level Elements

The high-level design may include some more entities than the abovementioned ones. The RiWAs essentially use databases (refer to Section 2.3.5.2) and may also use entities like users, files, web services, Enterprise Service Bus (ESB), and networks. These elements should be able to be nested into platforms in appropriate tiers as needed.

Moreover, the RiWA architecture benefits from including additional textual details and requires the use of proper elements for that purpose. A note is a potential element, and the RiWAsML should provide a note element to include related textual details on the designs.

Section 5.2.7 introduces the syntax for these required additional elements.

### 4.3.4. R6 – High-level Design Models

This section identifies the models required to provide abstract interpretations of different high-level aspects of the RiWAs and sets the requirements for them.

#### 4.3.4.1. Level 1 Applications Model Requirements

A RiWA is generally a collection of applications and databases (refer to Section 2.3.5). A model is required to realise all the *Application* elements and databases in a RiWA and show their configuration; it is proposed to be named the *Level 1 Applications* model (*L1 Applications* model, in short). The requirements for the *L1 Applications* model are as follows.

- The *L1 Applications* model should include all the *Application* elements in a RiWA.
- The *L1 Applications* model should also recognise the platforms the *Application* elements run and the tiers they belong to.
- The *L1 Applications* model should realise the data stores, external services, and also their tiers and platforms where necessary.
- The *L1 Applications* model should realise the communication between the *Application* elements by means of HTTP or DC and other standard communication between the *Application* elements and the data stores or external services.

Section 5.3.3 introduces the RiWAsML's *L1 Applications* model and its UML profile.

#### 4.3.4.2. Level 2 View-Process Model Requirements

MVC-based modularization and DC-handling connectors are essential features for RiWAs (refer to Section 2.3.5.2). RiWAs need a model to realise the high-level MVC modules, DC connectors, and

their configuration within the *Application* elements identified by the *L1 Applications* model. The required model is named the *Level 2 View-Process* model (*L2 View-Process* model, in short), which will help understand the high-level elements necessary to develop each *Application* element. The *L2 View-Process* model should satisfy the following features.

- For a given client-side *Application* element, the *L2 View-Process* model should assist in capturing the *Views* elements, *AppControllers*, client-side *AppModel*, *DC-engine*, and their configuration.
- For a given server-side *Application* element, the *L2 View-Process* model should realise the *app-server-model*, *DC-bus*, and, in the case of browser-based RiWAs, the server-side *Views* element. The configuration between these elements should be realised.
- The communication between the elements within the *L2 View-Process* model and the other *Application* elements of the RiWA should be realised.

Section 5.3.4 introduces the RiWAsML's *L2 View-Process* model and its UML profile.

#### **4.3.4.3. Level 1+2 Architectural Model Requirements**

RiWAs need an abstract architectural model to realise all the high-level aspects discussed under the *L1 Applications* model and the *L2 View-Process* model within a single model where necessary. This model may assist in capturing all the high-level details within a single design when the RiWA is not large and complex and a single diagram is sufficient. Section 5.3.5 introduces the *Level 1+2 Architecture* model (*L1+2 Architecture* model, in short) and its UML profile to design RiWAs' high-level aspects in a single diagram.

## **4.4. Requirements for Low-level Design of RiWAs**

This section performs Step 1.3 of the RiWAsDM implementing process (see Figure 1.4 in Section 1.5.3) in the direction of fulfilling research objective 2. This thesis mainly focuses on the structural modelling aspects of the RiWAs based on the RiWAArch style; therefore, in low-level design, the RiWAsML attempts to look into the detailed structural modelling requirements of the high-level elements proposed in Section 4.3.

The RiWAsML's low-level designs should provide enough details to support the development of the lowest-level elements, which participate in executing processes. The RiWAs generally offer rich GUIs (refer to Section 2.3.5.1) for users to interact more with the system towards improving the user experience, and the processes are initiated by the user's actions when interacting with the rich GUIs of the views. Therefore, rich GUIs play a vital role in the perspective of users and engineers. Considering this, the RiWAsML tends to be view-centric in low-level designing. The following sections set requirements for low-level model-elements and models to be satisfied by the RiWAsML.

#### 4.4.1. R7 – Low-level Modelling of Views

The high-level *Views* element in client-side *Application* elements (refer to Section 4.3.2) represents all the views in a RiWA, and low-level model-elements are required to capture the features of the individual view within a high-level *Views* element.

RiWAs offer rich GUIs in general (refer to Section 2.3.5.1); under view designing, there are two main traits: the content of the GUIs and their aesthetic considerations like colours, placements, animations, and responsive layouts. This research focuses on designing the content-related features required to implement interactions and functionalities instead of aesthetic designing. Views need proper low-level model-elements to capture the details required to implement the functionalities in the direction of improving the user experience. The requirements for different aspects of modelling low-level details of the views are discussed in the following sub-sections, aligning with the definition for view given in Section 4.3.2. The notations for the RiWAsML's view-related elements are introduced in Section 6.1.1.

##### 4.4.1.1. GUI elements

Most available solutions, like UWE [142], try to define the GUI elements explicitly, which is impractical. Modern GUI development concepts/tools – such as Bootstrap, JS-based widgets, GUI component-based development, and mobile GUI elements – provide a wide variety of GUI elements and defining them all for a modelling language is not feasible. Giving space to accommodate the available GUI development tools, the RiWAsML is expected to offer abstract GUI element classes instead of explicitly specifying the GUI elements.

GUI elements are defined for the RiWAsML as follows.

GUI elements are abstract visual element classes used to implement functions in views.
--

This thesis identifies the following abstract GUI element classes for the RiWAsML.

##### Input/Output Elements

The RiWA users primarily enter data and commands to the rich GUIs' input elements and receive information on the views' output elements in return. The input and output GUI elements are defined as below.

Input elements are the GUI elements that allow users to input data and commands to the RiWAs. Output elements are used to show information to the users on the GUIs.
---

The data input elements are textboxes, radio buttons, etc.; command input elements are buttons, enter key, drag and drop areas, etc. Some input elements can act as both data and command elements by triggering events. For example, when selecting the gender by using a radio button, it can trigger a

command to show the title compatible with the gender. Output elements can be tables, lists, images, etc. Advanced visualizing elements, such as media players and graphs, are available, which can be considered output elements. These output visual elements are bound with data, further discussed in section 4.4.1.3. In some cases, some GUI elements can act as input and output elements; for example, some data is shown on a check box group and asks the user to select the options.

The RiWAsML is supposed to offer model-elements for the input and output elements that participate in processes.

### **Containers/Sections**

The rich GUIs often implement multiple related features on a single view, and different containers/sections may be used to visually separate the content on a large GUI. For example, consider a view performing the CRUD operations on some items; on the top, there can be a form to enter data, and below the form, the list of items can be displayed. If sorting and filtering functionalities exist, the relevant GUI elements can be shown at the top of the list in a dedicated container/section, visually grouping and separating them from the list and the form. It is vital that the GUI elements are appropriately grouped, not only to improve user experience but also to increase the clarity of designs towards supporting the development. Based on this discussion, the containers/sections are defined as follows.

Container/section are GUI elements to visually group the view elements to improve the user experience, as well as increase the clarity of the view designs for higher usability towards support in development.

### **Popups and Toggles**

Since RiWAs are capable of partial view rendering, they tend to implement many functionalities on the same view instead of making the user navigate through a series of pages to complete a single task based on the traditional page sequence paradigm [143] [50]. Further, some RiWAs, like Google sheets, implement the entire app on a single page. In that setting, popups and toggles are widely used to manage the content while keeping the user in the same view yet performing more related functionalities. Popups/toggles are defined for RiWAsML as follows.

Popups/toggles are GUI widgets that show/hide or enable/disable content sections to implement more features on the same view with improved user experience.

When designing views, including details related to the popups/toggles would be beneficial when they contain GUI elements, which are part of functions. Moreover, it is necessary for a design to include details of the results of the actions performed on the popups/toggles, primarily if the results affect

the main view. For example, a popup allows changing the font of the text on the main view, and when the font is selected and applied in the popup, the font of the text on the main view is changed.

### **Viewparts**

In RiWAs, different types of users may use the same particular view, which allows them to perform a group of related functionalities, where some user types are allowed to perform additional functions based on the authorization level. For example, consider a list of items displayed on a view of an online shop. The buyers may see the list of items with their details, whereas the admins see more elements to perform special functions, for example, editing and updating items' details. In such cases, instead of developing multiple views per user type, the practice is to employ the advancement of the rich GUIs and DC development technologies to implement all the functionalities on a single view and show/hide or enable/disable the additional features based on the current user's authorization level. This thesis introduces the concept named "Viewpart", which is defined below, to provide the required support in such cases.

A *Viewpart* is a GUI section which implements functions for a particular user type.

The RiWAsML should assist in designing views with *Viewpart* elements specifying their target user types.

### **Shared Content**

In multi-view RiWAs, it is common for the views to share some content, such as headers, footers, menus, and sidebars. The shared content can be developed as separate GUI widgets and included in the sharing views. It is vital to identify them and design carefully toward improved simplicity, reusability, extensibility, and modifiability. Therefore, the RiWAsML is required to provide sufficient model-elements to design shared content in the direction of supporting its development.

Section 6.1.1.1 introduces the GUI-related elements for the RiWAsML.

#### **4.4.1.2. Navigation**

Designing/developing navigation in websites and web applications has been a concern, and different methods have provided solutions. In a multi-view RiWA, the navigation between the views can be complex, especially when the views contain *Viewpart* elements and different user types can navigate to these views via different routes (this is further discussed in Section 4.4.4.1). The RiWAsML should address these concerns since it helps identify the views with *Viewpart* elements for different actor types and supports their actual development.

RiWAs may use several navigation techniques. Link-based navigation is the typical approach that allows users to navigate via links. On the browser-based RiWAs, links can be implemented in hypermedia – including text, images, and videos – to navigate to a different place on the same view

or to a different view, principally without requiring any processing since it's a browser feature. The menu-based approach is the standard link-based navigation implementing method, where a menu is a list of links. Additionally, the RiWAs may also use process-based navigation, where the user is navigated as a part of a process. For example, in a shopping app, when the user adds items to the shopping cart and clicks the checkout button, the user is navigated to the checkout view to continue the process.

The RiWAsML should provide enough models and model-elements to capture the said navigational aspects, satisfying the following requirements.

- The RiWAsML should realise link-based and process-based navigation.
- For link-based navigation, the shared menus/links should be captured.
- Different navigation paths to navigate to common views should be denoted.
- Different navigation paths for different user types should be realised.

Section 6.1.1.2 introduces the RiWAsML's navigation-related elements.

#### **4.4.1.3. Data**

Views capture data to send to server-side *Application* elements and/or show information using the data received from the server-side *Application* elements; hence, the views associate datasets. Components are supposed to process these captured data and produce information to show on the GUIs. In that setting, the data to be captured by a view or information to be shown in a view are directly related to the data processed by the components. It is crucial to have a clear understanding of the data related to the views, and the RiWAsML should contain enough tools to present data-related aspects – like data types and structures – on view designs.

#### **4.4.1.4. Related Controller**

According to the BAW-MVC [105] used in the RiWAArch style [12], each view should have a dedicated controller; therefore, the controllers are directly related to the views, and modelling this relationship is essential to assist the development, and the RiWAsML must have sufficient models and model-elements to design the relationship between the view and its controller. The controller is a component; hence, the discussions on the controller and related aspects are given in section 4.4.2.1.

### **4.4.2. R8 – Low-level Modelling of Components**

There should be models and model-elements in the RiWAsML to design the internal structural aspects of the RiWAs high-level components discussed in Section 4.3.1.4. This section discusses the requirements related to the components' low-level aspects in the direction of supporting their development. The following requirements are set to be satisfied by RiWAsML's low-level component design.

1. A component should be designed using independent elements similar to UML classes.
2. It is necessary to show which high-level component's internal implementation is designed by the low-level design models.
3. The low-level implementation details of the component's interfaces should be depicted. For example, which method of which class implements a particular interface?

The requirements specific to the low-level component elements named *ControllerClass* and *ModelClass* are discussed in the following sub-sections.

#### 4.4.2.1. Low-level Modelling of AppControllers

The high-level *AppControllers* element (refer to Section 4.3.1.4) contains the individual *ControllerClass* elements of a given client-side *Application* element. The *AppControllers* element is an active and complex client-side component in RiWAs, which directly communicates with the other elements, as discussed in Section 4.3.1.4. The individual *ControllerClass* elements may communicate with the other low-level elements, aligning with the high-level communication, as discussed below.

**Controller-view communication:** according to the BAW-MVC style [105] used in the RiWAArch style [12], each view is coupled with a dedicated controller, and a *ControllerClass* element implements the client-side event handlers for a particular view. Available solutions like IAML [42] try to define a set of events for RiWAs explicitly. However, due to the availability of a wide variety of GUI elements for different platforms (as discussed in Section 4.4.1.1), the RiWAsML is not expected to define events for the *ControllerClass* elements explicitly. Instead, the RiWAsML requires a suitable technique for the controller design to accommodate platform/tool-specific events as needed, according to the development technologies.

**Controller-client-model communication:** when a *ControllerClass* element needs to process view data using business logic, the *ControllerClass* element should be able to utilise the client-model. The controller can call methods in the client-model by passing the data read from the view, catching the results returned by the client-model, and showing them on the view.

**Controller-server-application communication:** The *AppControllers* element contains the DC-engine, which is utilised for communicating with the server-side *Application* elements. The controller's DC-engine implements the DC response handlers to capture the DC responses and process the results sent by the server-side *Application* elements.

The RiWAsML is required to provide model-elements to capture all these low-level details related to the *AppController* element's *ControllerClass* elements; Section 6.1.2 introduces the syntax for related elements.

#### 4.4.2.2. Low-level Modelling of AppModel

An *AppModel* is an abstract unit that implements the system's business logic as a black box exposed to the outside via interfaces. An *AppModel* element is a collection of classes named by the RiWAsML as *ModelClass* elements, and they can be wrapped into packages. The *ModelClass* elements' interfaces can be implemented as public methods, exposing the *AppModel* to the other elements. Designing low-level aspects of the RiWAs *AppModel* elements is expected to be assisted with model-elements and models to capture the internal mechanism of both the server-model and client-model and their communication with the other elements. Section 6.1.3 offers the notations for the related elements.

#### 4.4.3. R9 – Low-level Modelling of Connectors

The RiWAsML is required to provide models and model-elements to design the internal mechanisms of the high-level *Connector* elements discussed in Section 4.3.1.5. The low-level modelling requirements for the connectors are discussed in the following sub-sections.

##### 4.4.3.1. DC-Engine

The *AppControllers* element uses the DC-engine to send DC requests to the server and receive the responses. The RiWAsML should be able to realise the relationship between the controller and the DC-engine and provide model-elements and models to design the DC-engine-related features as given below.

- Upon *ControllerClass* elements' need, the DC-engine sends DC requests to the DC-bus.
- DC-engine sets a DC response handler to catch the DC-bus's response.
- The *ControllerClass* elements should be able to utilise the data in the DC response captured by the DC-engine.

##### 4.4.3.2. DC-Bus and EndpointsCollections

The RiWAsML is expected to decompose the high-level DC-bus element to low-level elements, which can be realised based on the OODD. The DC-bus implements the APIs for DC-engines in a RiWA, which are usually developed as endpoints using a compatible technique/technology like SOAP [144] or REST [145]. Therefore, the RiWAsML names the internal elements of the DC-bus as *EndpointsCollection* elements and defines them as follows.

An *EndpointsCollection* element implements the APIs for a DC-bus based on the OODD practices. A DC-bus is comprised of a related set of *EndpointsCollection* elements.

The RiWAsML should provide models and model-elements to realise the following features of the DC-bus and *EndpointsCollection* elements.



- The relationships between the *EndpointsCollection* elements within a DC-bus should be depicted based on the OODD.
- An *EndpointsCollection* element's endpoints accept the DC-engine's DC requests, and an endpoint should be able to denote the data accepted from the request.
- An endpoint can call the methods in the server's *ModelClass* and catch the returned results.
- An endpoint can send the response to the DC-engine with a dataset, and the details of the dataset, such as data type, should be denoted in the design.

Section 6.1.4 introduces the required model-elements and their notations for the RiWAsML.

#### 4.4.3.3. Data and Structures

Connectors may use different types of data to communicate between the client and the server, such as URL query string, plain text, wrapping methods like XML or JSON, or files [10]. The connectors should be able to depict the data types used in the designs.

#### 4.4.4. R10 – Low-level Design Models for RiWAs

The RiWAsML is required to offer low-level design models for the detailed designing of all the high-level elements: *Views*, *AppControllers*, *AppModel*, and *Connector* elements. This section identifies the models needed to provide abstract viewpoints of different low-level aspects covering the general characteristics and essential features of RiWAs (refer to Section 2.3.5) based on the RiWAArch style (refer to Section 3.1.3) and sets the requirements for them.

##### 4.4.4.1. View-Navigation Model and View Model

Since the RiWAs combine related functions and develop them on common views, and such a view can be navigated via various routes by different user types, a model is required to capture these features detailed in Section 4.4.1.2. To explain this further, consider the *Item* view of a shopping RiWA. The *Item* view can implement the *view item details* function for the buyers, and they can navigate to the view via an item list on the *browse items* view. The same *Item* view can implement the *edit item* function for admins, where they can navigate to the *Item* view via the *manage items* view.

The RiWAsML proposes two models to capture the navigational and rich GUI aspects of the RiWAs.

**View-Navigation model:** The RiWAsML requires this model to design the navigational characteristics of the views in a RiWA. The *View-Navigation* model is required to assist in detailing the following aspects.

- **The set of views:** the *View-Navigation* model should include all the views in a RiWA, helping to decide to add/remove views based on the functions they implement.

- **Common views:** identify the common views that implement multiple related functions for different user types. The engineers can decide to merge or split views towards improving the user experience and managing the development.
- **Navigation to each view:** the model should assist in capturing the navigation paths to all the views, including multiple navigation paths to common views, which are used by different types of users.
- **Navigation type:** engineers can decide whether the navigation should be linked-based or process-based in the direction of improving the user experience.
- **Shared navigation elements:** the model should help identify the shared navigation elements, such as main menus.

**View model:** The RiWAsML needs this model for the detailed design of the views included in the *View-Navigation* model. Since rich GUIs are important in improving the user experience (refer to Section 2.3.5.1), it is vital to understand all the functions to be implemented on a view and design the view to support the development of the functions. The following requirements are set for the *View* model.

- The *View* model is required to capture the abstract GUI elements discussed in Section 4.4.1.1 to implement the view's functions, considering the users' interactions with the view and how these interactions should be handled.
- The *View* model is required to assist in denoting the view's interactions with its controller (refer to Section 4.4.4.5).

Section 6.1.1 introduces the syntax for the related model-elements, Section 6.2.1 provides the *View-Navigation* model and its UML profile, and Section 6.2.2 provides the *View* model and its UML profile.

#### 4.4.4.2. AppControllers Model and ControllerClass Model

Client-side event handling is an important general feature of the RiWAs (refer to Section 2.3.5.1) towards minimizing the response time, thus improving the user experience. High-level *AppControllers* element implement the client-side event handling in individual elements named by the RiWAsML as *ControllerClass* elements. Each *View* element may have a dedicated *ControllerClass*, which implements the event handlers and related functions of that particular *View*. Models are required for the detailed design of the *AppControllers* and *ControllerClass* elements. The features of these models are discussed below.

**AppControllers model:** This model is required to capture all the *ControllerClass* elements in a given high-level *AppControllers* element of a particular *Application* element of a RiWA, providing the following features.

- A *ControllerClass* is tightly coupled with its *View* and has no relationship with other *ControllerClass* elements.
- It is not mandatory for a *View* to have a *ControllerClass*; thus, the number of *ControllerClass* elements can be less than the number of *View* elements within a particular client-side *Application* element.

**ControllerClass model:** This model is required to depict the details of a particular *ControllerClass* element in an *AppControllers* model. The *ControllerClass* model should capture the following features.

- The event handlers for the *ControllerClass* element's view, the GUI element which triggers the event, and the event type should be captured. Some event handlers are required to show/hide GUI content, such as popups/toggles and hidden sections, which should be distinctly depicted.
- Reading data from its *View*, and which GUI elements need to be updated with the results should be included.
- A *ControllerClass* element's methods may implement some general view-related logic without using the *AppModel* element for processing. For example, reading the birthday from the view, calculating the age, and displaying it on the view without consulting the model.
- For business logic, *ControllerClass* elements must utilise the model in two ways: directly use the client-model or communicate with the server-model via the DC connectors.

Section 6.1.2 introduces the syntax for the related model-elements, and Section 6.2.3 provides the *AppControllers* model, *ControllerClass* model, and UML profile for them.

#### 4.4.4.3. AppModel Model and ModelClass Model

The MVC-based modularization is an essential feature for RiWAs (refer to Section 2.3.5.2), and models are required to capture the business logic details. In RiWAsML, *AppModel* elements implement the business logic, and the RiWAsML requires the following models to capture the low-level details of the *AppModel* elements.

**AppModel model:** this model is required to capture the *ModelClass* elements required to implement an *AppModel* model and their relationships within a given *Application* element. The *AppModel* model is similar to the UML class diagram.

**ModelClass model:** this model is required for the detailed design of a particular *ModelClass* element of an *AppModel* model, denoting the following features.

- Public methods, which are used to expose an *AppModel* element to the other elements.
- For server-models, the interactions with the databases.

These design models can directly assist in developing the *AppModel* elements aligning with the OODD practices.

Section 6.1.3 introduces the syntax for the related model-elements, and Section 6.2.4 provides the *AppModel* model, *ModelClass* model, and UML profile for them.

#### 4.4.4.4. DC-bus Model and EndpointsCollection Model

The RiWAsML requires the following models to design the connectors using the elements discussed in Section 4.4.3.

**DC-bus model:** this model is required to capture the *EndpointsCollection* elements within the server-side *Application* elements' connector of *DC-Bus* type and their relationships. The *EndpointsCollection* elements and their relationships should be captured based on the OODD practices.

**EndpointsCollection model:** this model is required for the detailed design of a particular *EndpointsCollection* element of a given *DC-bus* diagram.

Section 6.1.4 introduces the syntax for the related model-elements, and Section 6.2.5 provides the *DC-Bus* model, *EndpointsCollection* model, and UML profile for them.

#### 4.4.4.5. View-Controller Model

Since a view and its controller work closely to implement the client-side event handling, the *View-Controller* model is required to capture this relationship and related aspects. The *View-Controller* model is necessary to design the development-supportive details of a view's functions as a view-controller pair. This model would use the *View* model and *ControllerClass* model to denote their relationships. This model will help improve the user experience by identifying better ways to implement the view's functions and assist in developing the functions by ensuring all the required event handlers are included in the view's controller.

Section 6.2.6 provides the *View-Controller* model and its UML profile.

#### 4.4.4.6. View-Process Sequence Model

UML sequence diagram is a valuable tool for designing the execution flows of functions within a system, assisting in identifying the classes and their attributes and methods. The RiWAsML requires a model to feasibly integrate the UML sequence diagram aligning with the RiWAsML's other models and model-elements, offering the following features.

- The design should realise the *Application* elements and their internal elements – including *View* and its *ControllerClass*, *ModelClass* elements, and *EndpointsCollection* elements – which are participating in the process.
- The communication types between these elements should be reflected. For example, it should be shown as DC between the *ControllerClass* element and the *EndpointsCollection* element.

Section 6.2.7 provides the *View-Process Sequence* model and its UML profile.

## 4.5. Chapter Summary

This chapter sets the following attributes required to be satisfied by the RiWAsDM and the requirements for the model-elements and models for the RiWAsML.

- **RiWAsDM Attributes:** (**Attr 1**) Simplicity and (**Attr 2**) adoptability. The adoptability is expected under (**Attr 2.1**) comprehensiveness, (**Attr 2.2**) usability (learnability and readability /understandability), (**Attr 2.3**) development support, and (**Attr 2.4**) integrability.
- **Common model-elements:** (**R1**) naming label and (**R2**) communication channels with different styles of arrows.
- **High-level model-elements:** (**R3**) processing elements (*Application, Tier, Platform, AppControllers, AppModel, and Connector*), (**R4**) *Views* element, and (**R5**) additional elements (databases, files, web-services, ESB, users, and notes).
- (**R6**) **High-level models:** *Level 1 Applications* model, *Level 2 View-Process* model, and *L1+2 Architectural* model.
- **Low-level model-elements:** (**R7**) *View* (GUI elements, navigation, data, and related controller), (**R8**) *Component* (*AppControllers* and *AppModel*), (**R9**) *Connector* (DC-engine and DC-bus), and *EndpointsCollection* element.
- (**R10**) **Low-level models:** *View-Navigation* model and *View* model; *AppControllers* model and *ControllerClass* model; *AppModel* model and *ModelClass* model; *DC-bus* model and *EndpointsCollection* model; *View-Controller* model; and *View-Process Sequence* model.

Table 4.1 maps the quality attributes to the requirements to comprehend the relationship between them clearly.

Table 4.1 Mapping the quality attributes to the requirements

Requirements \ Attributes	R1 Element names	R2 Comm channels	R3 Processing elements	R4 Views (high-level)	R5 Additional elements	R6 High-level models	R7 Views (low-level)	R8 Components	R9 Connectors	R10 Low-level models
Attr 1 Simplicity		X	X	X	X		X	X	X	
Attr 2.1 Comprehensive		X	X	X	X	X	X	X	X	X
Attr 2.2 Usability	X					X				X
Attr 2.3 Dev support	X		X				X	X	X	X
Attr 2.4 Integrability	X					X				X

## Chapter 5. RiWAsML: High-level Modelling Language

---

This chapter introduces model-elements, models, and UML profiles to design high-level aspects of the RiWAs in the direction of satisfying the requirements set in Sections 4.2 and 4.3. The outputs of this chapter fulfil Step 2.1 of the RiWAsDM implementation process given in Figure 1.4 in Section 1.5.3, partially achieving the research objective 2.

### 5.1. Notations for General Model-elements of RiWAs

This section provides the notations for the element label and communication channels, which are common for high-level and low-level designing.

#### 5.1.1. R1 – Elements Label Notation

Section 4.2.1 sets the requirements for the element label.

**UML** meta-model does not provide a naming convention for the model-elements. In general, the element class/type is identified by the graphical symbol, and the element label is used to assign a custom name. For stereotypes, a pair of guillemets << >> are used above the element name to mention the stereotype name. The UML-based languages inherit this naming style. The engineers may use element names based on the project's scenario, domain, or development tools like frameworks.

The RiWAsML provides a unique element label format to satisfy the requirements set in Section 4.1.2, as discussed below.

**Label structure:** since the RiWAsML is a UML extension, the model-elements will be stereotypes of the UML meta-model. Therefore, the RiWAsML model-element label's text should be within a pair of guillemets << >>. The label content should include 3 segments: Element class, Type, and Name. The Guillemet helps wrap the label segments and visualize them as a single unit.

**Segments separator:** a space, comma, dash, and colon were considered to separate the label segments. A space would not provide a robust visual separation and may reduce the label's readability. A comma is a potential candidate; however, the comma is utilised for separating the platforms within the element's class and type segments (refer to Section 5.2.2). The dash is mainly used in computing to denote the subtraction operation; hence, it may express a different meaning. Colon is used in the natural language to join sections and has no strong and specific impression in computing. Considering these points, the colon has opted to separate the segments in the element label. The finalized element label format is shown below.

`<< Element : Type : Name >>`

Some guidelines to be followed as standards for element labels are stated below.

- Based on the ODD practices, for all the label segments, use the Pascal case for class-like elements and the Camel case for object-like elements (this will be indicated where necessary when introducing new RiWAsML elements). These values can be directly utilised in the development as names for namespaces/packages/classes.
- When the label is long, break the links after the colon.
- When a specific name segment value is not necessary, provide the type segment value for the name segment.

The RiWAsML model-elements will use this label notation, and the values for each new notation's element class and type segments will be specified. Section 5.3.1 provides the UML profile for RiWAsML's *Label* element.

### 5.1.2. R2 – Communication Channels Notations

Section 4.2.2 sets the requirements for the RiWAsML's communication channels.

- **UML** meta-model provides the *Relationship* abstract element, which is inherited to depict various relationships between the model-elements. The UML's *Deployment diagram* [146] uses a type of *association* relationship named *communication path* to denote the communication between the nodes using a regular line. A label can be used to indicate the type of communication which declines the readability of the design. Graphical bidirectional interface connectors are given for components whose readability is low compared to the arrows or lines.
- **TAM** [82] provides syntax named *Channles* to indicate the data flows of volatile information between agents (processing elements in TAM) using a line with a small circle in the middle, which is optionally directional. Unidirectional arrows are provided to indicate the access flows such as read, write, and modify. UML-like interface connectors are given for components.
- **Archimate** [121] uses a bidirectional dashed arrow notation named *Communication Path* for nodes to exchange data or material.
- **The C4 model** [79] uses dashed unidirectional arrows only to show the relationships between the elements.
- **UWE's** [147] *Navigation Model* uses association stereotypes with labels to show the links.
- **IFML** [124] uses a unidirectional arrow notation named *Navigation Flow* to show the “*sending and receiving of parameters in the HTTP request*” and a dashed arrow notation named *Data Flow* to denote data passing between *View Components*.

These notations of the available solutions are unable to denote different communication channel types, for instance, HTTP vs DC; they do not realise the request-response model in all the communication types, and they are not consistent across all the models of the language.

This thesis analyses the communication channels between the elements in RiWAs into 5 abstract types (refer to section 4.2.2). The RiWAsML addresses the direction and type aspects of the communication channels' syntax as below.

**Direction:** At the high-level design, bidirectional arrows can represent the request-response model. At the low-level design, unidirectional arrows can be used when the request and response are taking place between separate elements. The requesting and the responding elements should be understood based on the RiWAArch style, as discussed in Section 4.2.2.

**Type:** The RiWAsML offer the arrow styles in Figure 5.1 to realise the communication channel types specified in Section 4.2.2.3.

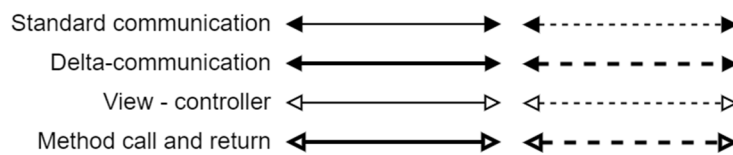


Figure 5.1 Proposed notation: communication channels

- **Standard communication:** a regular, black-headed arrow to denote standard communication like HTTP or other protocols like TCP/IP to communicate with databases or compatible external entities.
- **DC:** a thick black-headed arrow.
- **View-controller:** a regular white-headed arrow to denote the communication between the view and its controller.
- **Method calls and returns:** a thick white-headed arrow,
- **Sequence diagram returns:** the same arrow styles with dashed stems should be used to illustrate the returns of the calls. For example, for the return of DC, a black-headed arrow with a thick dashed stem should be used.

When the elements on the other end of the communication are unavailable on the same diagram, the RiWAsML offers numbered connectors, as shown in Figure 5.2. In that case, an arrow with the correct direction and type should be used between the element and the connector.

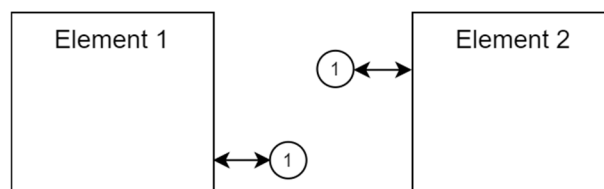


Figure 5.2 Proposed notation: communication channel with numbered connectors



Even though elements 1 and 2 are on the same diagrams, communication channels with numbered connectors may be used to reduce complexity and improve readability. The communication channel can be of any type, and the arrow can be either unidirectional or bidirectional, as required.

This thesis keeps push communication and related aspects out of the scope; however, a notation for push DC is proposed for future extensions, as shown in Figure 5.3. It is a thick black arrow similar to DC, with a doubled line stem.

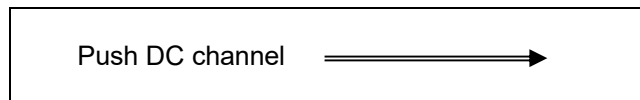


Figure 5.3 Proposed notation: push-DC

Further element-specific communication-related notations are given and discussed in relevant sections where necessary. Section 5.3.2 provides the UML profile for the RiWAsML's communication channel syntax.

## 5.2. Notations for High-level Model-elements of RiWAs

The following sections provide the RiWAsML notations for the high-level model-elements to satisfy the requirements set in Section 4.3.

### 5.2.1. R3 – Notation for Tier Element

Section 4.3.1.2 sets the requirements for the RiWAsML's *Tier* element. Some similar elements of available solutions are discussed below.

- **UML's** *Package* element can be exploited to denote the tiers [148]; however, it lacks the semantics.
- **Arc42's** [80] *Building Block View* allows showing tiers in the architecture design; anyhow, it suggests using lines and boxes instead of providing proper model-elements for different types of blocks.
- **TAM** [82] uses a dashed line to indicate the protocol boundaries and explains that "*Protocol boundaries usually partition a diagram in order to accentuate certain boundaries in communication.*" This notion of separation differs from the tier concept, and it will not indicate the role or distribution of the containing elements. Also, it cannot depict the use of multiple protocols between layers; for example, HTTP, XMLHTTP, and WS communication between the client and server cannot be shown.
- **Archimate's** [121] *Physical layer* provides some containers: *Equipment*, *Facility*, *Distribution Network*, and *Material* for different levels of separation; nevertheless, they do not provide a high level of separation similar to tier.

- **SysML** provides a more abstract concept called *Block*, which is likely to be used to denote tier [127]. The *Block* “defines a collection of features to describe a system or other element of interest” [127], and it can be exploited to model tiers; however, semantically, it’s a low-level element.

The RiWAsML specifies a rectangular block notation to indicate the tier. A RiWA comprises at least three tiers: the client, server, and data tiers. A tier block can be stacked either horizontally or vertically to denote multiple tiers. Vertical tiers may give an impression of a top-to-bottom parent-child-like relationship; horizontal tiers mainly provide an impression of left-to-right flow, which is more suitable with the client tier on the left as the workflows start with the client. Considering these points, the RiWAsML suggest horizontal tiers. The adjacent tiers may share the side borderlines, as shown in Figure 5.4.



Figure 5.4 Example: Tier elements

The tier label’s **element** segment should use the keyword “Tier”. Since the RiWAArch style is based on the 3-tier architecture, this thesis only specifies three values for the **type** segment of the label: *Presentation*, *Application*, and *Data/Storage*. The **name** segment may contain a suitable value to identify the tier based on the system’s requirements, as indicated in Figure 5.4.

### 5.2.2. R3 – Notation for Platform Element

The platform comprises three levels: the hardware, the OS, and the application-level virtualisation, as per the requirements set in Section 4.3.1.3.

- The **UML** meta-model uses the **node** element [137] to specify the platform details on the deployment diagram [149], which represents a deployment target of computational resources [119]. UML offers two levels of platform: device and execution environment, where the OS and application-level virtualizations are considered execution environments. The designer can specify the levels and denote the required details using multiple nested nodes, as shown in Figure 5.5.

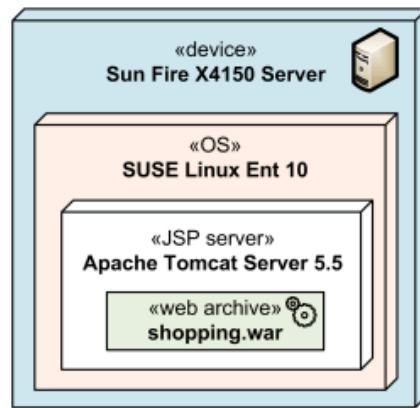


Figure 5.5 Example: using UML's nested nodes for platforms [146]

Using multiple nested nodes to model the platform details lacks consistency due to the unavailability of standards and reduces the readability by making the design untidy with many nested nodes. Refer to Section 8.1.3 to understand the issue with using the UML meta-model's *node* element and how it is addressed with RiWAsML's proposed platform syntax. The UML-based methods/tools use the UML meta-model's *node* to denote the platform, and they inherit the same issues with nested nodes, as discussed above.

- **Arc42's** [80] *Building Block View* enables denoting platform but uses boxes and lines without proper model-elements. Arc42's *Deployment View* uses the UML node without additional dedicated notations.
- **ArchiMate's** *Technology Layer* uses the node as a “computational or physical resource that hosts, manipulates, or interacts with other computational or physical resources” [150]. Archimate further provides some more notations – such as *System Software*, *Technology Function*, *Technology Service*, and *Technology Collaboration* – to include platform-related details in a model. The complexity, hence, the learning curve of Archimate, could be increased by having many notations for similar concepts.
- The **C4 model** [79] does not explicitly specify notations for platforms; however, platform details can be denoted in the level 2 *Container diagram* using boxes, which lacks semantics.
- **SysML's** [127] *Block Definition Diagram* provides notations like *AbstractDefinition*, which can be exploited to include the platform details into a model. However, semantically, they are low-level elements.

The RiWAsML proposes using the UML's *node* notation for the *Platform* element. Besides, the RiWAsML intends to reduce the use of nested nodes by exploiting the label to provide more details on a single node. The proposed *Platform* notation is given in Figure 5.6.

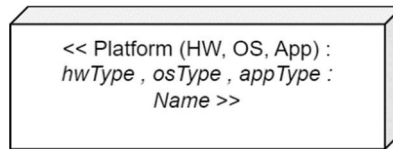
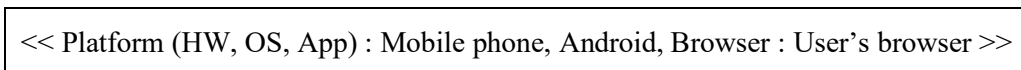


Figure 5.6 Proposed notation: Platform element

The following rules are provided to name the *Platform* element.

- The **element** segment of the label should be “Platform”.
- In addition, in the element segment, within brackets, the platform levels presented by the element should be indicated using the shortcodes **HW** for hardware, **OS** for operating systems, and **App** to denote the application-level virtualization. The levels should be separated using commas.
- For the **type** segment of the label, the technical details of the platform levels mentioned in the element segment should be specified in the same order, separated by commas.
- The **name** segment of the label should contain a name for the platform for identification purposes.

A browser on an Android mobile phone can be labelled as follows.



If explicit requirements exist to show different platform levels using distinct elements, the RiWAsML proposes the following rules to separate the platform levels into two nested *Platform* elements.

- In any device, the hardware and operating system are a tightly coupled pair of platforms; therefore, at the parent level *Platform* element, it is required to state the hardware and operating system, which specifies the device.
- A child *Platform* element can be used to represent the application-level platform details. In this case, the labels’ element and type segments should only indicate the necessary details.
- Moreover, when multiple application-level platforms are available within the same device, they can be denoted using children nodes within the same parent-level node. If the child platforms belong to multiple tiers, the parent platform can stretch across the tiers.

Figure 5.7 illustrates an example of nested platforms for each case described above.

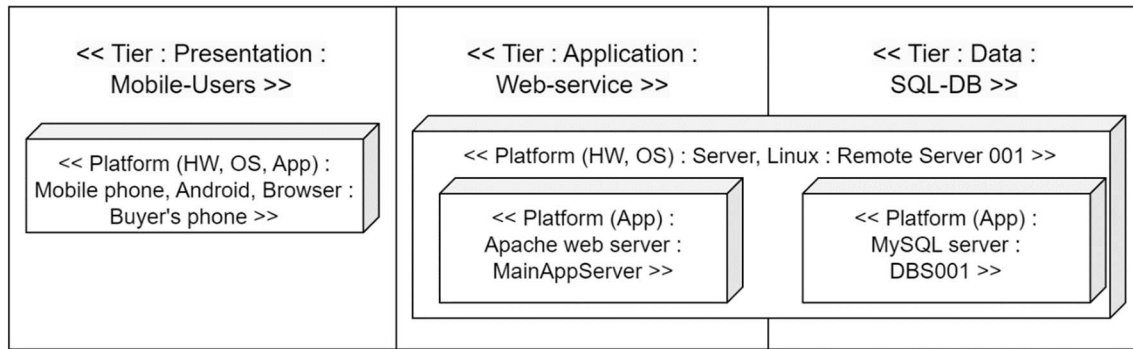


Figure 5.7 Example: nested platforms

In the presentation tier, the client uses a mobile device’s browser to access the system, and the mobile device is shown using a single *Platform* element. The hardware platform is the mobile phone, which uses the Android OS. The *Platform* element is named “Buyer’s phone” to express that this is the mobile phone device of the user type buyers. On the server side, both the web server and the DB server are installed on the same server device; in that case, the parent node is stretched across the *Application* and *Data* tiers. The children nodes are placed within the appropriate tier, stating only the application-level platform type and assigning names for each application-level platform.

When using cloud services to host *Application* elements and databases, where the hardware and OS details are not required, it is enough to provide a single *Platform* element with the cloud service details as the application-level platform.

### 5.2.3. R3 – Notation for Application Element

Section 4.3.1.1 sets the requirements for the *Application* element. None of the available solutions offer semantically similar elements to the *Application* element. Some comparable notations are discussed below.

- **UML** meta-model uses the *artifact* model-element [151], where an artifact can be a *script* or *executable file*. However, the *artifact*’s purpose is to represent some physical entity, including text document, source file, script, binary executable file, archive file, and database table, and it is conceptually different from the *Application* element.
- **Arc42** [80] can specify and denote an *Application* element using a box in its *Building Block View*, which lacks standard notation and, thus, semantics.
- **TAM’s** [82] *Component/Block Diagram* model-element named *common feature area* [45], which is likely to be exploited to represent applications by grouping components. Also, the *agent* element is an active element that can be exploited for an *Application* element. However, both *common feature area* and *agent* elements are semantically different from the *Application* element required by the RiWAsML.

- The model-element named *product* in **Archimate’s Business layer** [152] is closer to the concept of application; however, it characterizes a higher-level abstraction. The Archimate’s application layer’s *application component* is more of a UML component, which can be exploited to denote *Application* elements; yet, it’s semantically a low-level element.
- The primary purpose of the **C4 model’s** [79] *level 2 container diagram* is to show the applications and their associations. The container represents an application, yet since C4 uses boxes and lines, it lacks proper syntax and semantics.

The RiWAsML uses a rectangle with the label format << **Application** : *Type* : *Name* >> to model the concept of the *Application* element. An example is given in Figure 5.8, which shows a mobile application named “ShoppingApp”, which runs on an Android mobile phone’s browser.

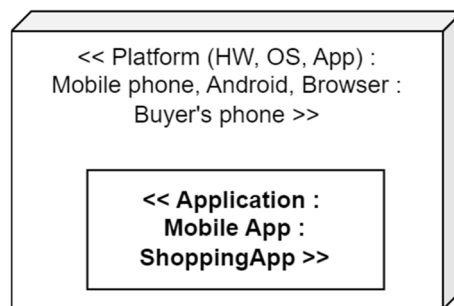


Figure 5.8 Example: Application element

The *Application* elements’ label’s **element** segment should be “Application”, the **type** should be a custom value indicating the actual type of the application, and the **name** should contain the suitable name, which can be exploited as a namespace or package name in development.

#### 5.2.4. R3 – Notations for Components

The RiWAsML required two types of high-level components: controllers and model, according to the requirements set in Section 4.3.1.4. Available solutions offer the following elements, which attempt to realise the controllers and model components.

- **UML** meta-model includes a model-element for components [153] with a notion similar to the architectural component, explained as “*a self-contained unit that encapsulates the state and behaviour of a number of Classifiers*” [119], and the UML-based methods/tools inherit it. UML component does not explicitly realise controllers or models semantically.
- Some documents on the web discuss a design pattern named **Entity-Control-Boundary (ECB)** pattern [154], originating from Ivar Jacobson [155], which is similar to the MVC where the Entity is like the Model, and the Boundary is like the View. These documents on the web provide some UML notations for the ECB elements [154], and some UML tools even support using them on diagrams [156]. According to the UML 2.5 specification, the UML meta-model uses a stereotype <<Entity>>, which applies to the component to denote a “*persistent information*”

*component representing a business concept*” [119], which is semantically the data aspects of a model. Other than that, the ECB-like concept was not found in UML.

- The **C4 model**’s component diagram uses the components to denote the processing elements within containers. However, these components do not have formal syntax and semantics.

The RiWAsML stays with the UML component notation instead of using different notations for the model and controller. The component’s label’s type segment is exploited to indicate if the component is a high-level controller or model.

Figure 5.9 illustrates the proposed notation for the component.

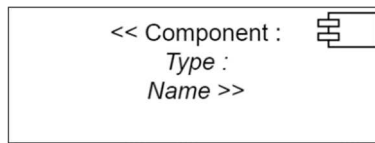


Figure 5.9 Proposed notation: component element

The label’s **element** segment should be “Component”; the **type** segment is proposed to denote the component type as follows.

- **AppControllers**: for the *AppControllers* elements, the label’s type segment should be “Controllers”, which is always plural.
- **AppModel**: there can be only one model for the MVC triad; however, for RiWAs, the RiWAArch style [12] splits the model between the client and the server based on the BAW-MVC [105]. In this setting, the RiWAsML suggests using the types “*ClientModel*” and “*ServerModel*” for the *AppModel* component elements.

A suitable **name** can be assigned to the components for identification purposes, which may be related to the application name. The component’s name can be used as a namespace or package name in the development. Examples of the components are shown in Figure 5.10.

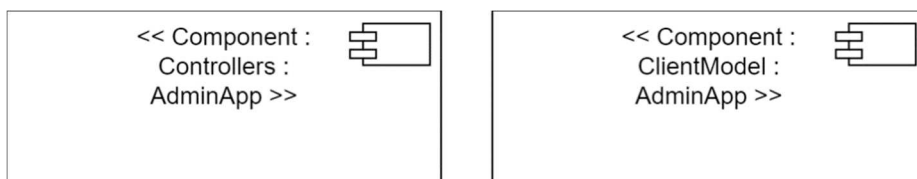


Figure 5.10 Example: high-level Controllers and Model components

RiWAsML considers the communication between the components and connectors in RiWAs based on the RiWAArch style and uses the communication flow arrows discussed in Section 5.1.2. Therefore, the UML meta-model’s component interfaces are not utilised. The communication flows between the components are discussed in necessary sections where required.

### 5.2.5. R3 – Notation for Connectors

This section provides the high-level notation for *Connector* elements satisfying the requirements set in Section 4.3.1.5.

- **UML** already has a model-element called *Connector* [103], which relates to the interfaces. The semantics of the UML’s connector discuss low-level communication between the components via the interfaces; therefore, it is unsuitable to express high-level connector elements, which wrap low-level details. Other UML-based methods/tools also use a similar concept to include communication details in a design at the interface level; thus, they are not reviewed here.

The RiWAArch style has a dedicated high-level connector pair for DC, DC-engine and DC-bus; thus, a high-level model-element is required to model the connectors in RiWAs. Since the connector is also considered a processing element (refer to section 4.3.1), the RiWAsML suggests using the same component notation given in Section 5.2.4 with a *Connector* stereotype. An example of the proposed notation is shown in Figure 5.11.

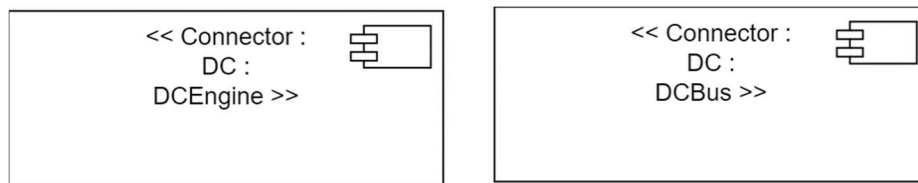


Figure 5.11 Example: DC-engine and DC-bus connectors

The **element** segment of the connector’s label should be “Connector”. The main types of communication in RiWAs are standard HTTP and DC, which can be stated in the connector label’s **type** segment. The type segment may explicitly specify *DCEngine* or *DCBus*. The RiWAArch style does not include HTTP connectors; anyhow, it is possible to have HTTP connectors in the server-side *Application* elements to handle the HTTP requests from the views or the controllers. In the case of DC, the **name** segment of the label can be set as *DCEngine*, *DCBus*, or any other suitable name for identification of the connectors. For example, the name of the DC-engine *Connector* element in a mobile application could be named as *MobileDCEngine*.

### 5.2.6. R4 – Notation for Views

Section 4.3.2 sets the requirements for the RiWAs’ high-level *Views* element. The positions of the available solutions are stated below.

- **UML** meta-model does not address presentation-related concerns and, hence, does not include model-elements for views and related aspects. Many other UML-based methods/tools try to fill this void by introducing new model-elements and models for views; however, they are primarily low-level design solutions and are discussed in Chapter 6 under RiWAsML’s low-level modelling.



- **TAM** includes a rectangular notation for views in some sample diagrams [82]; nevertheless, discussions were not found in TAM’s documentation.

The RiWAsML proposes to use a rectangle notation for the high-level views element, as illustrated in Figure 5.12.

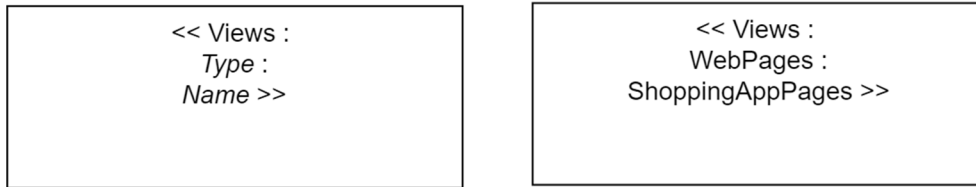


Figure 5.12 Proposed notation: Views element (on the left) and an example of the use (on the right)

The *Views* element label’s **element** segment should be “Views”; note that it’s in plural form. The **type** segment should indicate the type of views, as in “*WebPages*” for browser-based apps, “*Activities*” for mobile apps, and “*Windows*” for desktop apps. The **name** segment should use a proper name related to the *Application* element for identification purposes.

Modelling aspects of an individual view within this high-level *Views* element are discussed in Chapter 6 under low-level design.

### 5.2.7. R5 – Notation for Additional High-level Elements

Section 4.3.3 discusses the requirements for the additional high-level elements for the RiWAsML. Available solutions provide many additional elements to support high-level and low-level designing, as given below.

- **UML** – Users/actors, notes, artefacts, database systems, and schema.
- **Arc**<sup>42</sup> – Users/actors and notes.
- **TAM** – Users/actors and databases.
- **ArchiMate** – Users/actors, roles, objects, databases, and files.
- **C4 model** – Persons, devices, databases, and notes.

The RiWAsML offer the notations in Figure 5.13 for the model-elements specified in section 4.3.3, which help model high-level aspects of the RiWAs.

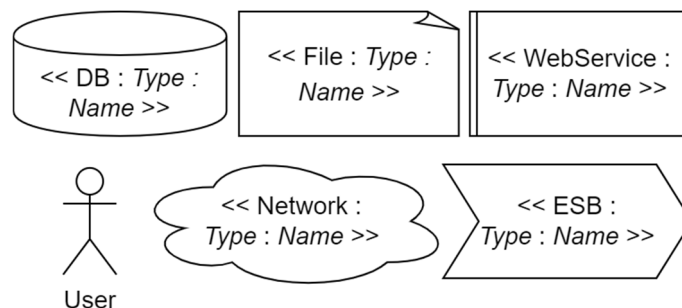


Figure 5.13 Proposed notation: additional high-level elements

The **element** segment of these support elements may use the values specified in Figure 5.13, the **type** segment may contain suitable values, and the **name** segment may have proper names to identify the elements.

The RiWAsML provides a *Notes* element with the notation depicted in Figure 5.14. When using a *Notes* element, it should point to the element which it describes. The *Notes* element does not use a label and contains the details of the element as text which it describes.

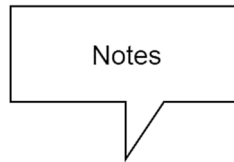


Figure 5.14 Proposed notation: *Notes* element

### 5.3. R6 – High-level Design Models and UML Profiles for the RiWAsML

This section provides the UML profiles for the RiWAsML's general model-elements discussed in Section 5.1 and high-level models set in Section 4.3.4, which use the notations proposed in Sections 5.1 and 5.2.

UML meta-model elements, which are mostly considered for the RiWAsML extensions, are specified below.

- **Namespace** is an abstract *named element* that can own (contain) other named elements and can be seen as a container for named elements [157].
- **Package** is a *namespace* used to group together elements that are semantically related and might change together [158]. A package can own the packageable elements such as *Type*, *Classifier*, *Class*, *Component*, and *Package* [158].
- **Classifier** is an abstract metaclass that describes (classifies) a set of instances with common features, and the *classifier* inherits from both the *namespace* and *redefinable element* [159].
- A **class** is a *classifier* which describes a set of objects that share the same features, constraints, and semantics (meaning) [160].

RiWAsML proposes two-level hierarchical models for the high-level designing of RiWAs (refer to Section 4.3.4), similar to Arc42's [80] *Building Block View* and C4 model [79]. It will help design large and complex RiWAs in two-level smaller diagrams for improved usability. However, in the case of small RiWAs, it is viable to amalgamate the two levels into the same diagram. These aspects are detailed in the Sections 5.3.3, 5.3.4, and 5.3.5.

### 5.3.1. Profile for Label Element

Section 5.1.1 offers the RiWAsML's syntax for the *Label* element.

UML meta-model uses label notation to indicate the element name for identification purposes. UML core includes a *UMLLabel* class, which inherits the *UMLShape* class [119]. The *UMLLabel* class is inherited by 4 sub-classes: *UMLKeywordLabel*, *UMLNameLabel*, *UMLTypedElementLabel*, and *UMLRedefinesLabel*, as shown in Figure 5.15.

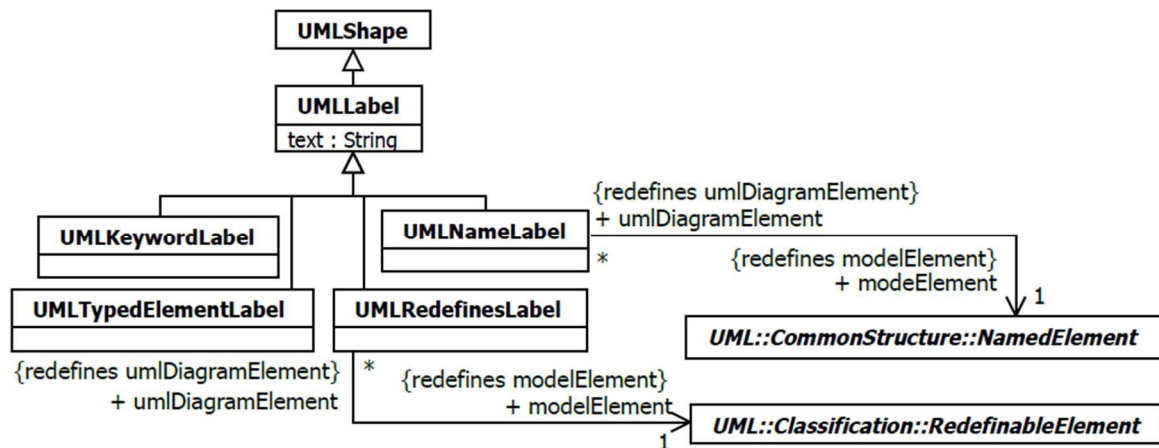


Figure 5.15 UML meta-model's labels [119]

*UMLNameLabel* is used to name the model-elements with text, which specializes the *UMLLabel* class. Usually, a label contains a single string segment. Since the RiWAsML requires a three-segmented element naming format (refer to section 4.2.1), the *UMLNameLabel* is extended to a new class named *ElementNameLabel*, which specifies the label format given in section 5.1.1. *ElementNameLabel* is a composition of *LabelSegmentElement*, *LabelSegmentType*, and *LabelSegmentName*, which are allocated to the element label's class, type, and name segments of the RiWAsML's *Label* element. The UML profile for the *Label* element is given in Figure 5.16.

RiWAsML specifies values for the label's **element** segment; therefore, the *Label* element's UML profile uses an enumerator with the stipulated values for the element segment, which includes the values of all RiWAsML's high-level and low-level elements. Even though some values are specified for the label's **type** segment, the *LabelSegmentType* stereotype is constructed with a string tag definition *labelSegmentType*, allowing the designers to use custom values according to the requirements. The label's **name** segment can use custom names; hence, it is constructed using a *UMLNameLabel* stereotype with a string tag definition *labelSegmentName*.

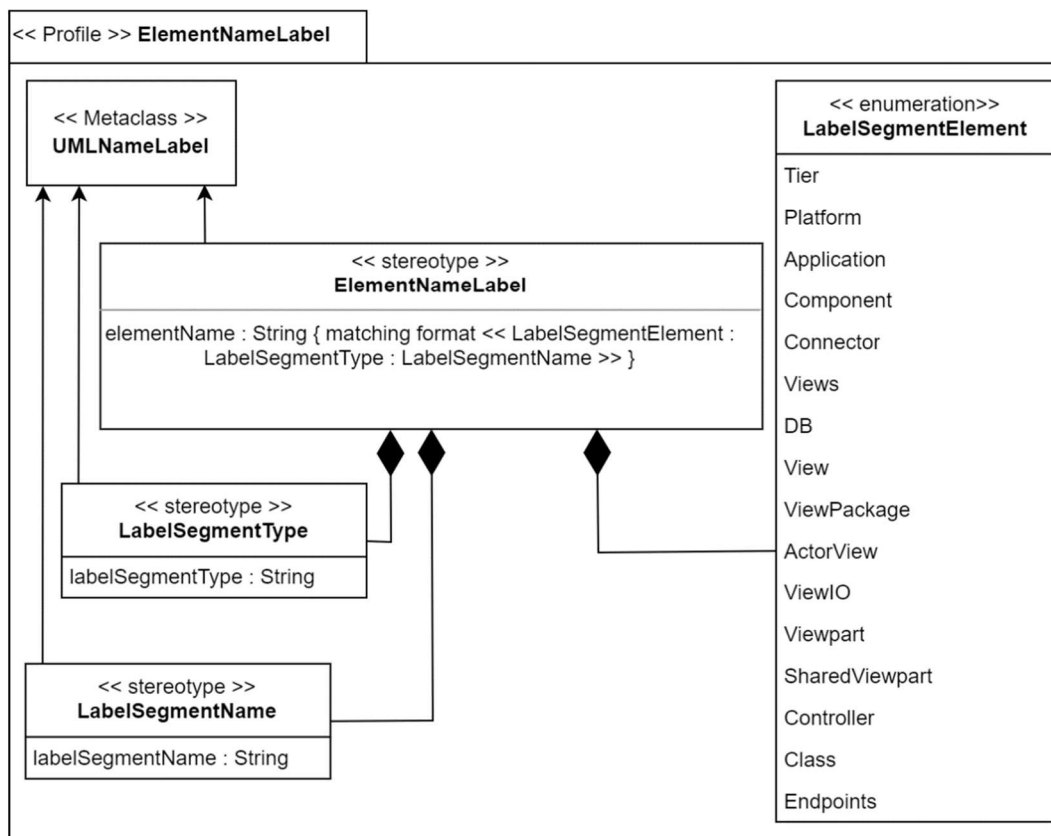


Figure 5.16 UML profile: RiWAsML Label element

### 5.3.2. Profile for Communication Channels

Section 5.1.2 provides the syntax for the RiWAsML's communication channels. The following UML elements are the candidates for the RiWAsML's communication channel profile.

- UML uses a communication channel named *InformationFlow* as “some kind of ‘information channel’ for unidirectional transmission of information from sources to targets” [161]. The notation uses a dashed unidirectional arrow, inheriting from *DirectedRelationship*.
- UML's *Object flow edges* use straight-lined unidirectional unfilled arrows for data flows of objects [162].
- UML *Messeges* offer a set of unidirectional arrows for *Synchronous call*, *asynchronous call*, *asynchronous signal*, *Create*, *Delete*, and *Reply* in the sequence diagram [163]. The *message* is a specific use of message flow and is not suitable for RiWAsML's communication channels.
- UML *Communication path* “is an association between two deployment targets, through which they are able to exchange signals and messages” [164].

RiWAsML's communication channels opt to extend UML *association*, similar to the UML *communication path*. The UML profile for RiWAsML communication channels is given in Figure 5.17. This profile extends the UML's *association* metaclass for the 4 types of RiWAsML's communication channels (standard, DC, view-controller, and method calls and return) for the

bidirectional arrows, unidirectional arrows, and unidirectional arrows with dashed stems for the *View-Process Sequence model* (refer to Section 4.4.4.6).

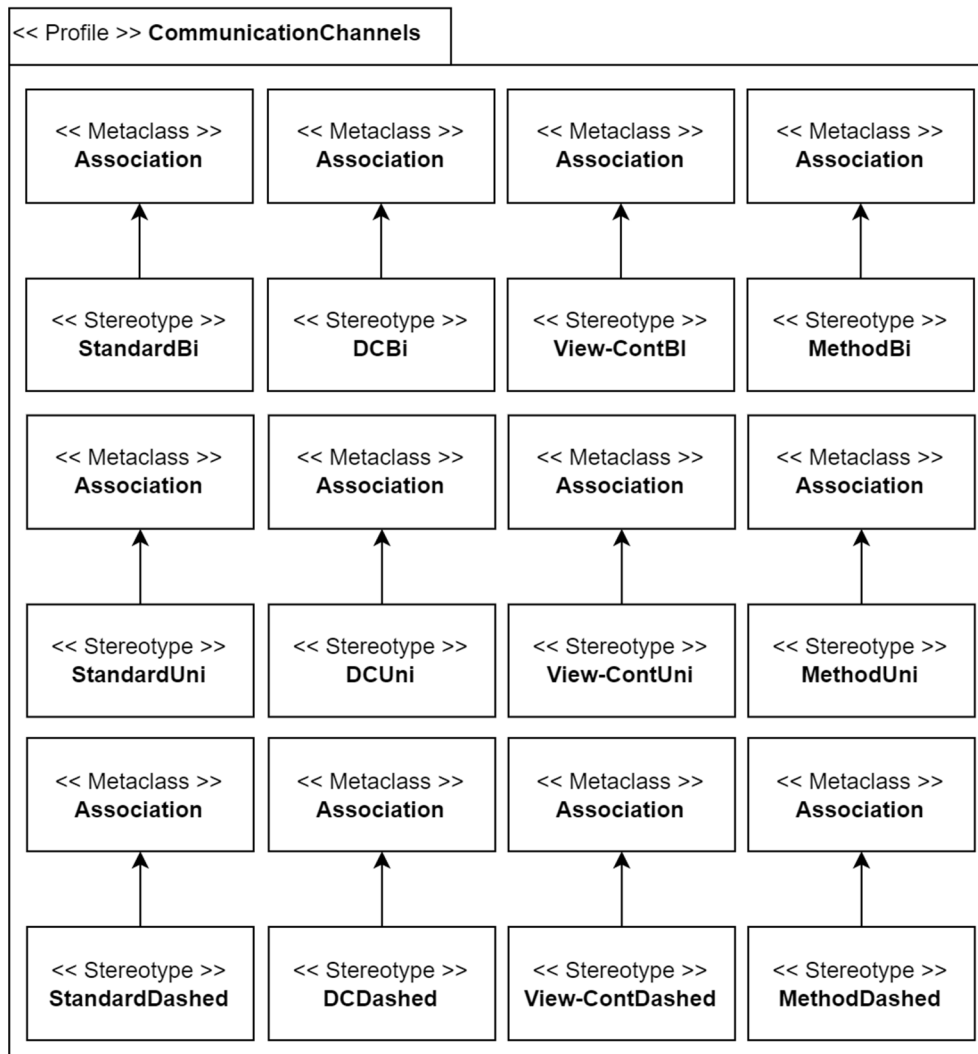


Figure 5.17 UML profile: RiWAsML communication channels

### 5.3.3. Level 1 Applications Model

The requirements for the RiWAsML *Level 1 Applications* model are set in Section 4.3.4.1. Some similar models in available solutions are as follows.

- **UML** does not provide any models for high-level architectural design. The UML's **package diagram** can be exploited to design high-level layered designs [148]. However, it does not have sufficient semantics for comprehensive high-level modelling.
- **UML** meta-model uses the **deployment diagram** to design the deployment details using platforms, and UML-based methods mainly utilise the deployment diagram to denote the platform and related information. The deployment diagram does not explain the grouping of platforms into tiers and may use multiple nested nodes to indicate the complete platform details. The deployment diagram can also include more information like deployable artefacts,

specifications, and schemas [149], which may provide additional information related to the core elements. The deployment diagram does not realise tiers and applications.

- **TAM's** [82] **Component/Block diagram** uses a tier-like separation primarily based on the communication protocol and does not include platform details or proper elements to denote the RiWAs *Application* elements.
- **Arc42's** [80] *Building Block View* tries to capture the architectural elements but lacks proper notations and definitions for its levels.
- The **C4 model's** *Container diagram* provides guidelines to capture the *Application* elements; however, it lacks readability without appropriate notations and rules. Also, the tier and platform details cannot be depicted in the C4 model.

None of the available solutions provides models to satisfy the requirements of the RiWAsML's *Level 1 Applications* model set in Section 4.3.4.1. Figure 5.18 illustrates an example *L1 Applications* diagram for a RiWA with a database.

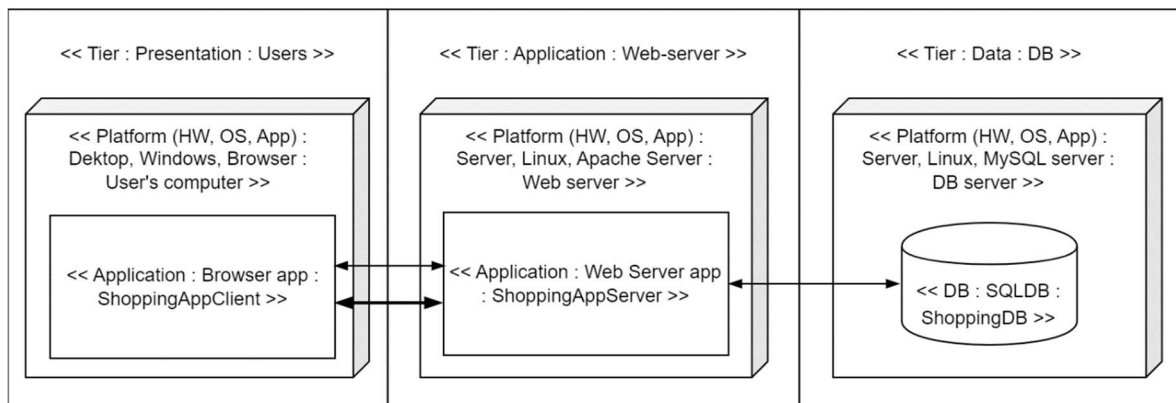


Figure 5.18 Example: *L1 Applications* diagram

This sample *L1 Applications* diagram satisfies the requirements for the *L1 Applications* model as follows.

- All the *Application* elements needed for the RiWA are realised.
- The *Platform* and *Tier* elements for the *Application* elements are realised.
- The database and its platform and tier are realised.
- The communication between the *Application* elements is realised. There are two communication channels between the *ShoppingAppClient* and the *ShoppingAppServer*: the upper channel is for HTTP communication, and the lower channel is for the DC. The *ShoppingAppServer* communicates with the database using the standard communication channel.

The UML profile for the *L1 Applications* model is given in Figure 5.19.

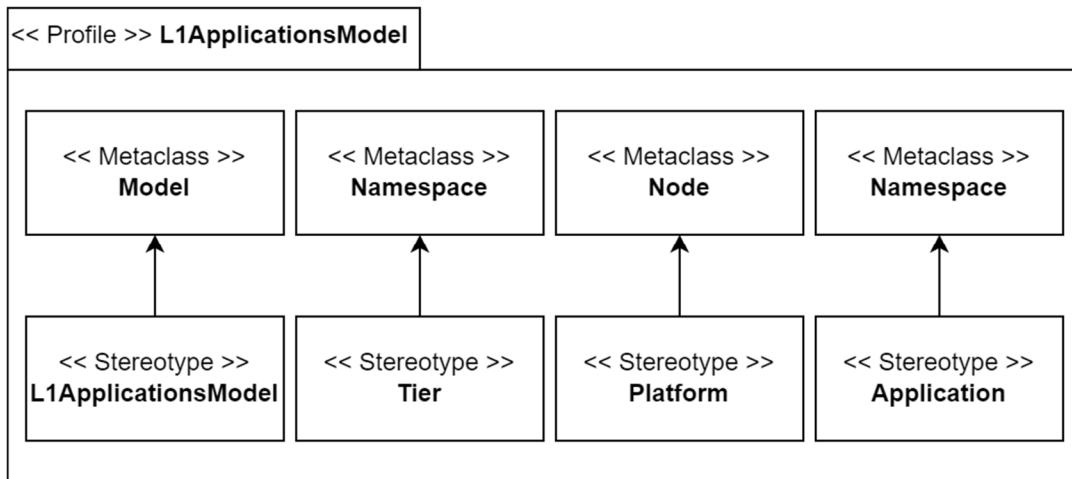


Figure 5.19 UML profile: applications model and its elements

The *L1ApplicationsModel* profile comprises the stereotypes *ApplicationsModel*, *Tier*, *Platform*, and *Application*, which extends the UML metaclasses as discussed below.

- **Tier:** Since the UML meta-model has no particular diagrams for high-level design, there is no notation to denote tier. UML suggests exploiting *packages* for designing layered architectures, even for web applications [148]. A layer in the layered architecture is not necessarily a *tier*, and a *layer* can even be an abstract concept like a view/controller/model for a standalone desktop application; thus, using the UML *package* to specify the tier concept is not straightforward. Due to the semantical difference between the RiWAsML's tier and the UML's package, extending the UML's package to derive the RiWAsML's tier is not recommended. Stipulating the tier as a high-level element with internal elements which can communicate, RiWAsML extends the UML *namespace* to derive the *Tier* stereotype.
- **Platform:** Since the UML's node can realise *device* and *execution environment* elements, RiWAsML extends the UML's node for the *Platform* element.
- **Application:** At the high level, the *Application* element acts as a package, which groups the views, controllers, and model. Generally, the purpose of a *package* is only to group the containing elements to show the ownership, and the communication between the *packages* or owning elements is not denoted. Therefore, the *package* is not considered a potential metaclass to extend for the *Application* element. At the lower level, the *Application* elements do not share common features since client-side applications like mobile apps and server-side applications like web services may have different features; hence, the *Application* element cannot be considered a *class* or a *classifier*. Considering these facts, RiWAsML opts for the UML's abstract *namespace* element to extend for the *Application* element.

A real-world use case of the *L1 Applications* model is discussed in Section 8.1.3.

### 5.3.4. Level 2 View-Process Model

This section discusses the RiWAsML's *L2 view-process* model in the direction of satisfying the requirements set in Section 4.3.4.2. The similar models provided by the available solutions are stated below.

- **UML Component diagram** [165] is used to design the configuration between a set of components and sub-components. However, these components are not specified as controllers, models, or connectors, and the views are not realised.
- **UML's Composite Structure diagram** can capture the internal structures of a classifier [166], which is similar to the component diagram and cannot realise views, controllers, models, and connectors.
- **Arc42's Building block view** may capture component-level formalism in levels 1 and 2. However, without a formal model and model-elements, it's not much usable.
- The **C4 model's Component diagram** can generally capture components without realising the controllers and model. Also, the views and connectors are not realised.

Example diagrams of the *L2 View-Process* model are given in Figure 5.20. These diagrams are for the *Application* elements: *Browser app* and *Web server app* in Figure 5.18. Note that two different *L2 View-Process* diagrams are included in the same figure for ease of demonstration and discussion.

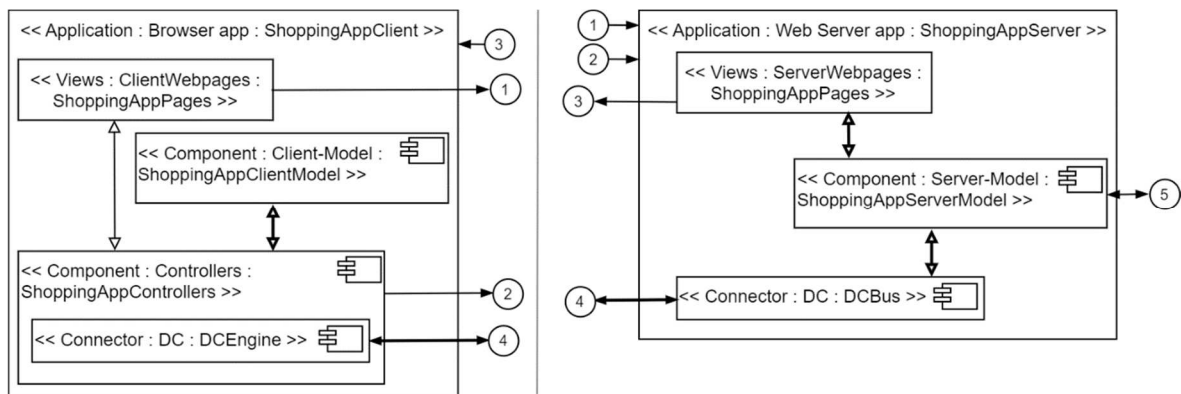


Figure 5.20 Example: View-Process diagrams for the Browser app (left) and Web server app (right)

These diagrams satisfy the requirements set in Section 4.3.4.2 as follows.

- The views, controllers, client-model, and DC-engine elements and their communication are depicted in the client-side *Application* element.
- The views, server-model, DC-bus, and their communication are denoted in the server-side *Application* element.
- Communication channels with numbered flow connectors indicate the standard HTTP communication between the client-side and server-side *Application* elements, which are based on the RiWAArch style. The client's *Views* and *AppControllers* elements can send HTTP



requests to the server-side *Application* element, and the client-side *Application* element receives the response sent by the server's *Views* element, which is a webpage and its resources.

- *DCEngine* and *DCBus* elements use bidirectional DC communication channels with numbered flow connectors.
- A bidirectional communication channel with a numbered flow connector denotes the communication between the *Server-Model* and the database.

Note that the *Views* elements are separated by stating the *ClientWebpages* and *ServerWebpages* on the type segment.

The UML profile for the *L2 View-Process* model is given in Figure 5.21.

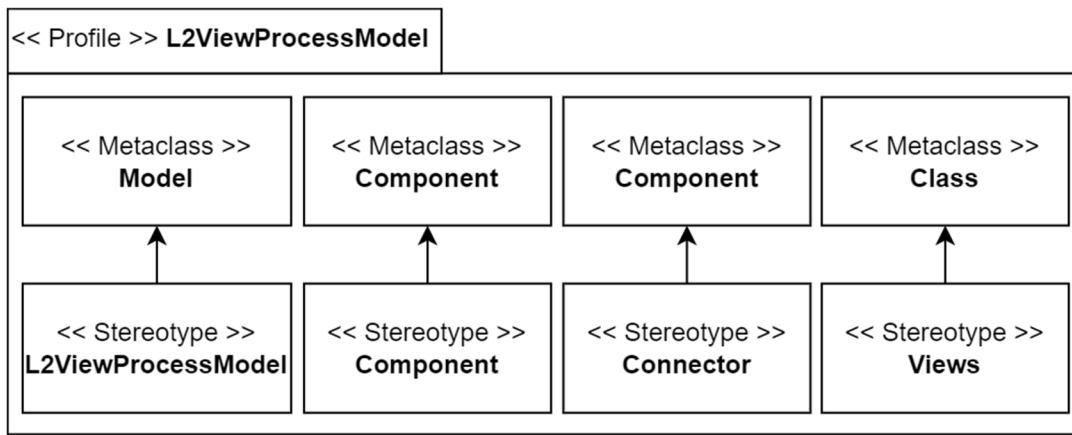


Figure 5.21 UML profile: view-process model and its elements

The UML *Component* is extended for RiWAsML *Component* and *Connector* elements. The UML *Class* is extended for the *Views* element, considering that all the views are composed of a set of abstract elements.

### 5.3.5. Level 1+2 Architectural Model

The *Level 1+2 Architectural* model is introduced here, satisfying the requirements set in Section 4.3.4.3. For small RiWAs, the RiWAsML allows combining the *L1 Applications* model and *L2 View-Process* model into a single model, named the *L1+2 Architectural* model, to support capturing all the high-level aspects into one diagram.

The TAM's [82] *Component/Block diagram* is the only similar model identified in the literature, which includes many details required for the RiWAs. However, it does not contain sufficient elements to realise the MVC and DC aspects.

An example of the RiWAsML's *L1+2 Architectural* model is given in Figure 5.22. A larger version of the same diagram is provided in Appendix A.

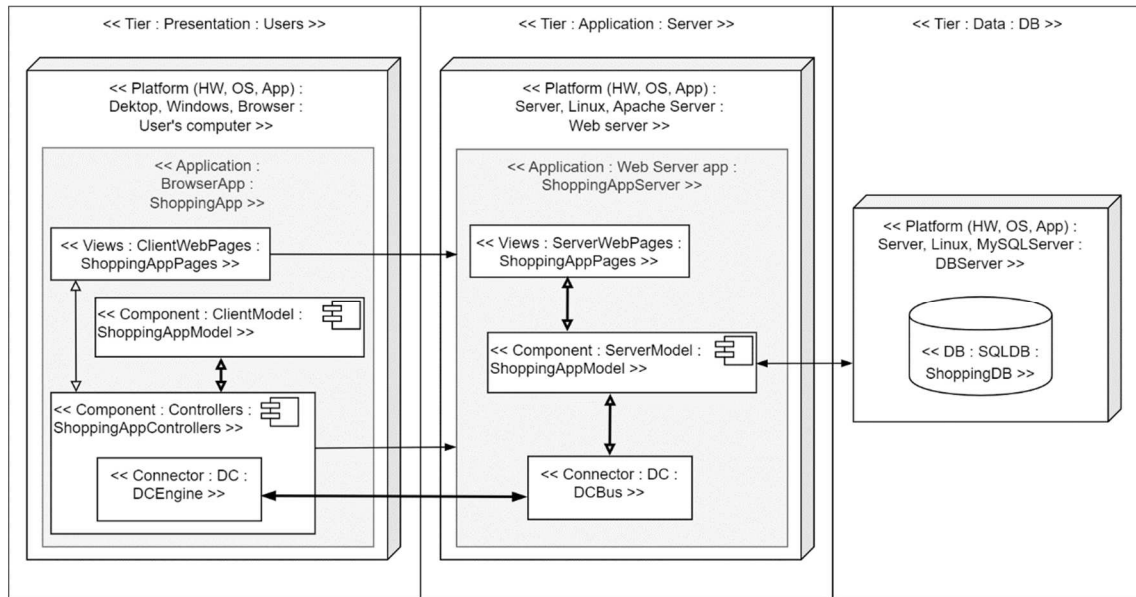


Figure 5.22 Example: shopping app – L1+2 Architectural diagram

The arrows from the server’s *Views* element to the client’s *Views* element, as depicted in the RiWAArch style, are not included in the *L1+2 Architectural diagram*. However, it should be understood that for the browser-based RiWAs, at the runtime, a single view is loaded to the client-side *Application* element from the server’s views collection. The RiWAs, with mobile and desktop clients (without a browser-based client), do not require *Views* elements on the server-side, and also, the HTTP requests from the client elements to the server are not necessary.

Figure 5.23 provides the UML profile for the *L1+2 Architectural* model. Since other model-elements are included in the profiles of the *L1 Applications* model and *L2 View-Process* model, the *L1+2 Architectural model*’s profile only consists of the *ArchitectureModel* stereotype.

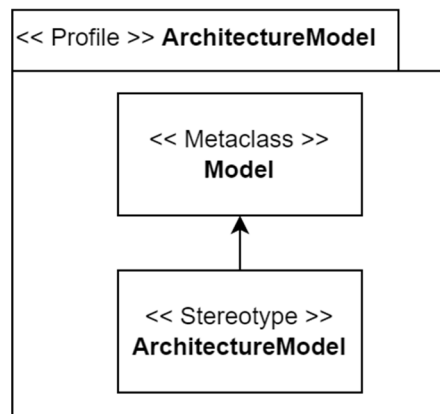


Figure 5.23 UML profile: L1+2 Architecture model

## 5.4. Chapter Summary

This chapter introduces the following model-elements, models, and UML profiles of the RiWAsML, for the high-level modelling of RiWAs.

- **General model elements:** *Label* element and *Communication channels* (Standard communication, DC, View-Controller communication, Method call and return).
- **High-level model elements:** *Tier*, *Platform*, *Applications*, *Component* (Controllers, Client-model, and Server-Model), *Connector* (DC-engine and DC-bus), *Views*, and additional high-level elements (Database, File, WebService, User, Network, ESB, and Notes).
- **High-level models and their UML profiles:** *L1 Applications* model, *L2 View-Process* model, and *L1+2 Architectural* model.

## Chapter 6. RiWAsML: Low-level Modelling Language

---

This chapter introduces model-elements and models for designing low-level aspects of the RiWAs in the direction of satisfying the requirements set in Section 4.4. Also, UML profiles, which are required for the new models and model-elements, are produced. The outputs of this chapter fulfil Step 2.2 of the RiWAsDM implementation process given in Figure 1.4 in Section 1.5.3. The results of this chapter, together with the results of Chapter 5, achieve research objective 2.

### 6.1. Notations for Low-level Model-elements of RiWAs

This section introduces the RiWAsML's low-level model-elements to satisfy the requirements set in Section 4.4. These model-elements use the same labelling format proposed in Sections 4.2.1 and 5.1.1. At the low-level design phase, the design models are closer to the development aspects like classes, attributes and methods. Hence, support for development is focused on when proposing the model-element notations and models, considering how easily the design can be mapped to code.

#### 6.1.1. R7 – Notations for Low-level Views and Related Elements

Many GUI designing tools are available to support both the content and aesthetic details [114], some of which provide advanced features like prototyping and code generation. However, these tools do not offer modelling notations or methods. The RiWAsML focuses on the view details, which assist the development of the functionalities, eliminating the aesthetic aspects, which explain how the views will actually look. According to the literature review performed in this thesis, UWE, UWE-R, IFML, SysML and the research publications concern view/GUI modelling aspects (refer to Sections 3.3 and 3.4 for their reviews), as investigated below.

- **UWE** [23] addresses some presentation and navigation concerns of web applications, and **UWE-R** [123] extends it for RIAs/RiWAs by supporting the design aspects of RiWAs' rich GUIs by introducing some GUI elements, whose scope is minimal compared to the modern RiWAs.
- **IFML**'s [124] main focus is on the presentation and interactive flows; hence, it provides some model-element groups, *View Containers* and *View Components* and an element named *Activationy Expression* for boolean expression-based GUI elements handling. Even though IFML offers some useful abstract GUI elements, the navigation and related controller aspects are not looked into.
- **SysML** [127] uses a stereotype named *View*, which extends the UML's class element to include view-related details in the designs, which concerns more about the data-related aspects rather than visual aspects.

- **RUX-model** [22] has introduced GUI-related model-elements for RIAs, mainly focusing on low-level aspects like special, temporal, and interaction presentations. The RUX model is more of a process and does not offer model-elements.
- **OOH4RIA** [35] focuses more on presentation and navigation-related modelling and introduces diagrams and notations for them. It tries to specify a lot of GUI elements, which is not an effective technique (refer to Section 4.4.1.1). It does not try to capture implementing functions and related aspects like the controllers.
- **IAML** [42] attempts to address view-related concerns by identifying visible elements, events on them, and navigational aspects. IAML explicitly specifies a set of GUI elements and events, which is a less effective technique (refer to Section 4.4.1.1).

The RiWAsML looks into offering abstract GUI element classes in the direction of assisting the implementation of the functionalities and views' navigation, data, and related controller aspects to satisfy the requirements set in Section 4.4.1. The following sub-sections discuss these aspects, producing the required model-elements.

#### 6.1.1.1. GUIs

The highest level abstraction of a GUI is a view, which can be of different types, for instance, a webpage of a browser-based RiWA, an activity of a mobile-based RiWA, or a desktop window of a window-based RiWA. Section 4.4.1.1 sets the requirements for the RiWAs GUI elements. Similar elements in available solutions are discussed below.

- **UWE's presentation model** [142] provides a class-like notation with an icon to denote a view as a *presentationPage* or *presentationAlternatives* elements using a label to name the element. These elements are only for designing webpages, and other view types are not supported.
- **IFML** [124] uses a class-like element called *View Container* to show a webpage, window, or pane, which may use a stereotype to indicate the type and a label to name the element.

The RiWAsML gives a rectangle with its label, as shown in Figure 6.1.



Figure 6.1 Example: View – a web page

The label's **element** segment should be “View”, the **type** segment can be “Webpage”, “Activity” (for mobile apps), or “Window” (for desktop apps), and the **name** segment may contain the custom name of the view. The body of the View can have the internal elements required to implement the functions of the View. The RiWAsML introduces the following abstract element classes to design the interior elements of views.

### Input/Output Elements

The View's input elements are used to enter the user's data and commands, and the output elements display results to the user. The available solutions commonly provide an explicitly defined set of input/output elements. This RiWAsML proposes a technique that allows engineers to define the input/output elements in a way that aligns with their development technologies. This technique uses a rectangle notation, which exploits the RiWAsML's label to define the input/output elements. An example is given in Figure 6.2.

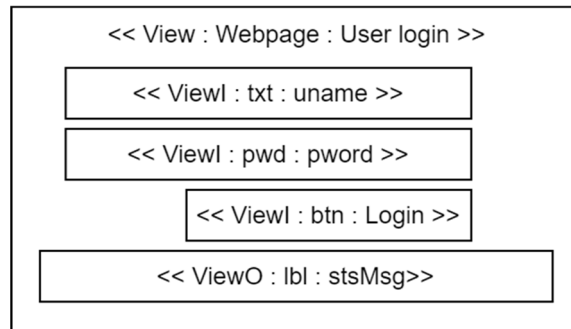


Figure 6.2 Example: View – GUI Input/Output elements

For the **element** segment of the label, 3 classes are provided: *ViewI* for input elements, *ViewO* for output elements, and *ViewIO* for the elements, which act as both input and output elements. For example, consider a list which shows a list of items; in that case, the list and the list items can be regarded as output elements. A user can click an item on the list and see details on a popup; in this situation, the list item acts as an input element by allowing the user to click, which is a command input. Further, additional functionalities can be implemented to enhance user experience, which makes the input elements produce output and vice versa. For example, an email field may verify the correct email format as the user types in the email and indicates the email text in red if it's wrong or in green if it's accurate. If these extra features were decided at the design time and included in the design, they would assist development. Therefore, it's vital to make decisions related to user experience and incorporate them in the design by precisely mentioning the nature of the required GUI elements.

For the **type** segment of the label, the RiWAsML recommends using a short code indicating the element type. A suggested list of shortcodes for the common GUI elements is given in Table 6.1, inspired by the components in Visual Studio IDE.

Table 6.1 A suggested list of shortcodes for GUI elements

Button - btn	List box - lstb	Password box - psd
Calendar - clnd	Menu - mnu	Progress bar - pabr
Check box - chk	Print dialog - prntdia	Radio button - rdo
Checklist box - chlb	File dialog - fildia	Scroll bar - scrbar
Colour dialog - clrdia	Font dialog - fontd	Tab - tab
Drop down list - ddl	Label - lbl	Text box - txt

The **name** segment should be utilised to indicate a meaningful name for the GUI element. The shortcode used for the type segment of the label and the name together would be used to make up the GUI element's development name. For example, consider the view in Figure 6.2; the username textbox's development name could be txtUname, and the login button's development name could be btnLogin. It may help to maintain the relationship between the design and the development.

Composite GUI elements like menus, lists, and dialogues may include nested GUI elements where required. An example is given in Figure 6.3.

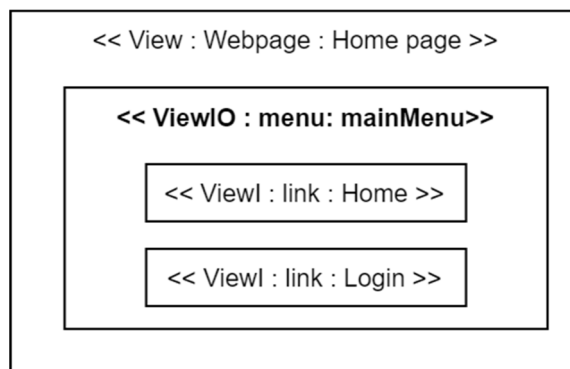


Figure 6.3 Example: View – nested GUI elements

This diagram explains a menu on the home page containing two links to home and Login. The menu element displays the available menu items and allows clicking them; therefore, it is considered *ViewIO*. The menu items are for clicking and navigating rather than showing the item name; hence, they are considered *ViewI*. Instead of explicitly defining the input and output elements, the RiWAsML provides space for the engineers to select the appropriate type according to the functions designed.

## Containers/Sections

Some available solutions offer similar elements.

- UWE's *presentation model* [142] uses named boxes with icons to illustrate containers: *presentationGroup* and *inputForm*.
- IFML [124] uses a model-element named *view component* to define a section or a widget, which is shown using a grey, rounded corner rectangle with a stereotype label to indicate the type of the element as a <<list>>, <<form>>, etc. and a name to denote the actual implementation of the element as in message list or customer information. IFML also allows marking an abstract section using a rounded corner rectangle drawn in a dotted line and then using a separate diagram to design lower-level aspects. It helps keep the diagrams more readable as the IFML uses many notations to include the interaction flow details, which may increase the diagram's complexity.

The RiWAsML suggests using the container/section only for content separation for readability. Inspired by the IFML, the RiWAsML gives a grey, rounded corner rectangle without a label. Some text could be used to state the function implemented in the section, which can be the section heading of the actual development. For example, consider the view in Figure 6.4, which contains a registration form for new members and a login form for the existing members on the same view.

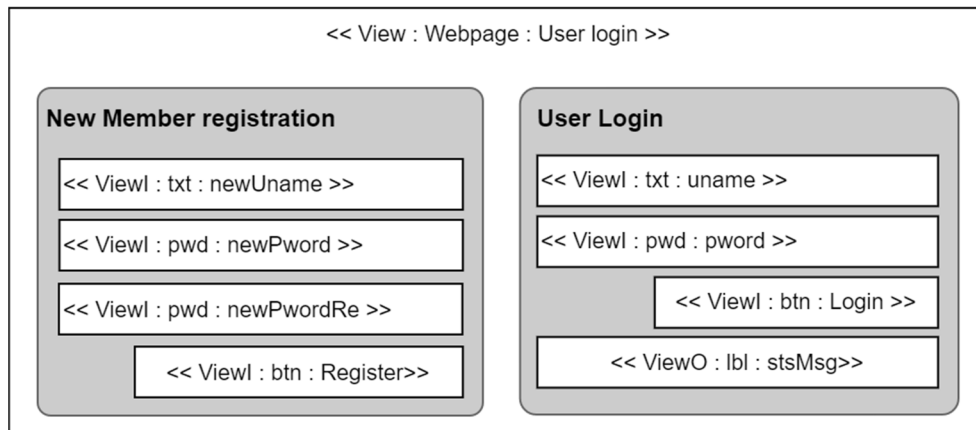


Figure 6.4 Example: View – GUI containers/sections

These containers are used to separate the content on the design for clarity and may or may not be used in the actual development.

## Popups/Toggles

Similar elements in available solutions are discussed below.

- UWE's *presentation model* [142] uses boxes with icons to include popups in a *presentationPage* as *presentationAlternatives*. UWE's approach is to design the popup in a separate presentation diagram as a *presentationAlternative* and include it as an object in the *presentationPage*. This is a good technique to reduce the design's complexity towards improving readability.



- **IFML** [124] provides two concepts named *modal window* and *modeless window* to design the popups, where the *modal window* blocks the main window, and the *modeless window* allows the main window to be interactive. Blocking/non-blocking factors are essential to consider. Moreover, IFML uses a notation named *XOR View Container*, which comprises “*child View Containers that are displayed alternatively*” [124].

RiWAsML provides popup and toggle elements whose **element** segment of the label should be Popup or Toggle. For popups, the **type** segment should specify if it is “Blocking” or “NonBlocking”; for toggles, it should specify the initial state as “Show/Hidden/Enabled/Disabled”. The **name** segment of the label may contain a suitable name for the popup or toggle. An example of the proposed notation is illustrated in Figure 6.5.

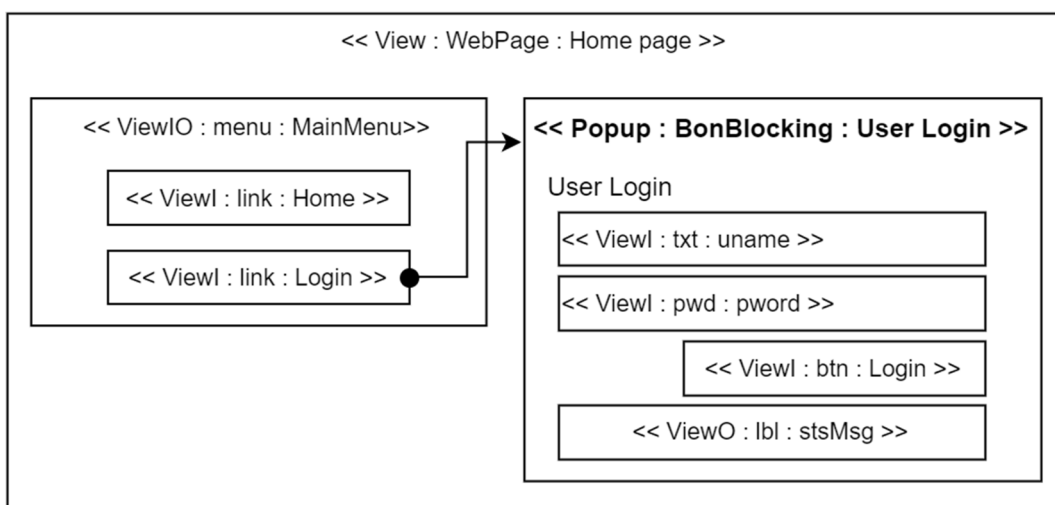


Figure 6.5 Example: View – GUI popup

The element on the main view, which triggers the popup/toggle to change its state, should be shown by drawing an arrow from the triggering element to the popup/toggle; a black circle at the beginning of the arrow indicates the GUI element triggers an event to cause the popup/toggle to change the state (refer to section 6.1.2.1 for more discussions on event handling). The diagram in Figure 6.5 states that the *Login* link in the menu opens the *User Login* popup on the same view instead of navigating to a different view. The internal elements of the popups/toggles can also be designed using the input/output elements.

Note that some *View* elements may directly invoke the popup/toggle without utilising an event handler. For example, Bootstrap provides many GUI elements/widgets to implement popups/toggles for browser-based views without the use of a controller. Some bootstrap widgets provide JS code; nevertheless, these codes can be considered a part of the view instead of the controller.

### Viewparts, ActorViews, and ViewPackages

The literature survey results show that the available methods/tools do not address these aspects explicitly. The RiWAsML offers notation for the *Viewpart* and related aspects discussed in this section.

The *Viewpart* uses a rectangle notation with a label; the **element** segment should be “Viewpart”, and the **type** segment should state the target actor of the *Viewpart*. If there are multiple actors, a comma-separated list can be provided. The **name** segment should indicate the functions implemented by the *Viewpart*. The RiWAsML gives 3 approaches to using *Viewparts* in a design, considering the case’s complexity.

1. **Less complex scenarios:** If a view has a few *Viewparts*, each including only a few GUI elements, the *Viewparts* can be included in the same view. Refer to Figure 6.6 for an example. The figure says that only an admin can see the Viewparts, which include the *add member* form on the top of the *View* element and the delete button on the member table.

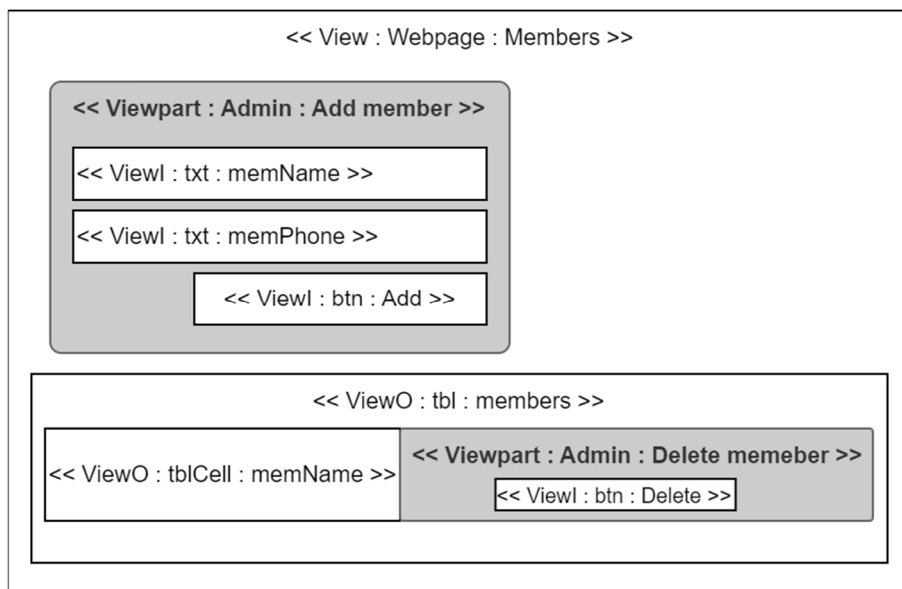


Figure 6.6 Example: View – Viewpart notation for less complex scenarios

2. **Moderately complex scenarios:** When including *Viewparts* on the same view makes the diagram complex, the RiWAsML recommends designing the *Viewparts* separately and wrapping all the view’s content using a *ViewPackage* element. For the label of the *ViewPackage*, the **element** segment should be “ViewPackage”, the **type** should be the containing view’s type, and the **name** should be the view’s name. The *Viewparts* should be designed outside the owning *View*, and placeholders for the *Viewparts* should be included inside the *View*. The placeholder should use the *Viewpart* object element without internal content; the **element** segment of the label should indicate that it’s an object of the actual *Viewpart* by using “ViewpartOBJ”. Figure 6.7 provides an example of the proposed notation.

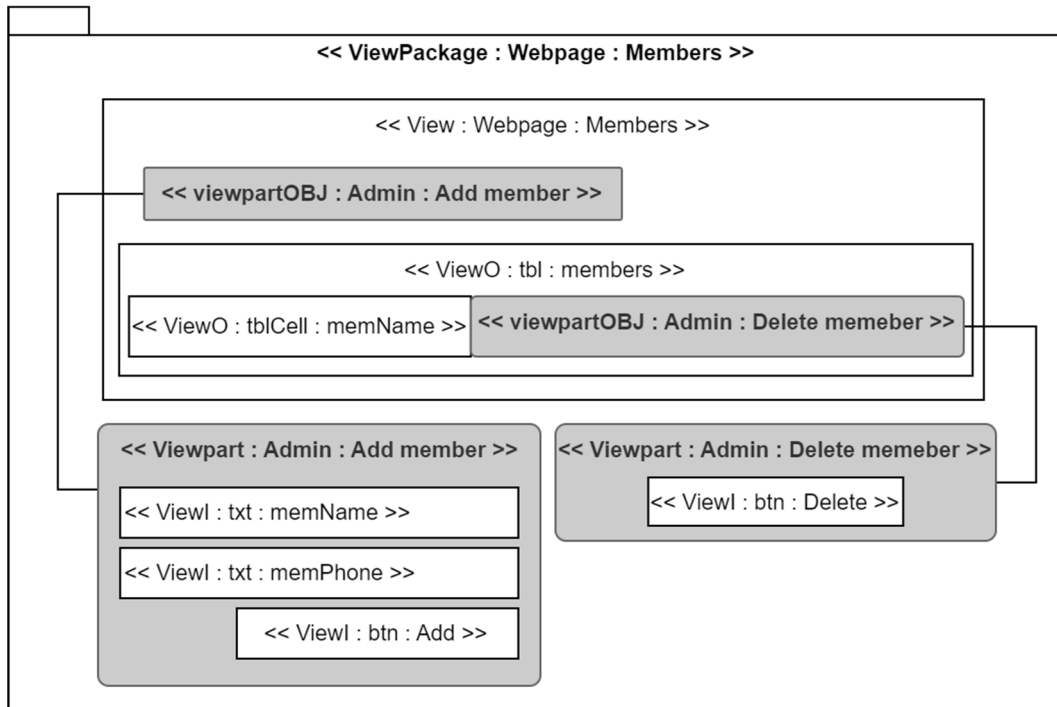


Figure 6.7 Example: View – Viewpart notation for moderate complex scenarios

The *Viewpart* object’s actual *Viewpart* design can be indicated using an association link for the clarity and readability of the diagram.

3. **Highly complex scenarios:** when the different *Viewparts* hold more distinguished content, the RiWAsML suggests designing them independently and wrapping them into a *ViewPackage*. The *ViewPackage* notation is similar to the moderately complex scenarios. Independent views are required to use “ActorView” for the **element** segment of the label and the target actor for the **type** segment. The **name** segment may contain a suitable name for each actor’s *View* or use the same name as given in the package, depending on the scenario.

An example of the proposed notation is shown in Figure 6.8, which models a scenario where the members can see the available member list using the “View member” *ActorView* web page, and admins can perform the add and delete operations on the member list using the “Manage Members” *ActorView* web page. Both *ActorViews* are actually implemented in a single web page named *Members* as specified on the *ViewPackage*; it should be understood that the web page’s development name is *Members*, and it’s presented to each type of actor using a display name as specified in each *ActorView*. The display names can also be the same as the development name where necessary.

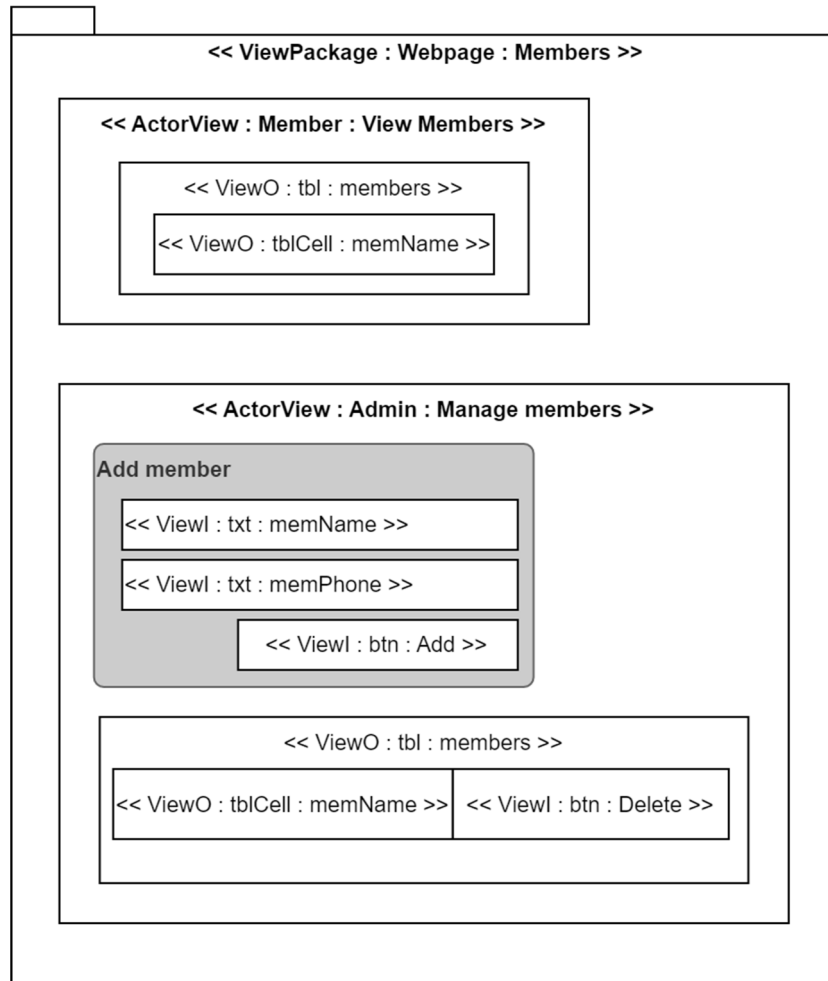


Figure 6.8 Example: View – Viewpart notation for highly complex scenarios

### SharedViewparts

UWE allows the modelling of the menus using separate nodes in the *Navigation Model* [147]; however, UWE does not discuss whether it is sharable. The RiWAsML offers an element named *SharedViewpart* to design the shared content separately and indicate its use on a view using an object similar to the *ViewPart*. The **element** segment of a *Viewpart* with shared content should use “SharedViewpart”, and the **type** segment should state the type of the content, for example, header, footer, or menu. The **name** segment may use a suitable name for identification.

The *SharedViewpart*’s placeholder on the *View* should use “SharedViewpartOBJ”. The *SharedViewpartOBJ* may use association links to connect to the *SharedViewpart* to improve the diagram’s readability. Refer to Figure 6.9 for an example of the proposed *SharedViewpart* notation.

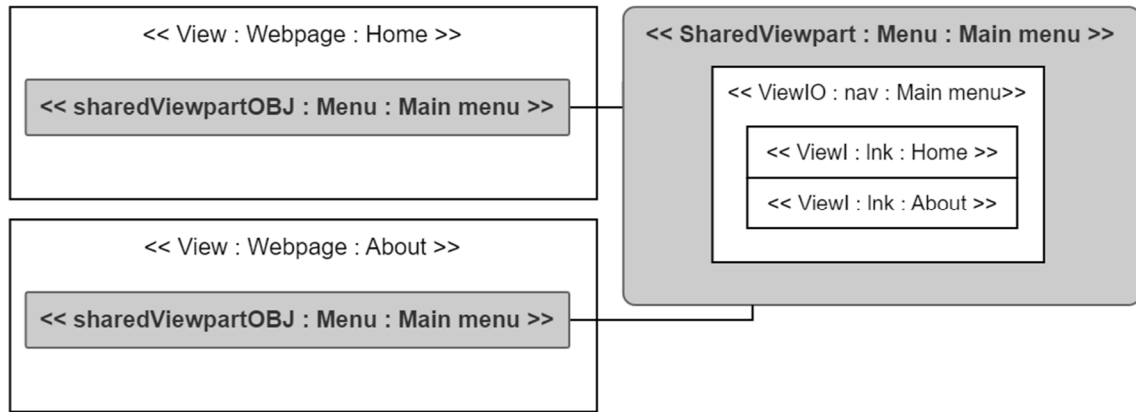


Figure 6.9 Example: View – SharedViewpart

The same techniques discussed in the previous section under *Viewparts* can be used if different actors require dedicated content in a *SharedViewpart*; an example is given in Figure 6.10. As per the diagram in this Figure, the *SharedViewpart* named *Main menu* provides *Home* and *Manage members* links for the admins and *Home* and *View members* menus for the members.

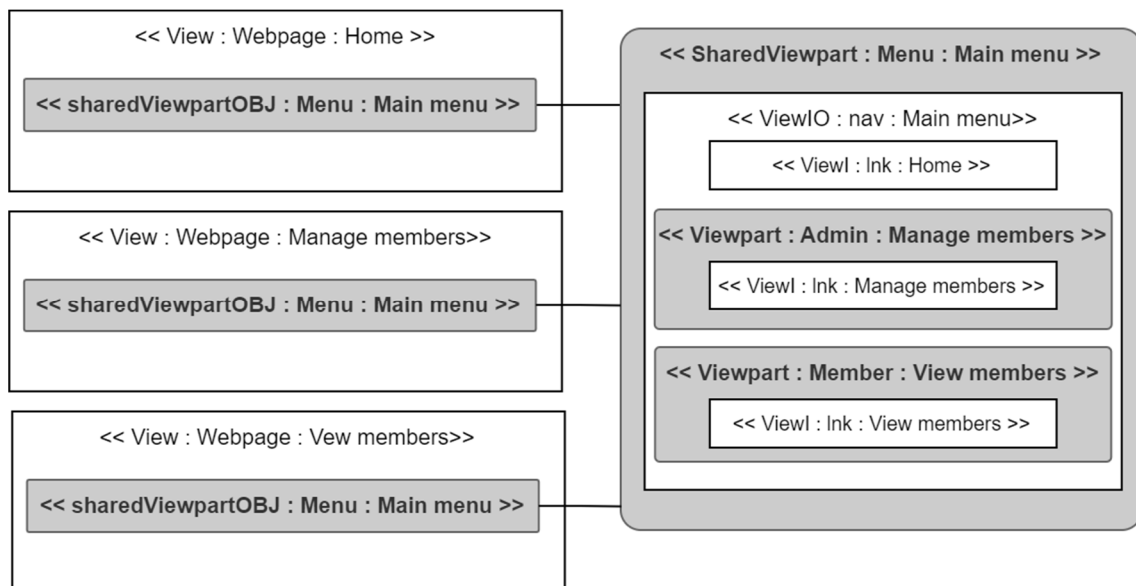


Figure 6.10 Example: View – SharedViewpart with nested Viewparts for different actors

### 6.1.1.2. Navigation

Section 4.4.1.2 sets the requirements for the RiWAs navigation aspects. Many available solutions and research work have looked into navigation aspects, which are briefly stated in Sections 3.3 and 3.4. Some of the significant work is detailed below.

- UWE provide a model named *Navigation model* [147] to capture the navigation details using an approach with node-link where nodes are not only the web pages but also can be menu, index, query, or processClass, and a link can be either a navigational link or a process link. UWE's *Navigation model* has a higher learning curve, and it can be much more complex as the number of views increases due to the various types of nodes and their links. UWE's *Navigation model*

[147] uses *processLinks* to depict the process-based navigation between the nodes. The nodes' details do not assist in understanding their implementation as in a view, a view section, or a popup. Also, the details of the GUI elements triggering the process are not included in the navigation model.

- IFML's [124] focus is on interactive flows; therefore, IFML does not address link-based navigation. However, while modelling the interactive flows, IFML captures some process-based navigation. Even though the IFML allows the events to be included in the view components, it does not explicitly indicate the GUI elements that trigger the events. The details of the action performed by the event are also presented in the design with the parameters bound to the action, mixing the pieces from views and controllers into the same diagram; still, some other required details supporting the actual development – such as elements triggering the event and the type of event – are missing.
- Popular GUI design tools like Adobe XD [100] and Figma [101] use storyboarding to implement prototypes [102] [103], which gives an overview of the navigation between the views.
- González et al. [167] have studied the available methods and introduced an MDA approach to design the navigational aspects, focusing on hypermedia, which is suitable for the RiWAs. However, their approach does not concern factors like views with *Viewparts* and multiple routes to navigate to views for different actors.

The RiWAsML's proposal for modelling the navigation in RiWAs is inspired by storyboarding, considering its simplicity. When designing navigation between views, authentication and authorization requirements of the RiWA should be taken into account while satisfying the requirements set in Section 4.4.1.3.

### **Link-based/Menu-based Navigation**

Link-based or menu-based navigation is the typical type of navigation in computer systems, which is valid for RiWAs. A shared menu can be modelled using a *SharedViewpart*, and the navigation is proposed to be modelled using RiWAsML's HTTP communication channel, denoting an HTTP request. When modelling the navigation, it is sufficient to show the request arrow from a view indicating the target view to navigate. Figure 6.11 provides an example of the menu-based navigation design, which is an updated version of the diagram in Figure 6.10.

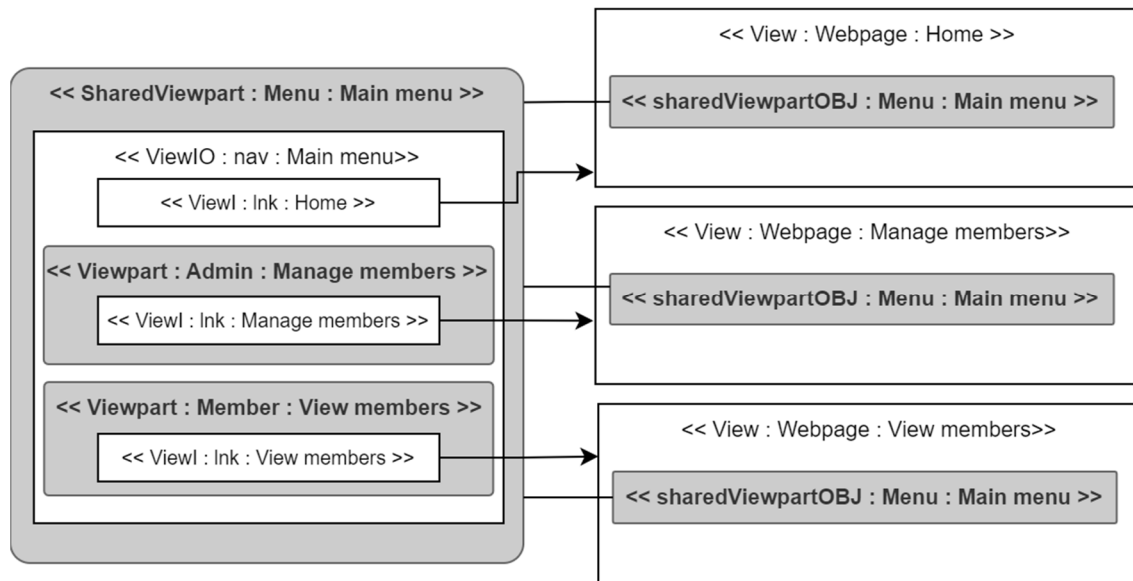


Figure 6.11 Example: Views – menu-based navigation design

The design in Figure 6.11 depicts a typical scenario of menu-based multi-paged RiWA. The two types of actors, *admin* and *member*, are given different main menu items where the admin gets *Home* and *Manage members* menu items, and the member is provided with *Home* and *View members* menu items.

This RiWA can be further improved by implementing a single view for *Manage members* and *View members*. In such cases, a common view or a *ViewPackage* with *Viewparts* for multiple actors can be introduced in the design. An example is given in Figure 6.12, which is an improved version of the diagram in Figure 6.11. This design also includes additional menu items and views to raise the complexity for demonstration purposes.

For navigation designing, detailed designs of the views are not required; including only the *View* or *ViewPackage* elements without internal elements – other than the shared menus – is sufficient. The detailed design of a *View* or the *ViewPackage* can be given in a separate diagram. For example, the detailed design of the common *ViewPackage* named *members* in Figure 6.12 can be considered the diagram in Figure 6.7. Note that the *View/ViewPackage* label in the navigation diagram and the detailed *View/ViewPackage* diagram should be the same.

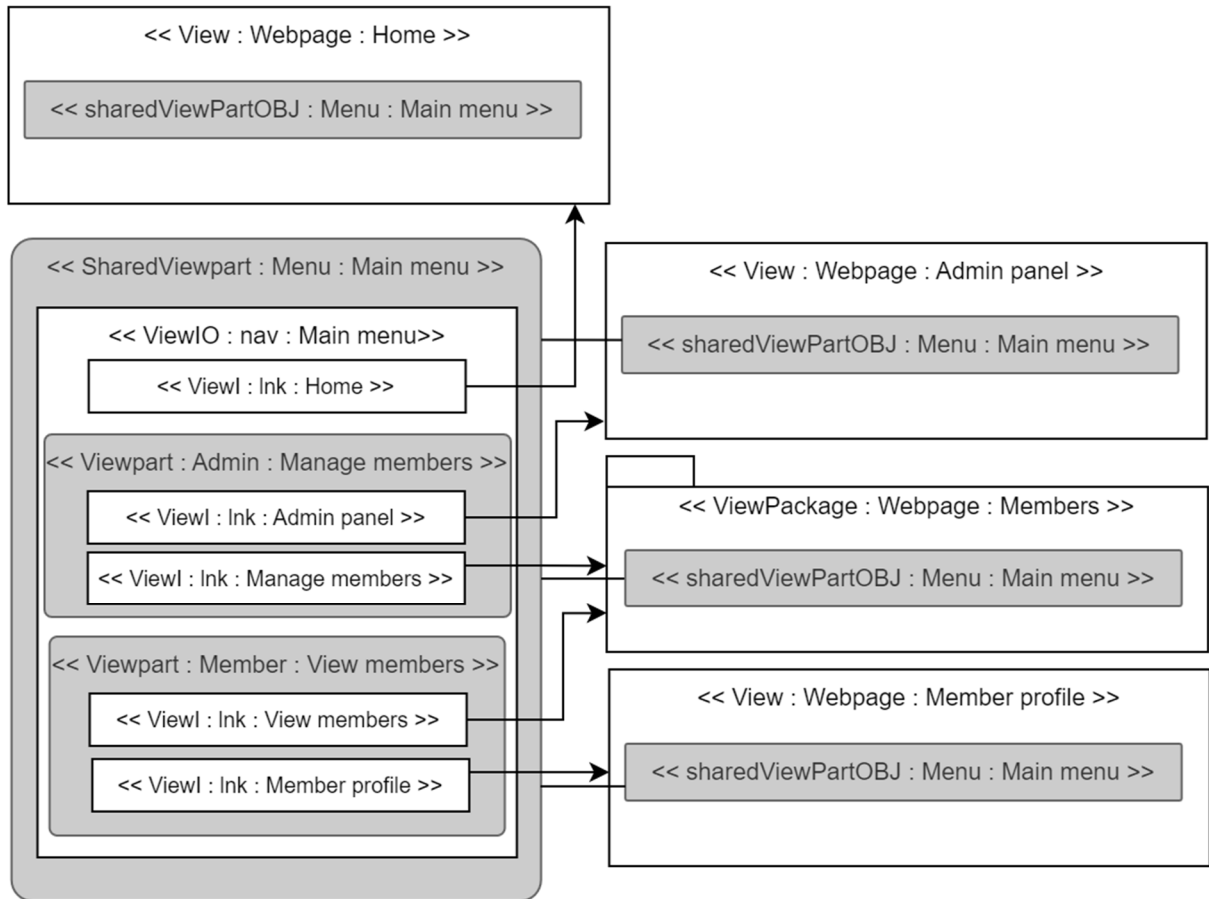


Figure 6.12 Example: views – menu-based navigation design with ViewPackage

### Process-based Navigation

The RiWAsML provides a more straightforward technique to capture only the navigation-related information of process-based navigation without mixing other process details. The proposed approach is to include the GUI element that triggers the process, which causes navigation, and to use an arrow from that triggering GUI element to the target view. An example is given in Figure 6.13.

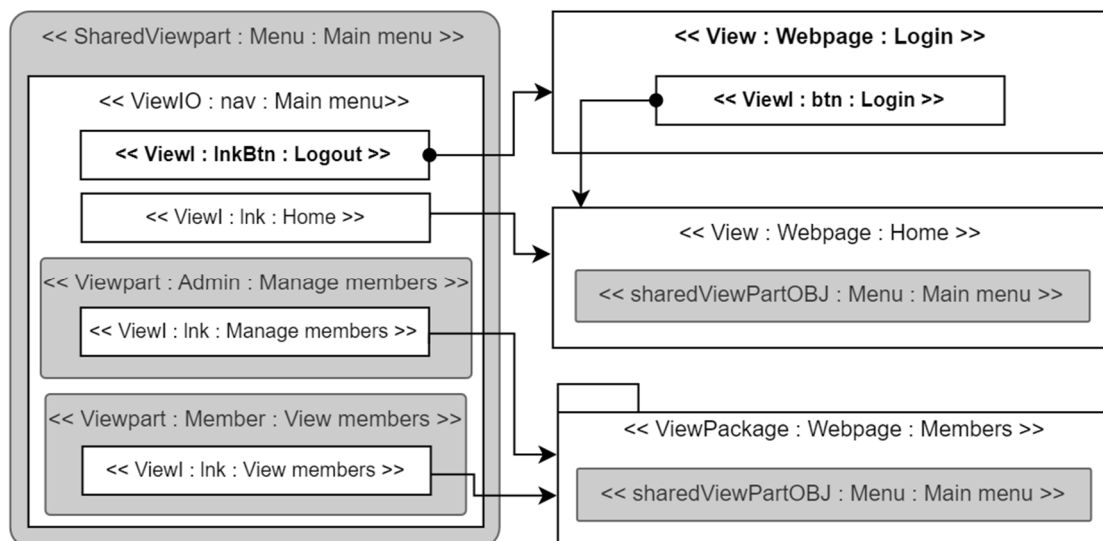


Figure 6.13 Example: Views – process-based navigation



This design explains that the *Login* button on the *Login* page triggers an event, which navigates the user to the home page. It should be understood that this navigation happens only on successful authentication. The other pages on the system use a shared *Main menu* containing the *Logout* link button to trigger an event to navigate to the login page. The *Logout* link button performs the logout function, which initiates a complete logout process involving the client and server processing elements instead of merely navigating the user to the login page.

This technique only captures the navigation and does not explain the logic of the process which causes the navigation. It should be understood that the navigation starting from hyperlinks is always link-based, and the navigation starting from other types of GUI elements is process-based.

### 6.1.1.3. Data

Views are often associated with datasets to display information on the GUIs; however, utilising and processing these data is a component task. Therefore, the view-related data discussions are delegated to the views' controllers in section 6.1.2.

### 6.1.1.4. Related Controller

Discussions on view-related controllers are allocated to section 6.1.2.

## 6.1.2. R8 – Notations for Low-level AppControllers and Related Elements

This section satisfies the requirements discussed in Sections 4.4.1.4 and 4.4.2.1. The high-level *AppControllers* element contains a set of controllers for the views within the *AppControllers* element's *Application* element.

### 6.1.2.1. ControllerClass Element

A controller is basically a collection of event handlers, related properties, and methods, and these event handlers can be developed using functions/methods; hence, a controller can be seen as a class. The UML tools [156], which support the Entity-Control-Boundary (ECB) pattern [154], also use the UML's class element to denote the controllers. The RiWAsML also use the UML's class element with specialized features. The RiWAsML's *ControllerClass* element notation is illustrated in Figure 6.14, along with its example use.

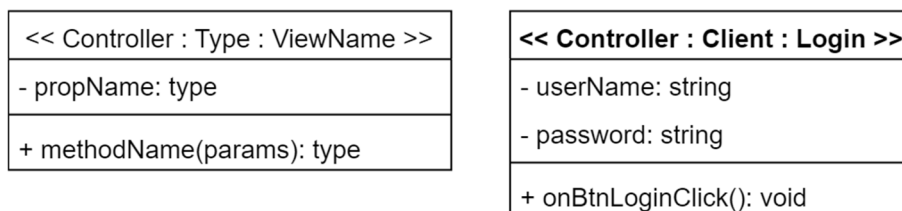


Figure 6.14 Proposed notation: *ControllerClass* element (on the left) and an example use of it (on the right)

The **element** segment of the *ControllerClass*'s label should be "Controller." Even though the controller is always on the client-side, for the consistency of naming, the **type** segment should be

indicated as “Client”. The *ControllerClass*’s label’s **name** segment should use the exact name of the related view’s name.

Note that the high-level element containing all the controllers is the *AppControllers* element (refer to section 5.2.4). When there are multiple client *Application* elements in a RiWA, there is supposed to be an *AppControllers* element per *Application* element. In such cases, RiWAsML suggests indicating the *Application* element’s type for the *ControllerClasses*’ type segment. For example, if there are two client *Application* elements for the browser and mobile phone in high-level design, then the type of the *ControllerClasses* in the browser app could be *BrowserClient*, and the type of the *ControllerClasses* in the mobile app could be *MobileClient*.

When modelling the communication between the *ControllerClasses* and other elements, the RiWAsML’s communication channels notations should be used instead of the UML meta-model’s standard interfaces: *Provided Interface*, and *Required Interface* [153]. The following sections will discuss the *ControllerClasses*’ communication with other elements, where necessary.

#### 6.1.2.2. View Events Handling in Controller

Some available solutions attempt to capture the details related to the views’ events handling.

- **UWE**’s *Navigation model* [147] tries to capture the event-based navigation but ignores the event-related details, such as which elements trigger the events, which structures/functions/methods handle the events, and what data requirements are related to the events. UWE uses a stereotype arrow <<processLink>> to indicate the process-based navigation upon triggered events.
- Compared to UWE, **IFML** [124] captures more details related to the events using the model-elements *Catching Event*, *Throwing Event*, and *Activation Expressions*. IFML indicates the events on the GUI elements, expressing the relationship between the GUI elements and the events. IFML uses an arrow with a circle at the beginning of the GUI element, which triggers the event to another GUI element or an *Action*. Further, IFML states the *Actions* related to the events on diagrams. IFML’s primary purpose is to capture the interaction flows, and the details required for the implementation of the event handlers or *Actions* are not concerned.

Event triggering is a special type of communication between the view and controller. Therefore, the RiWAsML proposes to use a new communication channel to denote a view that triggers an event to invoke a handler of its controller. This communication channel notation specialises the view-controller communication channel given in section 5.1.2 by using a black circle at the beginning of the arrow, inspired by the IFML, as introduced in Figure 6.5 in Section 6.1.1.1. An example is denoted in Figure 6.15.

This channel may also indicate that the event handler reads data from the view, eliminating using another communication channel to denote the data reading. The RiWAsML recommends the following options to denote event-triggers, depending on the complexity of the design.

### Less Complex View and Controller

When the view is not very complex, and there are a few event handlers in the controller, include both the *View* and the related *ControllerClass* on the same diagram, as shown in Figure 6.15.

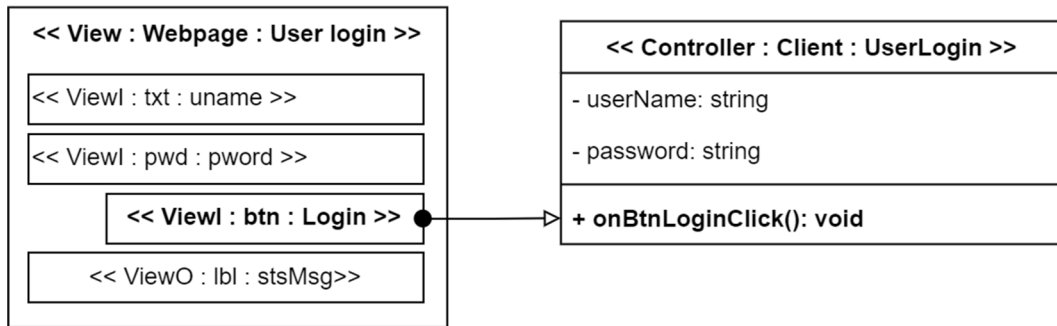


Figure 6.15 Example: View and its ControllerClass – event-trigger notation for a less complex scenario

Note that the *View*'s name “User login” is used for the *ControllerClass* in the class name format as “UserLogin.” The *ControllerClass* implements the *onBtnLoginClick()* event handler for the *View*'s *Login* button, and that relationship is depicted using an arrow from the button to its event handler. This relationship inherits the view-to-controller relationship from the high-level *L2 View-Process* model (see Figure 5.20 in Section 5.3.4). The RiWAsML suggests using the event handlers' names in the format - *on* + *GUI elementName* + *Event* - reflecting the event to handle, inspired by development languages/libraries/frameworks such as JavaScript, jQuery, and ASP.Net.

### Complex View And Controller

The RiWAsML provides notations for complex cases to design the *View* and its *ControllerClass* on separate diagrams and link them using communication channels with numbered flow connectors, as shown in Figure 6.16. Note that two different diagrams are drawn on the same figure for demonstration purposes.

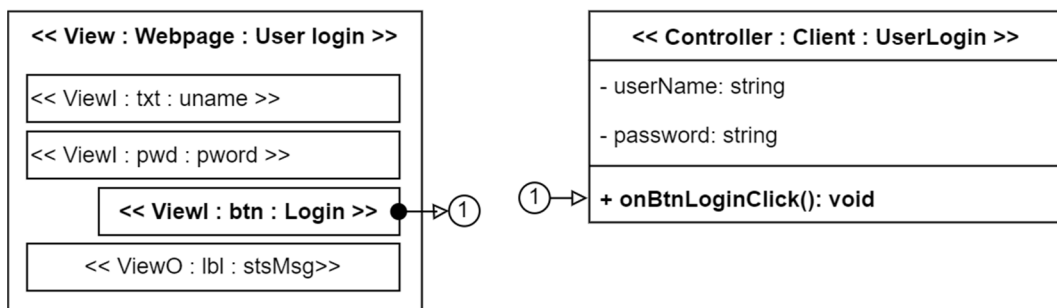


Figure 6.16 Example: View(on the left) and its ControllerClass (on the right) – event-trigger notation for a complex scenario

### 6.1.2.3. Read Data from View in Controller

When an event is triggered to initiate a process, the event handler may read data from its *View* for processing. The RiWAsML proposes to use the event handler's parameters to model this function, denoting the data to be read from the *View*. See Figure 6.17 for an example of the proposed notation.

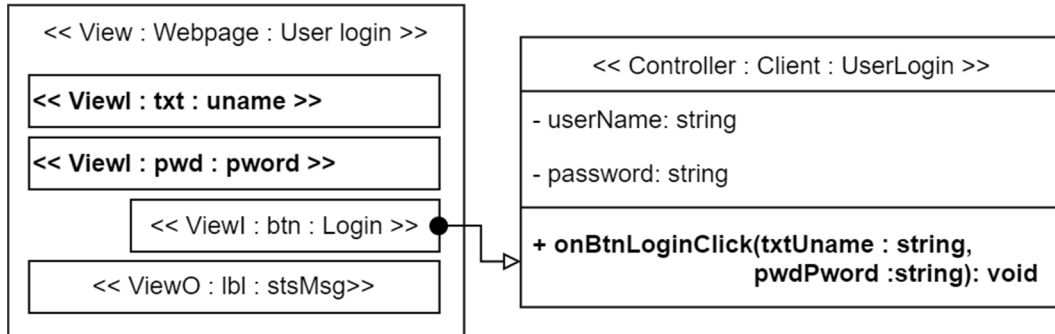


Figure 6.17 Example: View and its ControllerClass – event handler reading data from the view

The event handler's parameters follow the UML class notation as in *paramName : datatype*. The parameter name of an event handler is proposed to use the combination of the type and name segments of the view's GUI elements. The string value of the *User login* view's textbox named *uname* is specified as a parameter named *txtuname* for the *onBtnLoginClick()* event handler; similarly, the *pwdPword* parameter. When there are many elements from which to read data, it is advised to use a form element on the *View* and specify the form as the input parameter in the event handler; in that case, the data type could be defined as an object.

### 6.1.2.4. Show Information on View by Controller

The RiWAsML uses the response arrow from an event handler to a GUI element on the *View* to denote that the event handler is producing an output on the GUI element. Figure 6.18 depicts an example of this function.

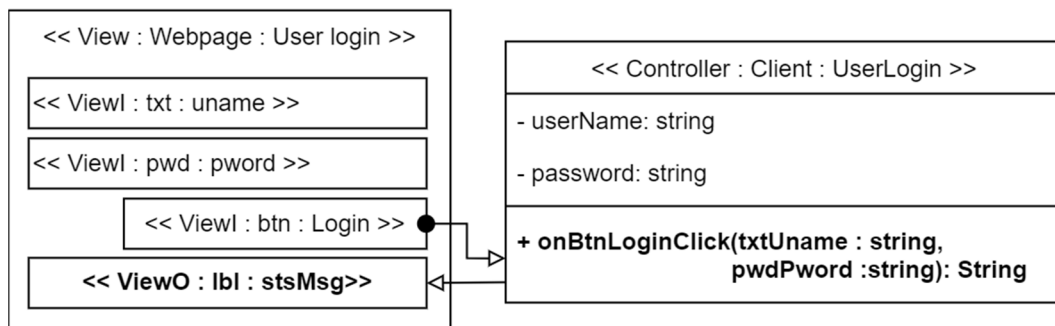


Figure 6.18 Example: View and its ControllerClass – event handler showing output on the View

This diagram explains that the *onBtnLoginClick()* handler's output is dynamically shown on the *stsMsg* label. Note that the event handler's return type is *String*. For example, an error message is shown on the *stsMsg* label when the username or password is wrong. It is possible for an event handler to update multiple view elements or sections to display output.

In complex situations, the output arrow may also use a numbered connector. An example is given in Figure 6.19.

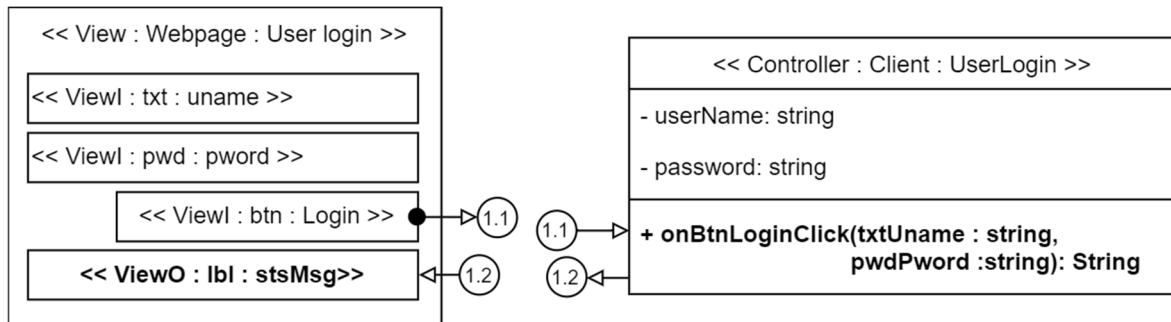


Figure 6.19 Example: View and its ControllerClass – event handler showing output on the View – communication channels with numbered flow connectors

The RiWAsML recommends using multi-level numbers for the flow connectors to group the request-response pairs. Note that in the Figure 6.19 diagram, the first level number is the same – in this case, (1) – for all the flow connectors, denoting they belong to the same request-response pair. The second level uses (1) for the request and (2) for the response. In the case of an event handler updating multiple view elements, all the output view elements should use the same corresponding number on the flow connector.

#### 6.1.2.5. Invoke Popups/Toggles by Controller

The popups/toggles are invoked by an event of another GUI element, and this event’s handler invokes the popup/toggle to change the state. The event-triggering GUI element to event handler communication is the same as discussed in the previous section. Event handler invoking the popup/toggle requires a specific notation as it should realise the invocation type: show, hide, enable, and disable.

The options to depict the invocation of a popup/toggle are (1) using new arrow notation and (2) using a stereotype label on the *Controller* to *View* communication arrow. The stereotype label was considered the better option towards maintaining usability for two reasons: (1) introducing new arrow notations for 4 different types may reduce the usability, and (2) the stereotype label only requires 7 or less characters to denote the invocation type.

To show the state change of a popup/toggle between show/hide/enable/disable, RiWAsML recommends using <<show>>, <<hide>>, <<enable>>, or <<disable>> stereotype labels appropriately. An example of this notation is given in Figure 6.20.

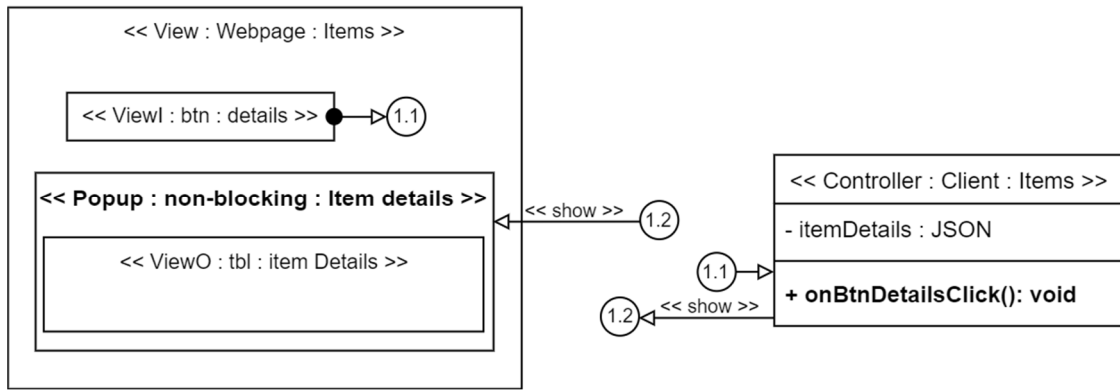


Figure 6.20 Example: View and its ControllerClass – invoking a popup/toggle

It should be understood that the popup’s default state is hidden, and when clicking the *Details* button, the *onBtnDetailsClick()* event handler shows the popup. Usually, the popups have a close button to hide it.

### 6.1.2.6. Use the Client-Model by Controller

The only communication between the controller and the client-model is when the *ControllerClass* calls a method in the *Client-ModelClass* and receives the return value. Hence, RiWAsML proposes to use the bi-directional communication channel to denote this scenario, inherited from the *L2 View-Process* model referring to Figure 5.20 in Section 5.3.4. An example of this communication is illustrated in Figure 6.21. Refer to Section 6.1.3 for the notation of the *ModelClass*. Note that it’s not only the event handlers that utilise the client-model; any method in a controller may communicate with the client-model as required.

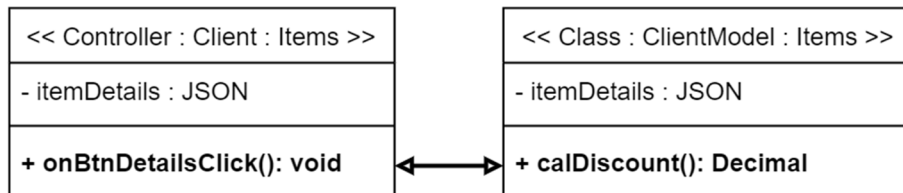


Figure 6.21 Example: ControllerClass – communicating with the Client-model

A flow connector can be used for complex diagrams, as discussed previously, and an example is given in Figure 6.22.

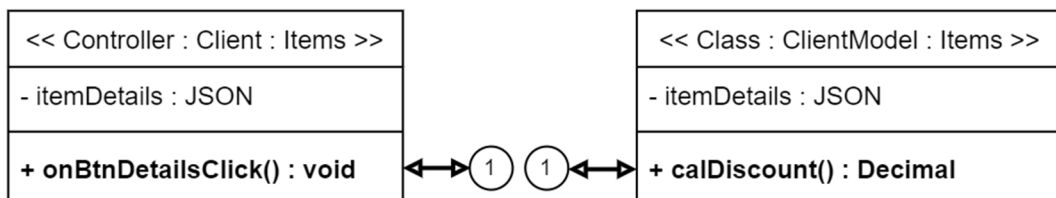


Figure 6.22 Example: ControllerClass – communicating with the Client-model – numbered flow connector notation

### 6.1.2.7. Use the Server-Model by Controller

The flow of controller-to-server-model communication is not straightforward as it happens via the DC connectors. It was noted during the literature survey that none of the available solutions address this and provide tools to model the DC-related aspects. RiWAsML uses a development-based approach to realise the mechanism of the communication between the controller and the server-model to propose model-elements and the ways of utilising them.

From the controller's perspective, even though the controller's intention is to communicate with the server-model, the immediate communication happens with the DC-engine, where the DC-engine communicates with the DC-bus. This section only focuses on modelling the communication between the controller and the DC-bus through the DC-engine.

The communication between the controller and the DC-engine is discussed in two steps, focusing on their development nature.

#### DC Request Through DC-engine

In actual development, the DC-engine can be a tool provided by a language/library/framework. For example, for pull-DC in browser-based RiWAs, JS Fetch API [168] or jQuery's AJAX function [169] can be utilised as the DC-engine. Consider the following sample code of an event handler in a *ControllerClass*.

```

$(document).on("click", "#btnDeleteItem", function (event)
{
    event.preventDefault();
    event.stopPropagation();

    itemID = $(this).val();

    $.ajax({
        url: "DC-bus/items/" + itemID,
        type: "DELETE",
        processData: false,
        contentType: "application/x-www-form-urlencoded",
        complete: function(response, status) {
            onDeleteItemDCComplete(response.responseText, status); }
    });
});

```

The handler is developed for a delete button to delete an item, where the handler sends an AJAX request to the server using jQuery's ajax() function. The ajax() function can be seen as the DC-engine, and the *ControllerClass*'s event handler's call to the ajax() function can be considered the controller to DC-engine communication. On the design, this can be denoted by indicating which event handler on the *ControllerClass* is utilising the DC-engine.

RiWAsML proposes to use a thick arrow – which is the DC communication channel – from an event handler or any other method to denote that the event handler or the method is utilising the DC-engine to send a DC request, as shown in Figure 6.23.

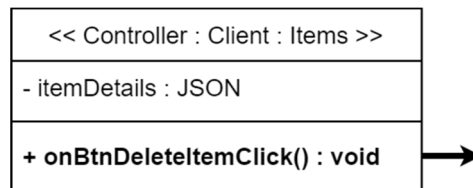


Figure 6.23 Example: ControllerClass – sending a DC request

This notation explains that the event handler utilises the DC-engine and sends a DC request even though the DC-engine is not included as a model-element on the diagram. The RiWAsML considers the DC-engine is a part of the *ControllerClass* and will not offer a model-element for the DC-engine.

### DC Response Handling

The RiWAsML uses a similar technique of depicting the DC request on the *ControllerClass* for the DC responses. In the previous sample code, the last line in the `ajax()` function registers a new event handler named `onDeleteItemDCComplete` to handle the DC response of that particular DC request, which should be developed separately within the *ControllerClass*. The sample code for this DC response handler is given below.

```

function onDeleteItemComplete(response, status)
{
    if (status == "success")
    {
        $("#lblStatus").html("Item is deleted successfully ");
    }

    if (status == "error")
    {
        const error = JSON.parse(response);
        $("#lblStatus").html ("Error: " + error);
    }
}
  
```

RiWAsML proposes the notation given in Figure 6.24 to model this implementation by updating the *ControllerClass* of Figure 6.23.

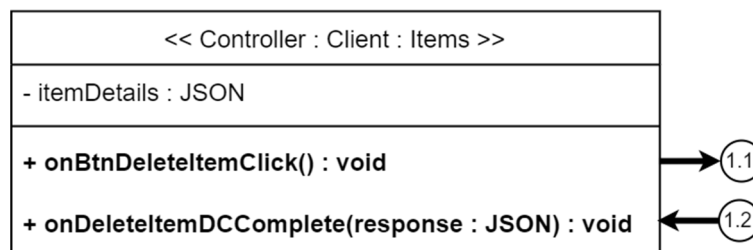


Figure 6.24 Example: ControllerClass – DC response handling



The *ControllerClass* in Figure 6.24 uses numbered flow connectors, which denote the communication as a request-response pair. This diagram explains that the *onBtnDeleteItemClick()* event handler sends a DC request, and the *onDeleteItemDCComplete()* event handler is set to handle the DC response for the said DC request. The *onDeleteItemDCComplete()* event handler accepts the results from the server as a JSON parameter. The other end of this communication should be designed in a separate *DC-bus* diagram, as discussed in Section 6.1.3.

### 6.1.3. R8 – Notations for Low-level AppModel and Related Elements

High-level *AppModel* elements are comprised of a set of related *ModelClasses*. This section introduces the model-elements required for the *AppModel* elements as specified in Section 4.4.2.2.

#### 6.1.3.1. ModelClass Element

The model component implements the business log; hence, designing the model is straightforward, as the **UML** provides the class diagram for this purpose. The available solutions also utilise the UML’s class diagram to design business logic, generally called the domain model. RiWAsML follows the same method; however, the class notation is altered to align with the rest of the model-elements using the RiWAsML *Label* element. The proposed class notation is given in Figure 6.25.

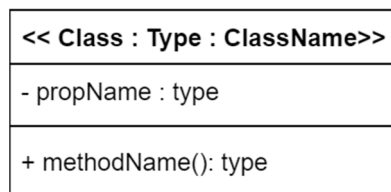


Figure 6.25 Proposed notation: ModelClass

The **element** segment of the *ModelClass*’s label should be “Class”, and when an object of the class should be represented, the element segment may use “Object”. The **type** should be either “ClientModel” or “ServerModel” (or only “Client” or “Server”), and the **name** segment should state the class’s name.

The class syntax is for the detailed design of a single class; the RiWAsML recommends using a class diagram to capture all the classes and their relationships of a particular *AppModel* element and wrapping the class diagram using the *AppModel* element, as shown in Figure 6.26.

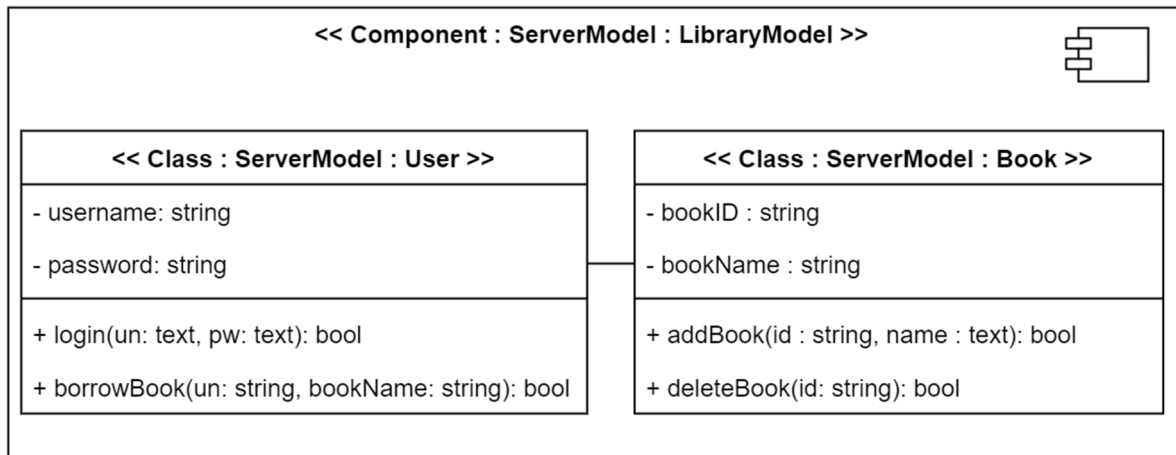


Figure 6.26 Example: AppModel diagram of a ServerModel

In this diagram, it is enough to state the class names of the classes instead of using the complete RiWAsML element label and denoting the classes' properties and methods is optional. When the detailed design of a particular class is required, that class can be designed in a separate diagram, including all the required details (refer to Section 6.2.4).

### 6.1.3.2. Communication with Other Elements

The RiWAsML provides a thick white-headed arrow to denote utilising the *AppModels* (refer to Section 5.1.2), and the same syntax should be used on the *ModelClasses* to indicate the utilisation of the methods. Unlike HTTP or DC request-response pairs, a *ModelClass* method is called, and the return is captured by the same event handler or method of other elements; hence, a single bidirectional arrow can be utilised for a low-level *ModelClass* element's method. An example of communication between a *ModelClass* and other elements is given in Figure 6.27.

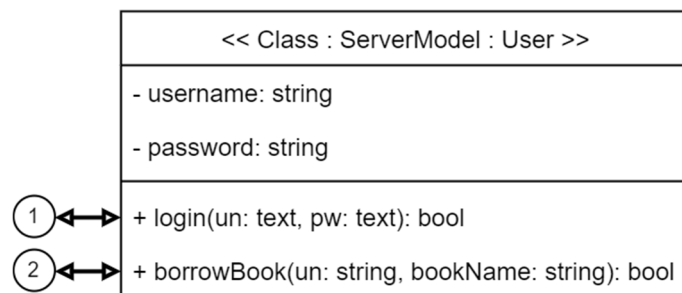


Figure 6.27 Example: ModelClass – communicating with the other elements

In the case of client-model, the event handlers or methods in the *ControllerClass* elements call the methods of the *ModelClass* elements. For the server-model, the *DCBus* utilises the *ModelClass* elements. It is possible for multiple controllers to utilise the same method in a *ModelClass*; in such a case, all the flow connectors of the controllers should use the same connector number of the *ModelClass* method.

*ModelClass* elements can use UML’s standard relationship notations as required, such as association, generalization, composition, aggregation, use, multiplicity, interfaces, and packages.

#### 6.1.4. R9 – Notations for Low-level Connectors and Related Elements

Even though there are two types of connectors (DC-engine and DC-bus), the DC-engine is considered part of *ControllerClass* elements (refer to Section 6.1.2.7); therefore, low-level notations are required only for the DC-bus and related aspects. This section satisfies the requirements set for the DC-bus in Section 4.4.3.2.

##### 6.1.4.1. EndpointsCollection Element

The *EndpointsCollection* element is defined in Section 4.4.3.2. When designing the low-level details of the DC-bus, it’s recommended to group the API endpoints based on OODD towards improved semantics, and implement them as *EndpointsCollection* units. A *DCBus* connector element may contain one or more *EndpointsCollection* elements. In actual development, an *EndpointsCollection* can be developed in a separate class or script/file for better management. Based on this notion, the RiWAsML offers UML class-like syntax for the *EndpointsCollection* element, as shown in Figure 6.28.

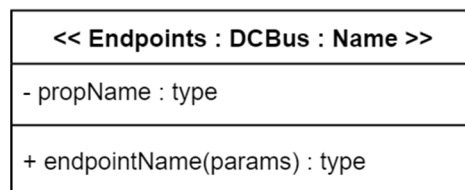


Figure 6.28 Proposed notation: *EndpointsCollection* element

The **element** segment of the label should be “Endpoints”, and the **name** segment may contain a suitable name for the group of endpoints. The **type** segment should be *DCBus*; however, it is possible to implement *EndpointsCollection* elements of the type *HTTPBus* to serve the standard HTTP requests from the views and controllers, even though it is not included in the RiWAArch style. If HTTP endpoints are required, a server-side *Application* element may include an *HTTPBus* component element, which comprises the *EndpointsCollection* elements of *HTTPBus* type.

##### 6.1.4.2. Communication with ControllerClass

RiWAsML proposes the two-level numbered flow connector notation for the *EndpointsCollection* element to denote the communication with the controller via the DC-engine, as discussed in Section 6.1.2.7. The communication channels should use thick-lined arrows since they are for DC. An example of the use of the notation is given in Figure 6.29. This *EndpointsCollection* can be considered the one communicated by the *ControllerClass* in Figure 6.24 in Section 6.1.2.7.

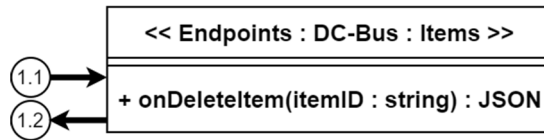


Figure 6.29 Example: *EndpointsCollection* – communicating with the controller

The *EndpointsCollection* named *Items* in the DC-bus accepts a DC request from the controller to delete an item. The *onDeleteItem()* endpoint takes the *itemID* as a parameter from the controller and responds with a JSON object.

Instead of ruling out how the endpoints should be grouped and how the *EndpointsCollections* should be implemented, the RiWAsML recommends following the ODD practices. A suitable technique/technology like SOAP or REST with Ajax can be used to implement the endpoints in DC-bus.

### 6.1.4.3. Communication with the Server-Model

To denote the communication between an endpoint in an *EndpointsCollection* and a method of the server-model’s *ModelClass*, a bi-directional arrow is proposed by the RiWAsML. Figure 6.30 provides examples of the use of the direct flow notation and numbered flow connector notation for denoting the communication between an *EndpointsCollection* and a *ModelClass*.

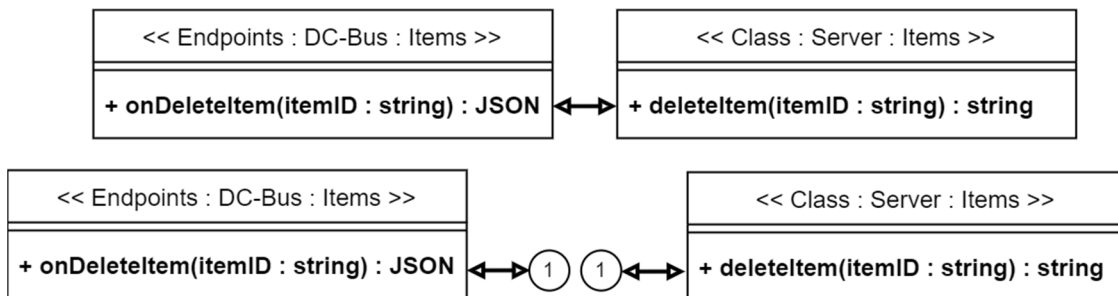


Figure 6.30 Example: *EndpointsCollection* – communicating with *ModelClass*

The diagrams in Figure 6.30 show that the *onDeleteItem()* endpoint calls the *deleteItem()* method on the server-model’s *Items* class. The *deleteItem()* method returns a string value – consider it as a status message – and the *onDeleteItem()* endpoint wraps it in a JSON object with other related data as required, which will be sent to the controller as the response.

Note that the RiWAsML does not specify the data and their structures for communication at this level, and it’s considered the designer’s responsibility to select them according to the scenario.

## 6.2. R10 – Low-level Design Models and UML Profiles for the RiWAsML

This section proposes models to design low-level aspects of the RiWAs utilising the notations offered in Section 6.1. Also, UML profiles for these models and model-elements – which extend the UML meta-model based on OMG’s Meta-model Hierarchy [85] – are given. These models satisfy the requirements set for them in Section 4.4.4.

### 6.2.1. View-Navigation Model

The requirements for the *View-Navigation* model are set in Section 4.4.4.1.

Some available solutions attempt to capture the navigational aspects of the web applications and RIAs. They primarily model the navigation between the views regardless of considering the functions implemented in the views and the navigation of the different user types.

- UWE’s *Navigation Model* [147] captures both link-based and process-based navigation. It does not provide a clear understanding of the set of views in the system and the functions they implement.
- IFML [124] does not intend to model the navigation; nevertheless, the process-based navigation can be understood to some extent through the interaction flows.

An example of a *View-Navigation* diagram is given in Section 8.3.2.1, and how it satisfies the expected features is also discussed. The RiWAsML’s *View-Navigation* model is not merely a behavioural model; it captures structural details, such as which views implement multiple related functions, and assists in making decisions on merging/splitting views to improve the user experience.

The UML profile for the *View-Navigation* model is given in Figure 6.31. Note that this profile contains a subset of the *View* model’s elements presented in the next section, which are required to depict the views and navigation-related elements in the *View-Navigation* diagram.

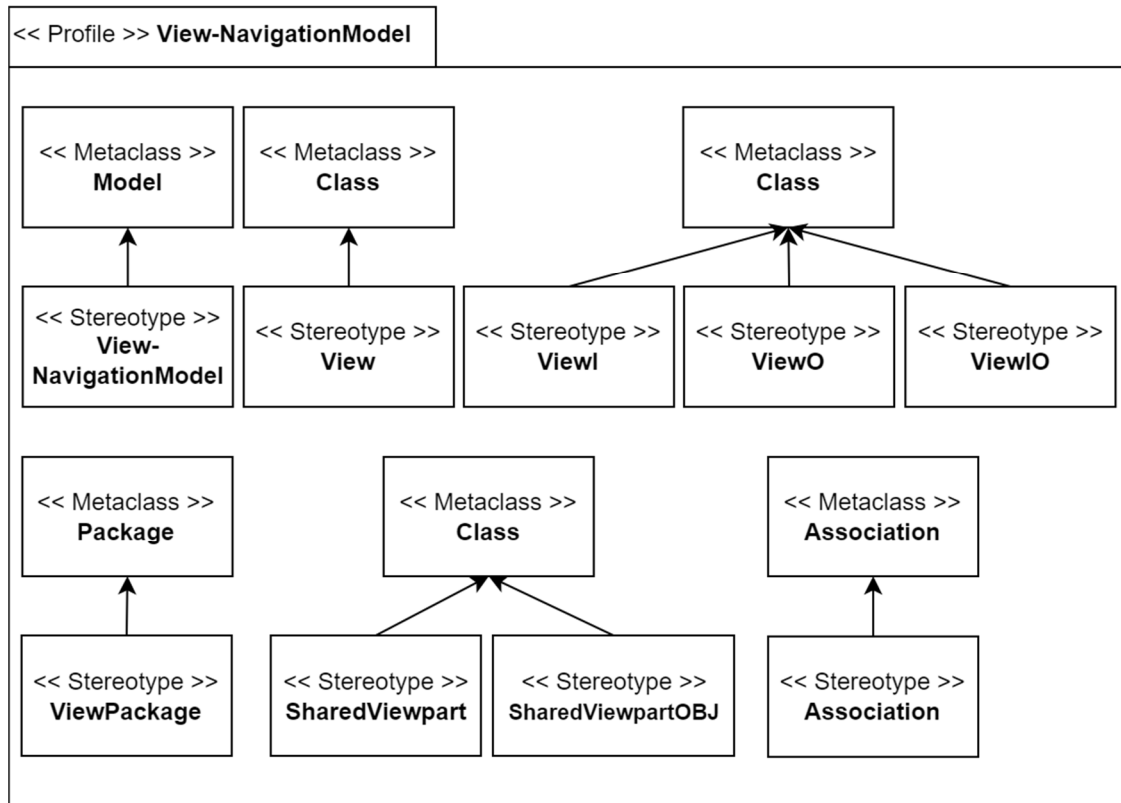


Figure 6.31 UML profile: View-Navigation model

The justifications for the *View-Navigation* model's elements are given in the next section under the *View* model's profile.

### 6.2.2. View Model

The requirements for the RiWAsML's *View* model to design a single view of a RiWA are set in Section 4.4.4.1. As mentioned in Section 6.1.1, the RiWAsML's intention is to capture and model the *View* elements required to implement the functionalities of the system and how they interact with the components of the system; therefore, the aesthetic details, like calligraphy, colours, media, and placements are not included in the *View* model.

- **UWE's *Presentation model*** [142] provides tools to design GUIs, including layout details. Notes can be added to the *Presentation model* to include information related to the functionalities of GUI elements; for example, a note on an update button may state that it shows the contact update page. However, the *Presentation model* does not capture the relationships of the GUI elements with the other components of the system to implement the functionalities; for example, a view's and its controller's relationship cannot be designed.
- **IFML** [124] diagrams capture more details related to implementing functionalities on views; however, since IFML does not model the low-level details of the actions, the IFML's view designs limit the scope for modelling the interaction flows. Further, the IFML designs lack GUI element details such as element type and name, which can help development.

Examples of *View* designing are given in Section 6.1.1 while introducing the *View* and related model-elements, and a more comprehensive example is provided in Section 8.3.2.3. The UML profile for the *View* model is given in Figure 6.32, based on the discussions in Section 6.1.1.

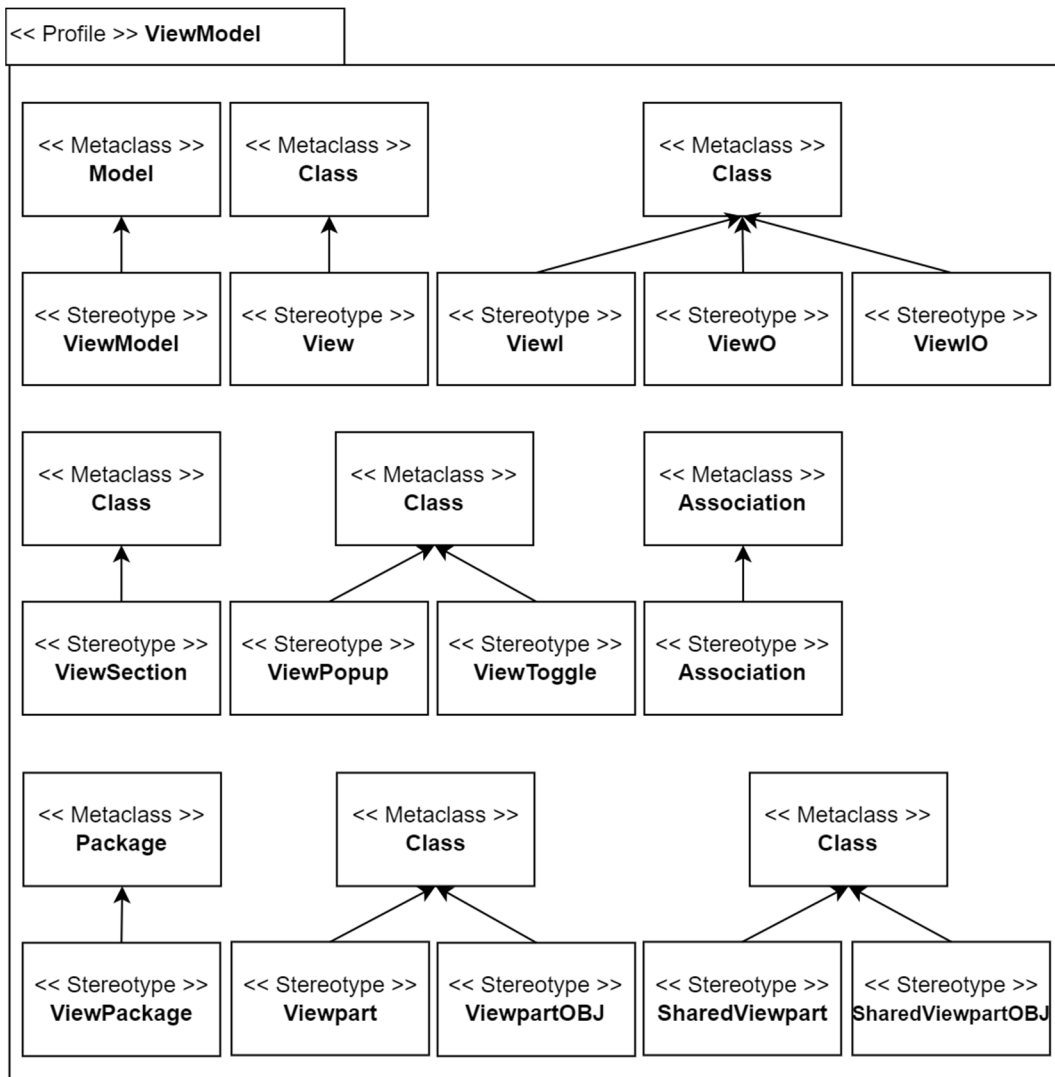


Figure 6.32 UML profile: View model

The GUI-elements extend the UML *Class* as they are abstract groups of elements. In the profile, they are grouped only for readability. The *ViewPackage* element extends the UML *Package* metaclass as its purpose is to package the GUI elements. The *Association* stereotype links the *ViewpartOBJ* and *SharedViewpartOBJ* elements with the corresponding *Viewpart* and *SharedViewpart* elements; it extends UML's *Association* metaclass.

### 6.2.3. AppControllers Model and ControllerClass Model

The requirements for the *AppControllers* and *ControllerClass* models are set in Section 4.4.4.2. The available solutions poorly address controllers and related features like events and DC handling, and none of them offer explicit models to design the RiWAs controllers and associated aspects discussed in Section 6.1.2. **IFML** [124] captures controller-related features such as events and actions;

however, they are mixed with the view’s details on the same diagram without providing enough details to support development.

The *AppControllers* model can be seen as a class diagram, which captures all the *ControllerClass* elements within a client-side *Application* element. The *AppControllers* model is a bridge to map the high-level controller component to the low-level *ControllerClass* elements. An example of an *AppControllers* diagram is shown in Figure 6.33.

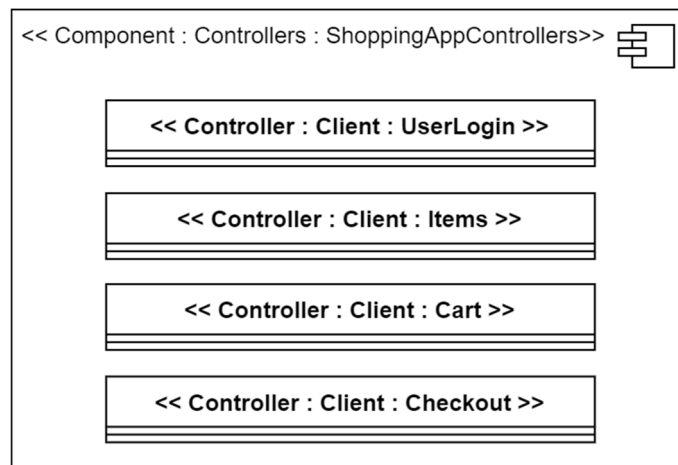


Figure 6.33 Example: *AppControllers* diagram

For the detailed design of a *ControllerClass* in an *AppController* diagram, the RiWAsML offers the *ControllerClass* model. The features to be satisfied by the model are set in Section 4.4.4.2. It is unnecessary to design all the *ControllerClass* elements in the *AppControllers* diagram; only the selected *ControllerClass* elements, requiring more low-level details, would be further designed using the *ControllerClass* model. Figure 6.34 provides the profile for the *AppControllers* model and the *ControllerClass* model. Since the *Component* element of the *Controllers* type is already included in the *L2 View-Process* model’s profile (see Figure 5.21 in Section 5.3.4), it is not included again in the *AppControllersModel* profile.

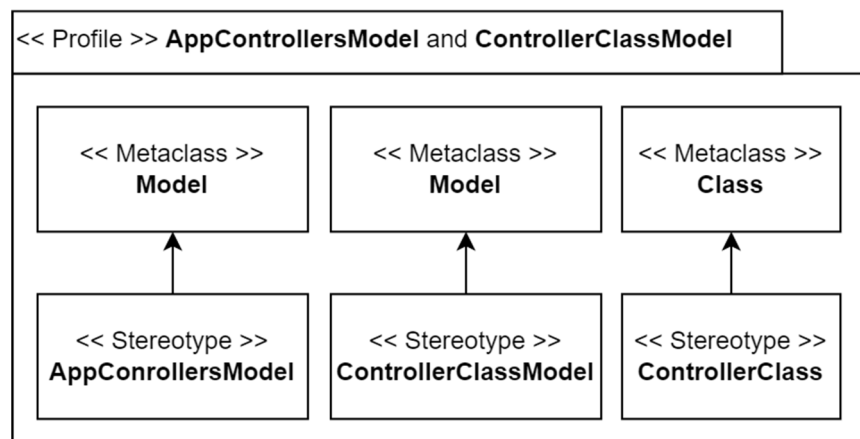


Figure 6.34 UML profile: *AppControllers* model, *ControllerClass* model, and their elements



### 6.2.4. AppModel Model and ModelClass Model

Section 4.4.4.3 sets the requirements for *AppModel* and *ModelClass* models. The UML provides the class diagram to design the business logic, which is commonly referred to as the domain model. UML or the available solutions do not attempt to capture the relationship between the *Application* element, *AppModel* element, and the *ControllerClass* elements as required by the RiWAsML.

The RiWAs *AppModel* model abstracts a particular *AppModel* component of a given *Application* element. An example *AppModel* diagram is shown in Figure 6.26 in Section 6.1.3, and a real-world example is provided in Section 8.3.2.5, including examples of *ModelClass* diagrams.

The profile for the *AppModel* model and *ModelClass* model is given in Figure 6.35.

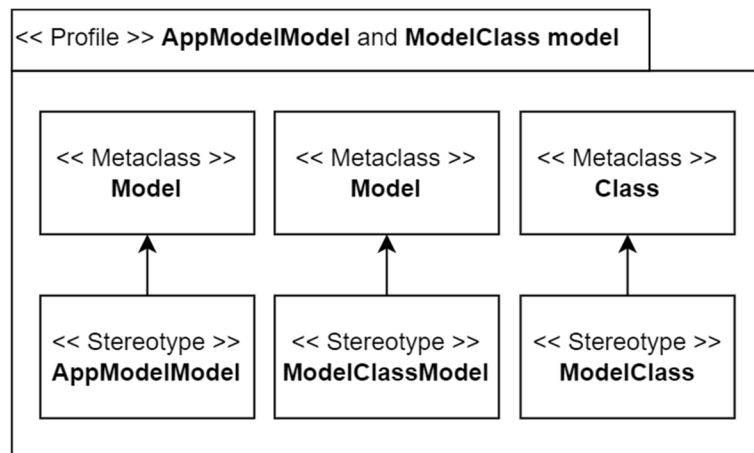


Figure 6.35 UML profile: *AppModel*

Since the *Component* element of the *model* type is already included in the *L2 View-Process* model's profile (see Figure 5.21 in Section 5.3.4), it is not included again in the *AppControllerModel* profile.

### 6.2.5. DC-bus Model and EndpointsCollection Model

The requirements for the *DC-bus* and *EndpointsCollection* models are set in Section 4.4.4.4. None of the available solutions explicitly provide models or model-elements to design the DB-bus and related aspects discussed in section 6.1.4.

RiWAsML offers the *DC-bus* model to capture the *EndpointsCollection* elements within a *Connector* element of *DC-Bus* type and their relationships. An example *DC-bus* diagram is given on the left side of Figure 6.36. The *EndpointsCollection* elements in a connector may or may not have relationships with each other. The RiWAsML recommends following the ODD practices to arrange the endpoints into *EndpointsCollection* elements and identify if they have relationships. Figure 8.15 in Section 8.3.2.4. provides a *DC-bus* diagram comprising the *EndpointsCollection* elements with relationships.

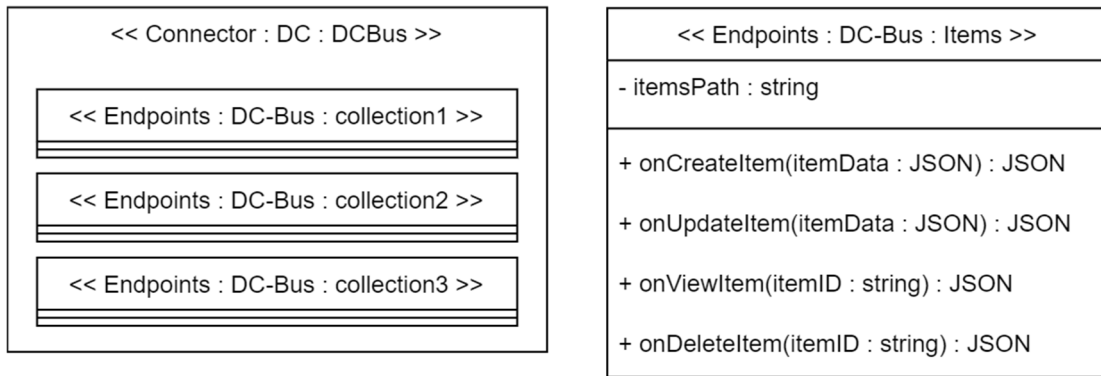


Figure 6.36 Example: DC-bus diagram (on the left) and EndpointsCollection diagram (on the right)

The *EndpointsCollection* model helps design the lower-level details of an individual *EndpointsCollection* element in a *DC-bus* diagram. An example is illustrated on the right in Figure 6.36. Other elements' communication with the endpoints can be depicted on the *EndpointsCollection* model as discussed in Sections 6.1.4.2 and 6.1.4.3. An example is given in Figure 8.17 in Section 8.3.2.5.

The UML profile for these models and their elements are given in Figure 6.37.

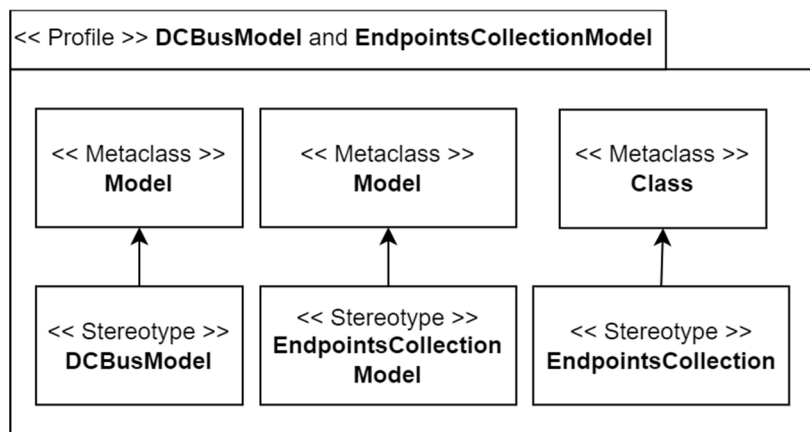


Figure 6.37 UML profile: DC-bus model and EndpointsCollection model

Since the *Connector* element is already included in the *L2 View-Process* model's profile (see Figure 5.21 in Section 5.3.4), it is not included again for the *DCBusModel* profile.

### 6.2.6. View-Controller Model

Section 4.4.4.5 sets the requirements for *View-Controller* model. **IFML** [124] models the relationship between the view and the event handlers. Yet, it lacks the controller concept and how the view and its event handlers interact with the rest of the system since the primary purpose of IFML is to capture the interaction flows.

Section 6.1.2. discusses much of the communication between the view and its controller. Section 8.3.2.3. gives an advanced *View-Controller* diagram from a real-world use case. Figure 6.38 provides the UML profile for the *View-Controller* model.

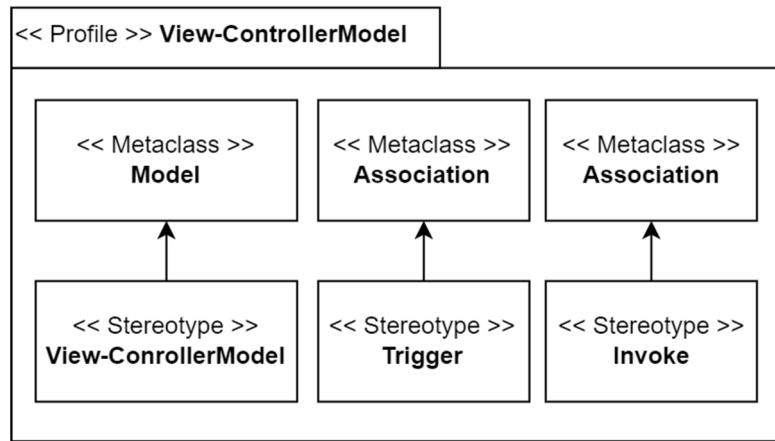


Figure 6.38 UML profile: View-Controller model

The model-elements related to the views and controllers are already included in the profiles of the *View* model and *ControllerClass* model. In addition, the *View-Controller* model requires special communication channels to denote the view triggering a controller’s event handler and a controller invoking the popups/toggles of a view, which are discussed in Sections 6.1.2.2 and 6.1.2.5. These communication channels are extended from the UML association, aligning with the RiWAsML’s other communication channels.

### 6.2.7. View-Process Sequence Model

Requirements for the *View-Process Sequence* model are set in Section 4.4.4.6. The UML offers the sequence diagram to model the execution flow of a function. UML-based solutions use the sequence diagram for the same purpose. The RiWAsML’s *View-Process Sequence* model integrates the UML sequence diagram, aligning with the RiWAsML’s other models and model-elements. An example diagram of designing a login process of a library system is given in Figure 6.39.

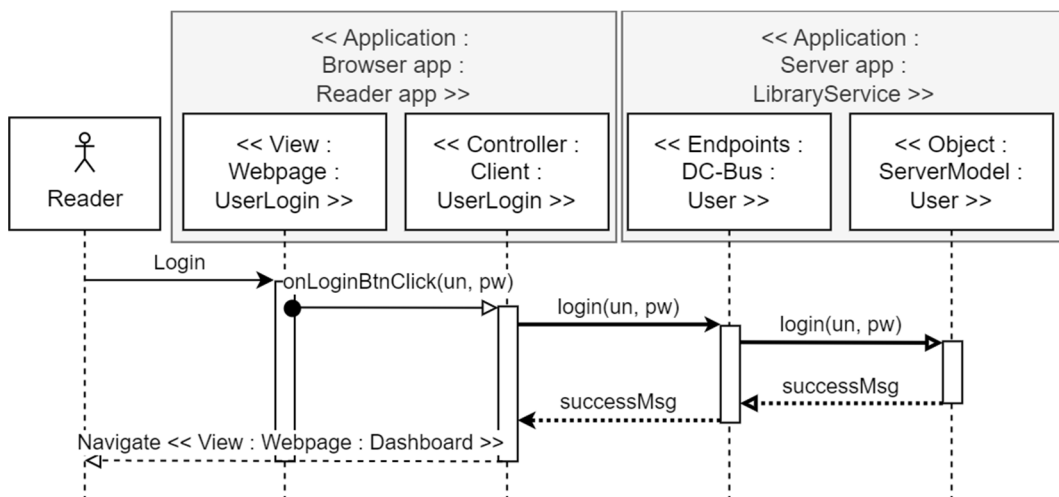


Figure 6.39 Example: View-Process Sequence diagram

The *View-Process Sequence* diagram uses the complete RiWAsML label on the lifelines. For view, controller, and DC-bus’s *EndpointsCollection*, the lifelines are the element classes, and for the

server-model, it's an object of the *User* class. *Application* elements wrap the lifelines to show which *Application* element contains the view and process elements. These *Application* elements should use the same labels as those mentioned in the high-level design.

The RiWAsML communication channels are utilised as follows for the communication between the lifelines.

- The view to the controller communication channel is illustrated as an event-based communication channel by indicating the event triggered on the view with a black circle at the beginning of the arrow. The event handler *onLoginBtnClick()* call is given parameters username *un* and password *pw*, implying the controller is reading the values from the view.
- The communication channels between the controller and the DC-bus's *EndpointsCollection* are illustrated with thick lines representing DC.
- The server-model to DC-bus and DC-bus to controller return arrows are labelled with the return value to understand the flow control's status.
- The controller's return arrow continues to the actor through the view's lifeline with the label "Navigate" to denote that the controller redirects the actor to the view specified on the label (*Dashboard* web page) on the actor's successful login.

Figure 6.40 provides the UML profile for the *View-Process Sequence* model.

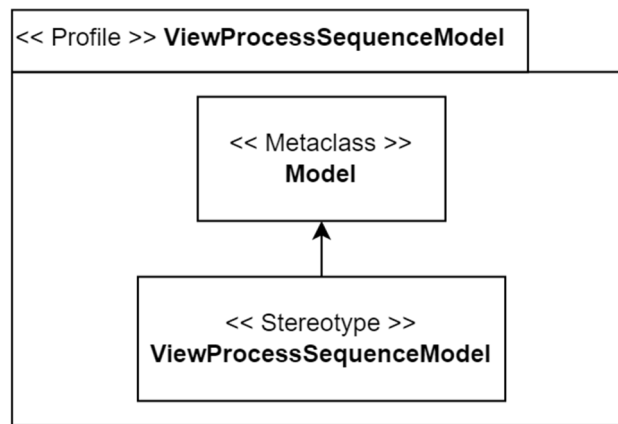


Figure 6.40 UML profile: *View-Process Sequence* model

Model-elements required for this model are already included in the profiles of other models; therefore, the profile for the *View-Process Sequence* model contains only the stereotype for the *ViewProcessSequenceModel*.

### 6.3. Chapter Summary

This chapter introduces the following models, their model-elements, and UML profiles.

- **View-Navigation model and View model:** View, GUI elements (Input/output elements, Containers/sections, Popups/toggles, Viewpart, ViewpartOBJ, ActorView, ViewPackage,

SharedViewpart, and SharedViewpartOBJ), and Navigation (linked-based navigation, process-based navigation).

- ***AppControllers model and ControllerClass model***: ControllerClass
- ***AppModel model and ModelClass model***: ModelClass
- ***DC-bus model and EndpointsCollection model***: EndpointsCollection
- ***View-Controller model***: Invoke and Trigger
- ***View-Process Sequence Model***

Together with the results of Chapter 5, this chapter fulfils the research objective 2.

## Chapter 7. Rich Web-based Applications Design Methodology

---

This chapter introduces the proposed Rich Web-based Application Design Methodology (RiWAsDM). Section 7.1 presents the elements of the RiWAsDM, and the following sections discuss these elements in detail. Refer to Section 3.2.3 for the review of available software designing methodologies. The introduction of the RiWAsDM achieves the research's objective 3 (refer to Section 1.4), aligning with step 3 (refer to Section 1.5.3.3) of the RiWAsDM implementing process (refer to Figure 1.4 in Section 1.5.3).

### 7.1. Introduction to the RiWAsDM

The introduction of a DSML would not be sufficient unless processes, rules, and guidelines for efficient utilisation of the DSML are provided. Based on this notion, this thesis introduces the RiWAsDM to gain the maximum benefits out of the RiWAsML. The RiWAsDM discusses the architecture of the RiWAsML and offers rules and guidelines for its accurate utilisation; the RiWAsDM also provides a design process to govern the RiWAs designing activities within RiWAs engineering. MDA, Arc42, ArchiMate, and SysML are identified as available design methodologies; they are reviewed in Sections 3.2.3 and 3.3.3 and evaluated against the RiWAsDM in Section 9.2.

The modules of the RiWAsDM and their focused attributes (refer to Section 4.1) are given in the following list.

1. **The RiWAsML:** This modelling language is the core of the RiWAsDM
  - 1.1. **RiWAsML's architecture:** the architecture of the RiWAsML helps understand the DSML, exhibiting the RiWAsML's simplicity, comprehensiveness, and usability.
  - 1.2. **RiWAsML reference:** A complete DSML reference is given to assist in the DSML's usability.
  - 1.3. **A set of rules and guidelines for the RiWAsML-based design and development:** these rules and guidelines improve the RiWAsML's usability and development support.
2. **The RiWAsDM process:** This process intends to satisfy the usability and integrability of the RiWAsDM by discussing the following.
  - 1.1. **Design approach:** discusses the RiWAsML's support for the design approaches.
  - 1.2. **Engineering approach:** discusses the RiWAsML's support for the engineering approaches.
  - 1.3. **Guidelines for RiWAsML-based AMDD:** discusses adopting RiWAsML-based designing into *Agile Model-Driven Development (AMDD)*.

The following sections discuss these modules of the RiWAsDM in detail. The evaluation chapter discusses how these RiWAsDM modules satisfy the attributes stated in the above list.

## 7.2. RiWAsML Architecture, Language Reference, and Rules and Guidelines

This section first discusses the RiWAsML's architecture, then provides a complete reference to the RiWAsML models and model-elements introduced in Chapters 5 and 6, and finally, offers a list of rules and guidelines to be followed when using the RiWAsML.

### 7.2.1. RiWAsML Architecture

The architecture of a language helps realise the language towards accurate utilisation of it. The RiWAsML architecture is given in Figure 7.1. This architecture is based on the RiWAArch style [12], reviewed in Section 3.1.3. The RiWAArch style provides the foundation for the RiWAsML, enriching the simplicity, which is further discussed in section 9.1.11, under evaluation.

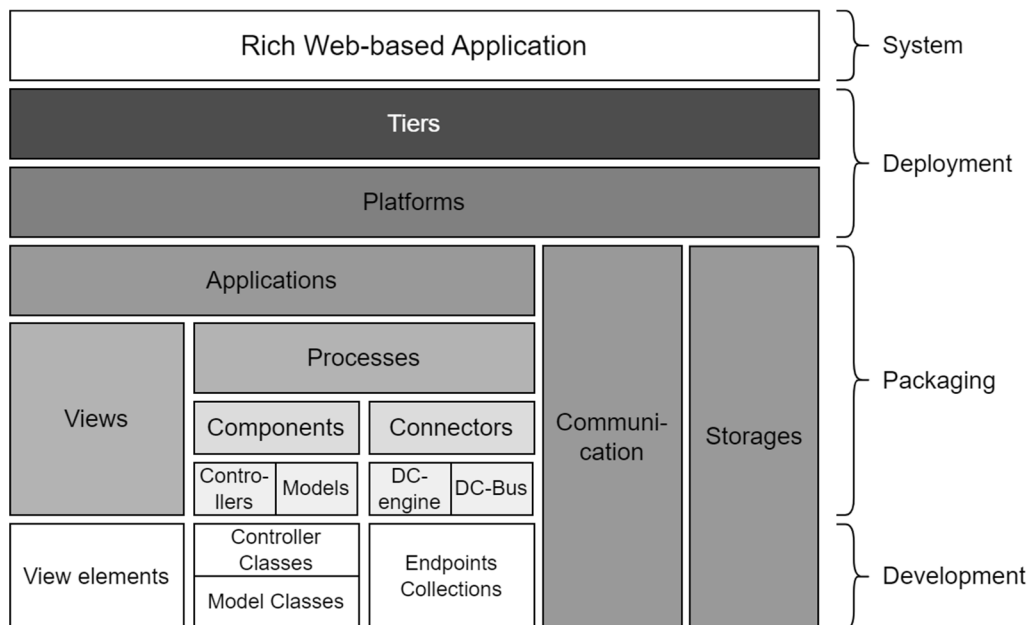


Figure 7.1 RiWAsML architecture

The RiWAsML architecture consists of 4 horizontal layers: system, deployment, packaging, and development, which are elaborated below.

1. **System Layer:** this layer sets the context to be modelled, which is RiWA.
2. **Deployment Layer:** the RiWAs comprise the tiers and platforms on the highest level; these platforms offer environments for the *Application* elements and storages to be deployed and run, where these platforms are organized into tiers by preserving simplicity at the uppermost level, improving the readability and understandability of the design. RiWAsML's *L1 Applications* model realises the deployment layer.
3. **Packaging Layer:** *Application* elements package their internal elements – views and processes. Processes encompass components and connectors where controllers and models are components, and DC-engine and DC-bus are connectors. The *Application* elements and their internal elements

communicate with each other using the RiWAsML’s communication channels. The *Level 2 View-Process* model realises the packaging layer and helps denote the view and process elements packaged by the *Application* elements in a RiWA.

4. **Development Layer:** RiWAsML provides models and model-elements to design the low-level aspects of the views, controllers, models, and DC-buses, which can assist in mapping these elements into development based on the OODD concepts and practices.

The packaging layer to the development layer mapping is given in Table 7.1.

Table 7.1 Models for package and single size of different types of elements

Size \ Element	View	Controller	Model	DC-bus
<b>Package</b>	View-Navigation model	AppControllers model	AppModel model	DC-bus model
<b>Single</b>	View model	ControllerClass model	ModelClass model	EndpointsCollection model

The mapping in Table 7.1 is detailed below.

1. **View:** the *View-Navigation* model captures the views of a client-side *Application* element, and each view can be detailed using the *View* model.
2. **Controller:** the *AppControllers* model identifies the *ControllerClass* elements required for a client-side *Application* element, and individual *ControllerClass* element in it can be detailed using the *ControllerClass* model.
3. **Model:** the *AppModel* model realises the *ModelClass* elements of a client-model in a client-side *Application* element or server-model in a server-side *Application* element. Each *ModelClass* element within an *AppModel* diagram can be detailed using the *ModelClass* model.
4. **DC-Bus:** the *DC-Bus* model encompasses the *EndpointsCollection* elements of a DC-bus in a server-side *Application* element, and each *EndpointsCollection* element can be detailed using the *EndpointsCollection* model.

An example of this mapping is given in Figure 7.2 using a shopping RiWA use case. Each *Views*, *AppControllers*, *AppModel*, and *Connector* element shows a single example inner element.



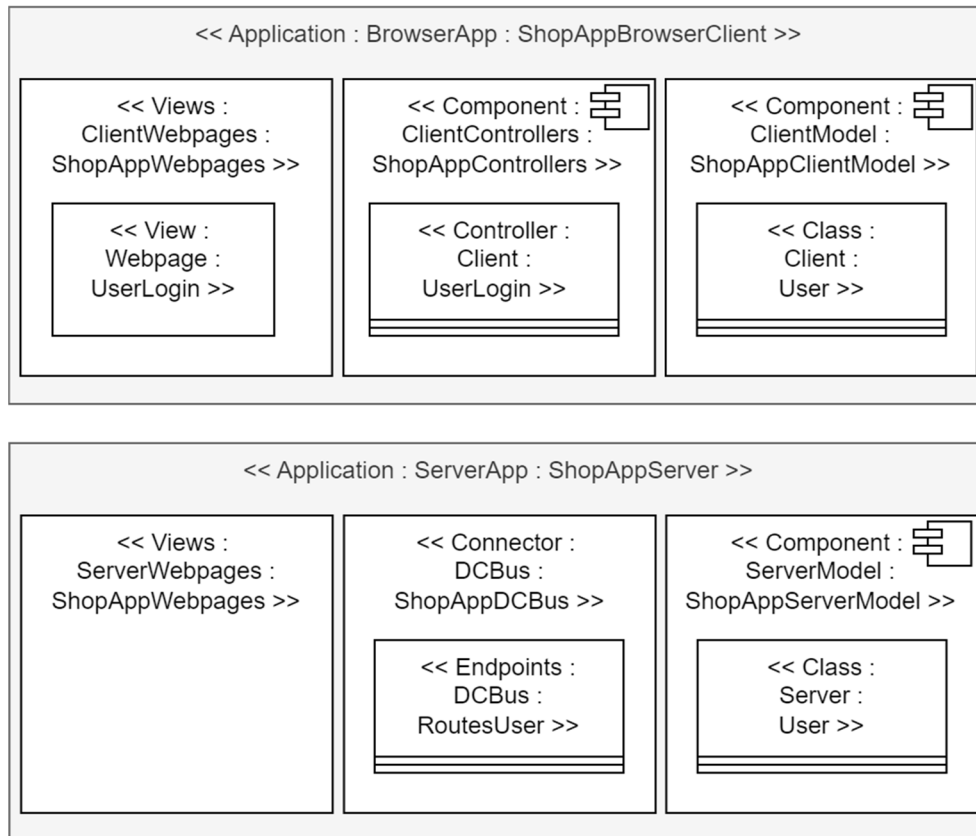


Figure 7.2 Example: mapping between the RiWAsML's Packaging layer and development layer

### 7.2.2. RiWAsML Language Reference

Figure 7.3 shows all of the RiWAsML's models under structural and behavioural categories; the models are referred to as "diagrams", considering it as the generic term. This section gives the RiWAsML complete language reference for all these models and their model-elements.

Under the structure diagrams, the *L1+2 Architecture* diagram is for RiWAs' high-level designing; for large and complex RiWAs, the architecture can be decomposed into two different diagrams in two levels: *Level 1 Applications* diagram and *Level 2 View-Process* diagram. All the other models are offered to design the low-level aspects of the RiWAs.

To design the execution flow of a task, the RiWAsML's version of the UML *Sequence diagram* is the *View-Process Sequence* diagram, which provides more details of the elements within the *Application* elements and their interactions to complete a specific task's execution.

The RiWAsML suggests drawing multiple diagrams together for inclined realisation; by default, the *View-Controller* diagram is given as the view and its controller are tightly coupled. Some beneficial and practical combinations are *View+ControllerClass+ModelClass* (clientModel) and *ControllerClass+EndpointsCollection+ModelClass* (serverModel). There is an example of an *EndpointsCollection* and a *ModelClass* in one diagram in Figure 8.17 in Section 8.3.2.5.

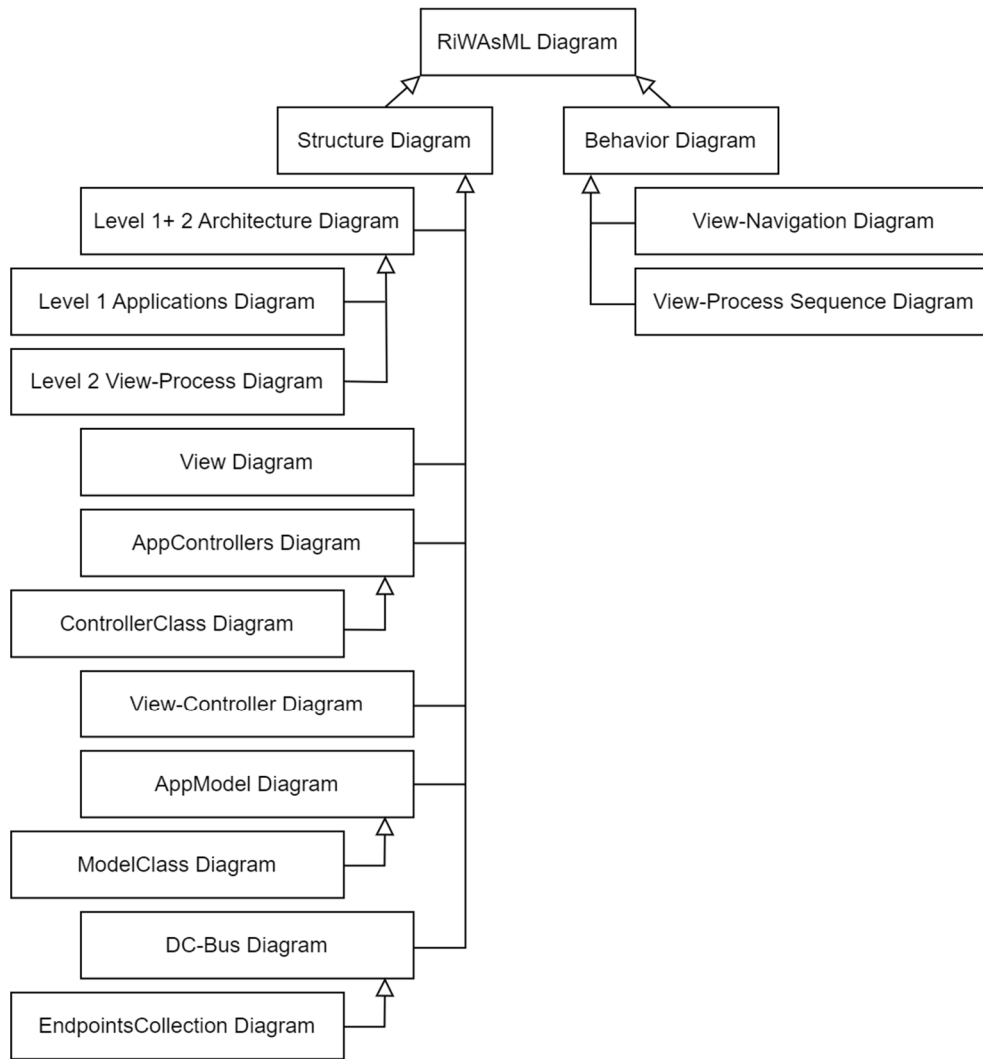


Figure 7.3 RiWAsML diagrams

**7.2.2.1. General Notations**

This section provides notations for the RiWAsML’s label and communication channels. The elements and their notations are given in Table 7.1.

Table 7.2 RiWAsML’s general elements and their notations

Element	Type	Notation
Label	-NA-	<< Element : Type : Name >>
Communication channels	Standard communication (HTTP and other standard protocols)	 Regular
		 Return (sequence diagram)

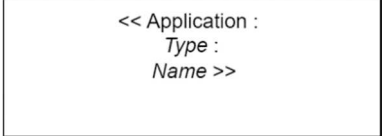
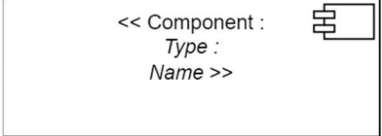
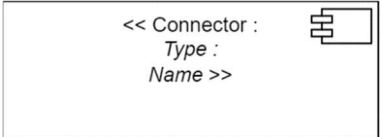

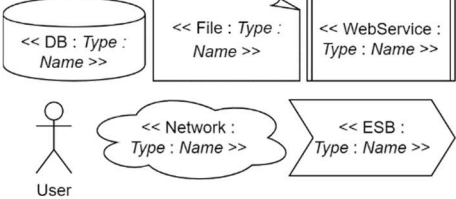
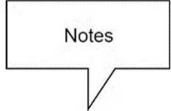
Element	Type	Notation
Communication channels	Delta-Communication	Regular
		Return (sequence diagram)
	Delta-Communication – push (reserved)	Regular
	View-Controller	Regular
		Return (sequence diagram)
	Method call and return	Regular
Return (sequence diagram)		
With numbered connectors	① ② ③	

**7.2.2.2. High-level Diagrams and Their Notations**

This section provides notations for the RiWAsML’s high-level diagrams and their elements in Table 7.3. For most of the elements, the types are specified by the RiWAsML, as given in Table 7.3.

*Table 7.3 RiWAsML’s high-level diagrams and their notations*

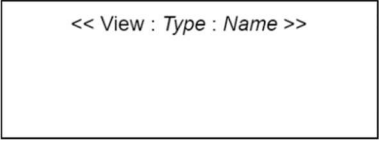
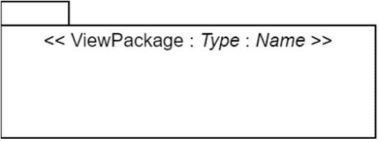
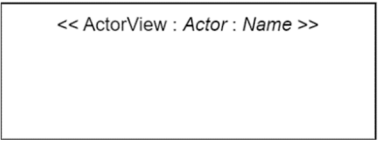
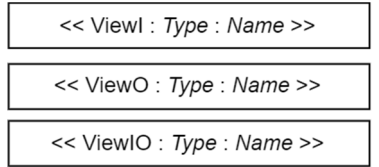
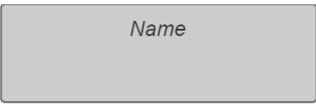
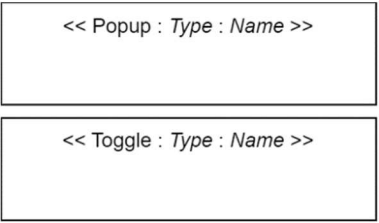

Element	Label (Element and Type segments)	Notation
<b>Level 1 Applications Diagram</b>		
<b>Tier</b> – horizontal grouping of elements based on their roles	<b>Element:</b> Tier <b>Type:</b> Presentation, Application, and Storage.	
<b>Platform</b> – provides the environment for an <i>Application</i> element to run.	<b>Element:</b> Platform <b>Level:</b> HW, OS, and App	

<p><b>Application</b> – An element executable within its platform, communicating with other Application elements in the system to perform functions.</p>	<p><b>Element:</b> Application  <b>Type:</b> WebApp, MobileApp, DektopApp, and ServerApp</p>	
<p><b>Level 2 View Process Model</b></p>		
<p><b>Component</b> – a logic processing element within an <i>Application</i> element.</p>	<p><b>Element:</b> Component  <b>Type:</b> Controllers, ClientModel, and ServerModel</p>	
<p><b>Connector</b> – a communication processing element within an <i>Application</i> element, enabling the <i>Application</i> element to communicate.</p>	<p><b>Element:</b> Connector  <b>Type:</b> DCEngine and DCBus</p>	
<p><b>Views</b> – element containing a collection of views of a client-side <i>Application</i> element.</p>	<p><b>Element:</b> Views  <b>Type:</b> WebPages, Activities, and Windows</p>	
<p><b>Other high-level elements</b> – High-level elements commonly used in RiWAs</p>	<p><b>Element:</b> DB, File, Webservice, User/Actor Network, ESB</p>	
<p><b>Notes</b> – element to add related text-based details.</p>	<p>-NA-</p>	

### 7.2.2.3. Low-level Diagrams and Their Notations

This section provides notations for the RiWAsML’s low-level diagrams and their elements in Table 7.4. For most of the elements, the types are specified by the RiWAsML, as given in Table 7.4.

Table 7.4 RiWAsML’s low-level diagrams and their notations

Element	Label (Element and Type segments)	Notation
<b>View Model, View-Navigation Model</b>		
<b>View</b> – a single GUI implementation	<b>Element:</b> View <b>Type:</b> WebPage, Activity, and Window	
<b>ViewPackage</b> – an element that wraps a GUI implementation and its supporting elements.	<b>Element:</b> ViewPackage <b>Type:</b> WebPage, Activity, and Window	
<b>ActorView</b> – A view implementing element containing elements of a particular actor.	<b>Element:</b> ActorView <b>Type/Actor:</b> The actor of the Viewpart, according to the scenario	
<b>Input/output elements</b> of GUIs/views.	<b>Element:</b> ViewI, ViewO, View I/O <b>Type:</b> Refer to Table 6.1	
<b>Container/Section</b> – a virtual section of a view.	<b>Element:</b> -NA- (identified by the grey, round cornered rectangle) <b>Type:</b> -NA-	
<b>Popup/Toggle</b> – a widget in a view which can be shown/hidden or enabled/disabled.	<b>Element:</b> Popup/Toggle <b>Type:</b> Blocking, NonBlocking, Show, Hidden, Enabled, and Disabled	
<b>Trigger communication channel</b> – connect the GUI element, which triggers an event to the event handler of the controller.	-NA-	

<p><b>Viewpart</b> – a GUI section for a particular actor.</p> <p><b>ViewpartOBJ</b> – an object of a Viewpart.</p>	<p><b>Element:</b> Viewpart, ViewpartOBJ  <b>Actor:</b> The actor of the Viewpart according to the scenario.</p>	<div style="border: 1px solid gray; padding: 5px; margin-bottom: 5px; background-color: #f0f0f0;">                 &lt;&lt; Viewpart : Actor : Name &gt;&gt;             </div> <div style="border: 1px solid gray; padding: 5px; background-color: #f0f0f0;">                 &lt;&lt; viewpartOBJ : Actor : Name &gt;&gt;             </div>
<p><b>SharedViewpart</b> – a Viewpart for a particular actor, shareable with multiple views.</p> <p><b>SharedViewpartOBJ</b> – a SharedViewpart object.</p>	<p><b>Element:</b> SharedViewpart, SharedViewpartOBJ  <b>Type:</b> Refer to table 6.1</p>	<div style="border: 1px solid gray; padding: 5px; margin-bottom: 5px; background-color: #f0f0f0;">                 &lt;&lt; SharedViewpart : Type : Name &gt;&gt;             </div> <div style="border: 1px solid gray; padding: 5px; background-color: #f0f0f0;">                 &lt;&lt; sharedViewpartOBJ : Type : Name &gt;&gt;             </div>
<b>AppControllers Model, ControllerClass Model, and View-Controller Model</b>		
<p><b>AppControllers</b> – wraps the <i>ControllerClass</i> elements of an <i>Application</i> element.</p>	<p>Same as the High-level <i>Component</i> element of the Controller type</p>	<p>Same as the High-level <i>Component</i> element of the Controller type</p>
<p><b>ControllerClass</b> – a class which implements event handling and related methods for a particular view.</p>	<p><b>Element:</b> Controller  <b>Type:</b> Client  <b>NOTE:</b> The name should be the same as the related view’s name</p>	<div style="border: 1px solid gray; padding: 5px;">                 &lt;&lt; Controller : Type : ViewName &gt;&gt;                 <hr/>                 - propName: type                 <hr/>                 + methodName(params): type             </div>
<p><b>Invoke</b> – indicate a controller’s method invoking a popup/toggle.</p>	<p><b>Type:</b> Show/hide/enable/disable</p>	<< Type >> →
<b>AppModel Model, and ModelClass model</b>		
<p><b>AppModel</b> – wraps the <i>ModelClass</i> elements of an <i>Application</i> element.</p>	<p>Same as the High-level <i>Component</i> element of the Model type</p>	<p>Same as the High-level <i>Component</i> element of the Model type</p>
<p><b>ModelClass</b> – a class that implements domain logic for an <i>Application</i> element.</p>	<p><b>Element:</b> Class  <b>Type:</b> ClientModel, ServerModel</p>	<div style="border: 1px solid gray; padding: 5px;">                 &lt;&lt; Class : Type : ClassName &gt;&gt;                 <hr/>                 - propName : type                 <hr/>                 + methodName(params): type             </div>
<b>DC-bus Model and EndpointsCollection Model</b>		
<p>DC-Bus – wraps the <i>EndpointsCollection</i> elements of an <i>Application</i> element.</p>	<p>Same as the High-level <i>Connector</i> element of the DC-Bus type</p>	<p>Same as the High-level <i>Connector</i> element of the DC-Bus type</p>
<p><b>EndpointsCollection</b> – a class that implements communication APIs of a server <i>Application</i> element.</p>	<p><b>Element:</b> EndpointsCollection  <b>Type:</b> DCBus and HTTPBus</p>	<div style="border: 1px solid gray; padding: 5px;">                 &lt;&lt; Endpoints : Type : Name &gt;&gt;                 <hr/>                 - propName : type                 <hr/>                 + endpointName(params) : type             </div>

### 7.2.3. Rules and Guidelines for Designing RiWAs with RiWAsML

This section offers the rules and guidelines to follow when using the RiWAsML. Most rules and guidelines are already discussed in related sections, while the RiWAsML models and model-elements are introduced. Those rules and guidelines are listed in this section, with further elaborations as required for clarity and reference. The rules are given as “Rule:” to be followed when using the RiWAsML, and the guidelines are mentioned as “Guideline:” to be considered best practice. This section performs Step 3.2 of the RiWAsDM implementing process (refer to Figure 1.4 in Section 1.5.3).

#### 7.2.3.1. Rules and Guidelines for General Elements of RiWAsML

This section sets the rules and guidelines for the RiWAsML’s *Label* and communication channels. Further, some overall guidelines are given in the direction of improving the simplicity and readability of the design.

##### Label

**Rule:** The RiWAsML model-elements should use the RiWAsML label format.

**Rule:** The RiWAsML model-elements should use the specified values for the element segment of the label.

**Rule:** The RiWAsML model-elements should use the specified values for the type segment of the label when the element type is exact.

**Guideline:** When the element is of a type the RiWAsML does not specify, a suitable custom type could be assigned.

**Guideline:** RiWAsML suggests using the following cases for the values of the element, type, and name segments of the RiWAsML’s model-elements to align with OODD practices.

- Use the Pascal case for naming, in general.
- Use the Camel case to name objects such as *ViewpartOBJ* and *SharedViewpartOBJ*, as well as method names and attributes.

##### Communication Channels

**Guideline:** RiWAsML realises the communication between the architectural elements based on the request-response model. Thus, the bidirectional arrows (or unidirectional arrow pairs in low-level models), which depict the communication channels, have a specific meaning, and the requesting and requested elements are known as follows.

- **Between view and controller:** The view requests the controller by triggering events and passing the data to the controller. The controller responds by processing the data and updating the view with information.
- **Between controller and client-model:** A controller calls a client-model's method, and the method processes the request and returns the result.
- **Between controller and DC-bus:** A controller sends a DC request to the DC-bus via the DC-engine. The DC-bus processes the request, and the results are sent using the DC response to the controller via the DC-engine.
- **Between DC-bus and server-model:** The DC-bus calls a server-model's method, and the method processes the request and returns the result.
- **Between other elements:** Standard communication protocols can be used to request services from external elements and receive the responses. For example, the server-model may request the database for CRUD operations and receive the results using TCP.

**Guideline:** It is possible to use unidirectional arrows when required. For example, when views and controllers of a browser-based client send HTTP requests to the server.

**Guideline:** A bi-directional arrow can still be used when required to denote communication between the methods of the same class or multiple classes. In controllers, event handlers always call the other methods; hence, the calling and the called methods are straightforward. For communication between methods other than the event handlers, use a black circle on the calling method to depict the caller, similar to the event handling view elements.

### General Guidelines

**Guideline:** the RiWAsML suggests using colours for the model-elements as necessary to improve simplicity and readability.

#### 7.2.3.2. Rules and Guidelines for High-level Designing with RiWAsML

This section specifies the rules and guidelines for the *Level 1 Applications* diagram, *Level 2 View-Process* diagram, *Level 1+2 Architecture* diagram, and their elements.

**Guideline:** The high-level aspects can be hierarchically designed using the separated *Level 1 Applications* diagram and *Level 2 View-Process* diagram or can be included in a single design using the *Level 1+2 Architecture* diagram depending on the size and the complexity of the RiWA. Priority may given to the readability of the design.

**Guideline:** In RiWAs with browser-based clients, the RiWAsML does not depict the case of the server sending views to the client as the response to the HTTP requests on the diagrams, considering it is a well-known fact.



**Guideline:** The *Tier* elements are not required to have the same height and width (see Figure 8.10 in Section 8.2.2).

**Guideline:** The *Platform* elements can expand across the tiers as required (see Figure 8.3 in Section 8.1.1).

**Guideline:** On high-level diagrams, the *Views* and *AppControllers* elements always use plural form for the label's element, type, and name segments.

### 7.2.3.3. Rules and Guidelines for Low-level Designing with RiWAsML

This section specifies the rules and guidelines for the low-level design diagrams and their elements. Additionally, some development-supportive rules or guidelines are given as “Rule [development]:” or “Guideline [development]:”.

#### **View Diagram**

**Guideline:** A *View* diagram may not contain all the GUI elements and may show only the GUI elements required to implement functionalities.

**Guideline:** A *View* diagram is not required to provide the actual layout.

**Guideline:** It is suggested that a view be designed with its dedicated controller as a pair on the same diagram as a *View-Controller* diagram.

**Guideline [development]:** Use a GUI element's type and name label segment values to derive its development name. For example, consider the element with the label `<< ViewI : btn : Delete >>`; in that case, use `btnDelete` as the development name of it. This value can be used for both the ID and Name attributes of the GUI elements of webpages.

**Guideline:** Use *Viewparts* to improve the readability of the design; they may or may not be used in actual development.

**Guideline [development]:** For a *ViewPackage* with *ActorViews*, the view's actual development name is the name of the *ViewPackage*. Use the names of the *ActorViews* as the display names of the view for the actors of each *ActorView*.

**Rule [development]:** For *Popups*, use the label's type segment to specify the behaviour of the *Popup* as “Blocking” or “NonBlocking”. For *Toggles*, use the label's type segment to specify the initial state as Show/Hidden/Enabled/Disabled.

**Guideline [development]:** Do not mix the JavaScript code with the view's HTML code for browser-based client apps. If the controller has less code and decides to write it on the HTML file, always write the JS code in the head's script section without mixing it with HTML code anywhere else in the document. However, on the design, it should be given as a separate controller.

**Guideline:** Each view should be available on the *View-Navigation* diagram. There cannot be any views not included in the *View-Navigation* diagram.

### **View-Navigation Diagram**

**Guideline [development]:** The primary purpose of the *View-Navigation* diagram is to identify the views that can implement multiple sets of related features and denote the different routes to navigate for different actors (refer to Section 4.4.4.1). It is crucial to identify the multiple features developed in each view, the actors of these features, and the various navigation paths to these views, and then design them using the *View-Navigation* diagram to assist in realising the arrangement of all the views in a RiWA towards reducing the complexity of the development.

**Guideline [development]:** The *View-Navigation* diagram captures all the views to be developed in the RiWAs and the navigation between them (refer to section 4.4.4.1). There cannot be any diagram of view that is not included in the *View-Navigation* diagram. However, it is sufficient to design only the required views using the *View diagram*, and it is not necessary to design all the views available on the *View-Navigation* diagram using *View* diagrams.

**Guideline [development]:** Navigation links starting from hyperlink GUI elements are always link-based navigation and developed as standard hyperlinks on the views. Navigation links starting from other types of GUI elements are process-based and should be implemented in the view's controller.

### **AppControllers Diagram and ControllerClass Diagram**

**Guideline:** It is not mandatory for all the views to have a dedicated controller; for example, views with only readable content, like a help page, do not necessarily need a controller. Therefore, the number of controllers in the *AppControllers* diagram can be less than the number of views in the *View-Navigation* diagram. It is practical to identify which views require a controller and include them on the *AppControllers* diagram.

**Guideline:** It is beneficial to design a *ControllerClass* with its related view on the same design using the *View-Controller* diagram.

**Guideline:** Decide the rich features to be developed on the view, identify all the event handlers required on the controller to implement them, and include them on the design with notes explaining their functionalities.

**Rule:** Even though the parent *AppControllers* element of the *AppControllers* diagram is a component, the parent *AppControllers* element or the *ControllerClass* elements in it should not use the UML component's standard interfaces; instead, they should always use the RiWAsML communication channels.

**Guideline [development]:** Do not mix the event handling code with the view’s code; always write the event handlers’ registering code on the controller. Examples of improper vs proper coding style suggested by the RiWAsML are given in Table 7.5.

Table 7.5 RiWAsML’s recommended events handling coding style

	Improper code style	Proper code style
<b>View – HTML</b>	<pre>&lt;button onclick="deleteItem()"&gt;     Delete &lt;/button&gt;</pre>	<pre>&lt;button id="btnDelete"&gt;     Delete &lt;/button&gt;</pre>
<b>Controller – JS/jQuery</b>	<pre>Function deletedItem() {     //delete item code }</pre>	<pre>\$("#btnDelete").on( "click", function() {     //delete item code } );</pre>

### **AppModel Diagram and ModelClass Diagram**

**Guideline:** Each *AppModel* diagram comprises a complete class diagram.

**Rule:** An *AppModel* diagram should use a parent *model component* element.

**Rule:** Even though the model is a type of *Component*, it does not use UML’s standard interfaces; instead, the model component and its classes should always use the RiWAsML communication channels to indicate the communication with the outside.

**Guideline [development]:** Even if the client-model is developed using a language like JavaScript without ODD practices, the *ModelClasse* elements could be developed in separate files as depicted in the design.

**Guideline [development]:** Some web server-side frameworks provide a concept named “model” for database *Object Relational Mapping/Model* (ORM) or *Object Data Model* (ODM) implementations [170] [171]. These ORM/ODM models are different from the RiWAArch style’s concept of model, which is based on the MVC. The ORM/ODM models can be considered a part of the RiWAArch style’s model, which assists in implementing the database-related functionalities within the RiWAArch style’s model.

### **DC-Bus Diagram and EndpointsCollection Diagram**

**Guideline:** Even though the RiWAsML considers the connector a type of component, using the component symbol on the connector elements is not mandatory to maintain the visual separation between the connectors and components.

**Guideline [development]:** A web service API development concept like SOAP [144] or REST [145] can be used to develop the DC-bus. Consider the relevant rules and guidelines of the selected API development technology when designing and developing DC-bus and *EndpointsCollections*.

**Guideline [development]:** Web server-side development frameworks typically use the term “controller” for the DC-bus-related aspects [139] and “route” for the endpoint [172] [173] [174], which are derived from MVC and REST. Be aware of these terms, and do not substitute the RiWAArch style’s and RiWAsML’s controller and endpoint concepts with them.

**Guideline [development]:** Multiple *EndpointsCollections* can be developed on the same class/script where necessary, maintaining the conceptual separation as denoted in the design.

**Guideline [development]:** Select a consistent data wrapping technique/technology like XML or JSON for the endpoints and denote that on the *DC-bus* diagram and/or *EndpointsCollection* diagram using notes. Also, the structure of the datasets can be included in the design using notes.

### **View-Process Sequence Diagram**

**Guideline:** Wrap the swimlanes of the same application using an *Application* element.

**Guideline:** The *Application* elements and their view, controller, connectors, and models should be aligned with the other design diagrams of the same RiWA.

### **Other Related Design Concerns**

**Guideline:** Use a suitable designing method/tool like Entity-Relationship diagrams (ER diagrams) to design the databases.

**Guideline:** Use a suitable designing method/tool like Flowcharts to design the algorithms inside methods of the controller, model, and *endpointsCollection* classes.

**Guideline:** The RiWAsML suggests drawing multiple diagrams together for improved realisation. Some beneficial and practical combinations are *View+ControllerClass+ModelClass* (clientModel) and *ControllerClass+EndpontosCollection+ModelClass* (serverModel). There is an example of an *EndpointsCollection* and a *ModelClass* in one diagram in Figure 8.17 in Section 8.3.2.5.

**Guideline:** Other standard UML diagrams, like *network architecture* diagram and *activity* diagram, shall be used as required. However, it is advisable to use the RiWAsML label on the UML elements in a suitable way that aligns with the rest of the RiWAsML diagrams.

## **7.3. The RiWAsDM Process**

This section discusses adopting the RiWAsML-based designing into the RiWAs engineering. These discussions align with step 3.1 of the RiWAsDM implementing process (refer to Figure 1.4 in Section 1.5.3).

### **7.3.1. RiWAs Design Approach with RiWAsDM**

Refer to Section 2.2.1.3 for software design approaches. RiWAsDM appreciates using the top-down design approach; the RiWAs share high-level common characteristics and essential features, which can be realised with the RiWAArch style (refer to Section 2.3.5); hence, it is easy to start from the high-level architecture and design from top to bottom.

Besides, it's not inappropriate to use the bottom-up approach when the low-level details of the features of the RiWA are precisely identified, and the engineering team members have some experience in RiWA engineering. In a case where the engineers have knowledge of the RiWAArch style, they can emphasize recognising the low-level elements and use the bottom-up approach based on the high-level realisation offered by the RiWAArch style.

### **7.3.2. RiWAs Engineering Approach with RiWAsDM**

Refer to Section 2.1.3 for software engineering approaches. RiWAsDM recommends using the *Agile Model-Driven Development* (AMDD) approach for RiWAs engineering to benefit from both MDSE and agile SE since design thinking is essential for improving the quality of software [30]. Available solutions like SysML [47] strongly appreciate using AMDD [175], which is pointed out in Section 9.5.15.

### **7.3.3. Guidelines for Adopting RiWAsML-based Designing into Agile Environments**

This section discusses adopting the RiWAs designing activities into RiWAs engineering, aligning with the top-down design approach and AMDD approach. RiWAsDM has selected three popular agile development life cycles: Scrum [60] [61], DevOps [3], and *Continuous Integration, Continuous Delivery* (CI-CD) [4] to discuss the integration of RiWAs designing activities.

Since the agile development methodologies are discussed in the literature [3] [4] [60] [61], this section's attention is given to providing guidelines to integrate the RiWAsML-based RiWAs designing activities into RiWAs engineering into these agile methodologies. The life cycles of these agile methodologies are briefly discussed to obtain an overview of them, focusing on the design activities. The scrum framework is given in Figure 7.4, the DevOps tools chain is shown in Figure 7.5, and the CI-CD life cycle is illustrated in Figure 7.6.

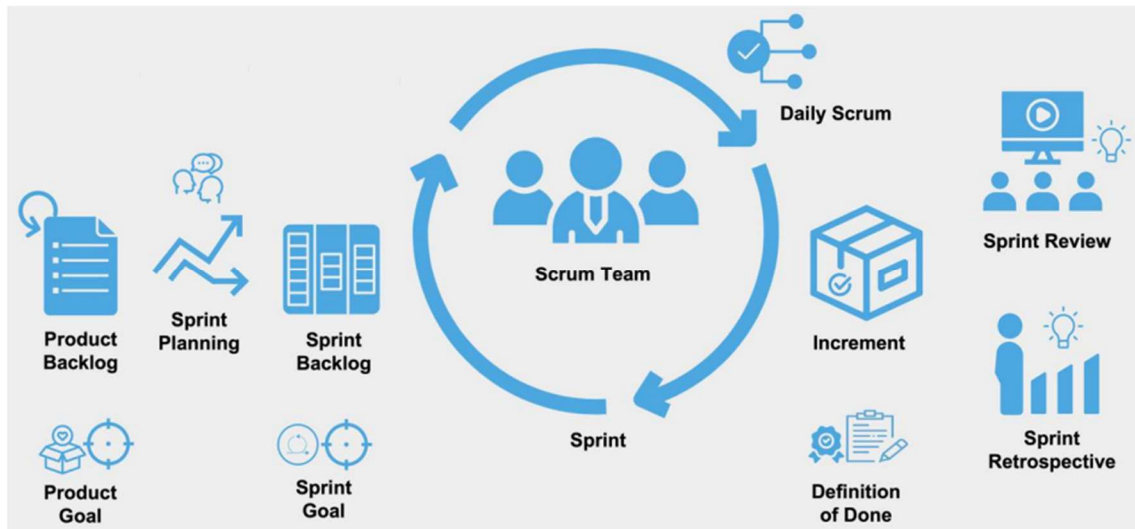


Figure 7.4 The scrum framework [176]

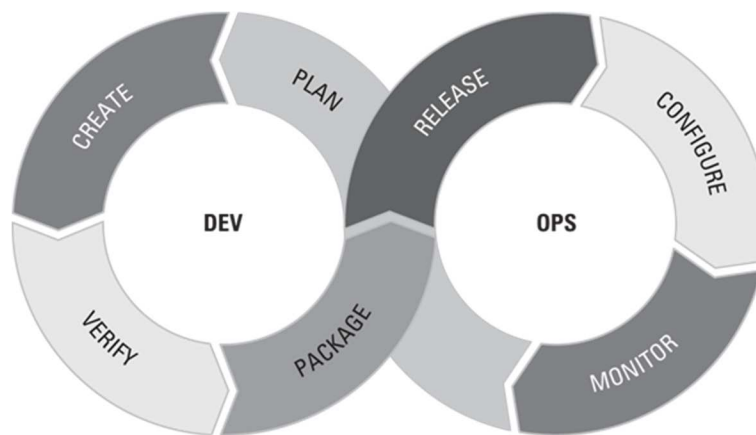


Figure 7.5 The DevOps tool chain [3]

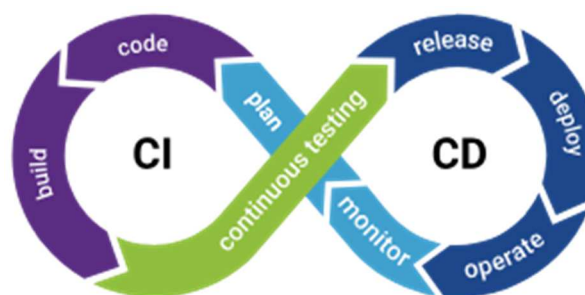


Figure 7.6 CI-CD life cycle [177]

A common feature of these agile-based engineering life cycles is that they develop systems incrementally and iteratively. An iteration usually takes a few days, and a planning stage is included in each iteration. RiWAsDM suggests exploiting this planning stage to integrate design activities. Some guidelines on integrating the RiWAsML-based designing activities are given below.

**Guideline:** Include a dedicated designer on the team, or the developers may also act in the designer role and design the features they develop.

**Guideline:** Perform basic requirements modelling and high-level design at the beginning of the engineering project, then revise them and do the amendments as required in each iteration throughout the project's life cycle.

**Guideline:** In each iteration, during the planning stage, design and document the selected functions to be developed and revise the design at the end of the iteration if required.

**Guideline:** At the end of each iteration, verify that the low-level design diagrams align with each other and the high-level design. Resolve any conflicts and apply them to the development before leaving the iteration.

**Guideline:** When design-related issues are identified in other phases like development, testing, deploying, or configuring, document them and keep them in a backlog to be addressed at both design and development levels in the following iterations.

## 7.4. Chapter Summary

This chapter introduces the RiWAsDM with the following modules.

1. The RiWAsML
  - 1.1. RiWAsML's architecture (Section 7.2.1)
  - 1.2. RiWAsML reference (Section 7.2.2)
  - 1.3. A set of rules and guidelines for the RiWAsML-based design and development (Section 7.2.3)
2. The RiWAsDM process (Section 7.3)
  - 2.1. Design approach (Section 7.3.1)
  - 2.2. Engineering approach (Section 7.3.2)
  - 2.3. Guidelines for RiWAsML-based AMDD (Section 7.3.3)

## Chapter 8. Use Cases

---

This chapter demonstrates the adoption of the RiWAsDM into RiWA engineering through real-world use cases in which I was an engineering team member. These use cases demonstrate the use of RiWAsML towards providing proof of concept, primarily focusing on the attributes expected from the RiWAsDM (refer to Section 4.1). Most diagrams are simplified to fit into A4 pages; thus, they may not contain all the details; however, they have enough details to demonstrate the features of the RiWAsML's models and model-elements. Larger versions of some diagrams are given in appendices as cited in relevant sections. The development aspects are not discussed in depth to maintain the length of this chapter. Demonstration of adopting the RiWAsDM through use cases fulfils the research's objective 4.

### 8.1. High-level Design Attributes of the RiWAsML

RiWAsML pays much attention to modelling the high-level structural aspects based on the RiWAArch style towards improving the realisation of the system through comprehensive architectural design. This section's use cases principally attempt to evidence how the RiWAsML improves diagrams' simplicity and usability (learnability, readability/understandability).

#### 8.1.1. Shopping System: Improve Simplicity and Readability with Tiers

This section focuses on discussing how the simplicity and readability of a RiWA architecture can be improved using the *Tier* elements. The architecture discussed in this section is of an online shopping system of an existing shopping chain. The system is already integrated with modern technologies to use AI, ML, and big data concepts for functionalities like analytics, customer classification, and recommendations towards improved sales. The recommendation system required ranking the recommending items, and my role was to research introducing a *Learning To Rank* (LTR) module [178], which was commenced by the end of 2018. I had to study LTR to identify the required parameters and implement a prototype using JAVA to integrate LTR into the available shopping system to rank the recommended products based on the customer's preferences.

My primary necessity was to understand the current system in order to integrate an LTR module, and I was given the system's architecture by the lead architect. A simplified version of the architecture is shown in Figure 8.1, and a larger figure of the same is given in Appendix B.1. The architecture is drawn using the informal boxe-and-line approach (refer to Section 3.2.1), and the *Application* elements and storage elements are scattered all over the diagram, which immensely reduces the simplicity and usability of the design. Due to the informal syntax and lack of simplicity, the architecture is less readable and substantially vague. Separate teams develop different modules of the system, and only the team leads have the proper architectural understanding of the elements they



work with. I had to consult all these team leads to obtain details of the high-level elements they were working with.

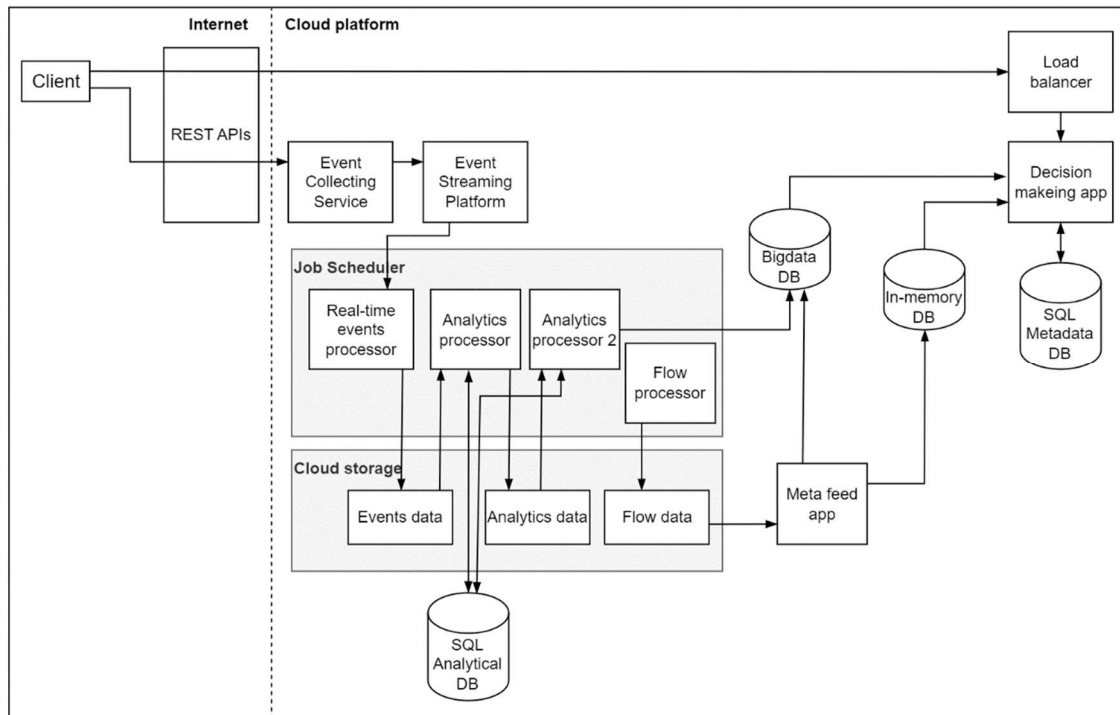


Figure 8.1 Use case: shopping system – original architecture

It was tedious to consult many individuals to learn diverse sections of architecture and combine the knowledge to realise the complete architecture in order to extend it to integrate an LTR module. This task could be much easier if the architecture is drawn using a formal language and all the necessary details are included. After implementing the RiWAsML, I selected this use case to test how the RiWAsML can improve this architectural design, focusing on simplicity and usability.

The significant issues of this architectural design are as follows. The dashed vertical line, which separates the architecture into two sections, looks like a tier separation, which is not. The left section is labelled as “Internet”, which is extremely vague and the right side is labelled as “Cloud platform”, which is a type of platform as the name implies. The “Cloud storage” box contains more boxes; their labels indicate that these elements represent data; however, the symbols used give an impression of components.

I first attempted to organize the architectural elements into tiers to improve simplicity by separating the *Application* and storage elements. The resulting architecture is given in Figure 8.2, and a larger version is provided in Appendix B.2. This version helps realise the type of the elements by their tiers distinctly; therefore, it’s more readable compared to the architecture without the tiers. The application tier contains all the *Application* elements, and the data tier holds the storage elements. Also, the client is wrapped into the presentation tier, further improving the simplicity of the application tier. It is



version of OS, and configurations, RiWAsML recommends using a *Note* element on the *Platform* elements.

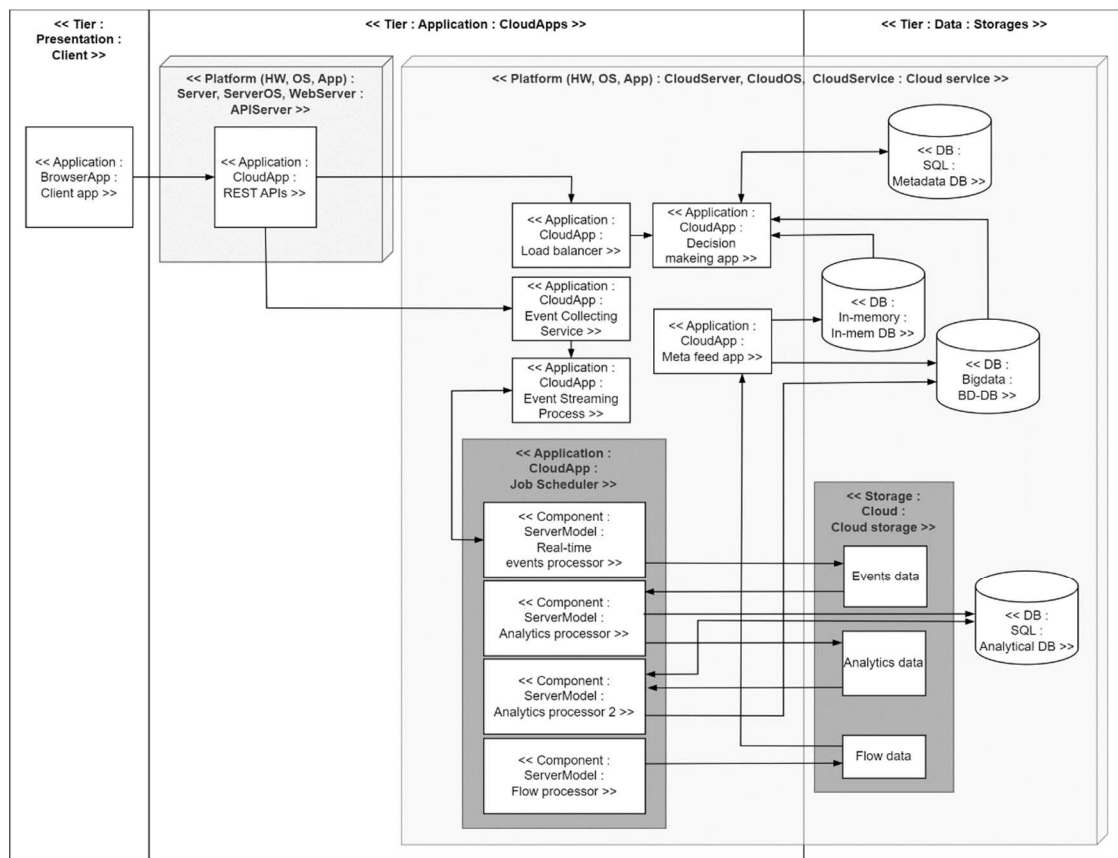


Figure 8.3 Use case: shopping system – Level 1 Applications diagram

The *Job Scheduler* element is considered an *Application* element, and its internal elements are treated as components. Even though the RiWAsML *L1 Applications* diagram does not contain components, they were kept on this diagram to demonstrate the flexibility of the RiWAsML.

This high-level diagram still lacks details and requires further improvement. For example, most of the arrows are uni-directional and do not denote the actual communication, hence provide a false impression. The arrow from the *Client app* to the *REST API* can be considered a request, and the response is missing. Arrows are given from the *REST API* to the *Load balancer* and the *Event Collecting Service* elements, but a return is not given to the *REST API* from any of the other elements. Arrows are coming to the *Analytical DB* denoting data income; however, data is not read from this database. The component named *Flow Processer* sends data to the *flow data storage*; nevertheless, this component is not connected to any other element. It is important to address all these concerns and produce a complete diagram for all the stakeholders to have a mutual realisation, eliminating ambiguities. This use case also emphasizes the importance of using a formal design language for producing usable design diagrams.

### 8.1.2. Book Club App: UML Node vs RiWAsML Platform

This use case utilises a *deployment diagram* of a *book club web application*, which is given as an example of the UML deployment diagram by *uml-diagrams.org* [179]. This use case attempts to explain how the RiWAsML *Platform* element improves the readability of the high-level diagrams by eliminating/reducing the nested elements required to denote the platform details compared to the UML *node* element.

The original UML example *deployment diagram*, which is taken from *uml-diagrams.org* [179], is given in Figure 8.4, which uses 3 nodes to specify the platform for the *book\_club\_app* and 2 nodes to set the platform for the database schemas.

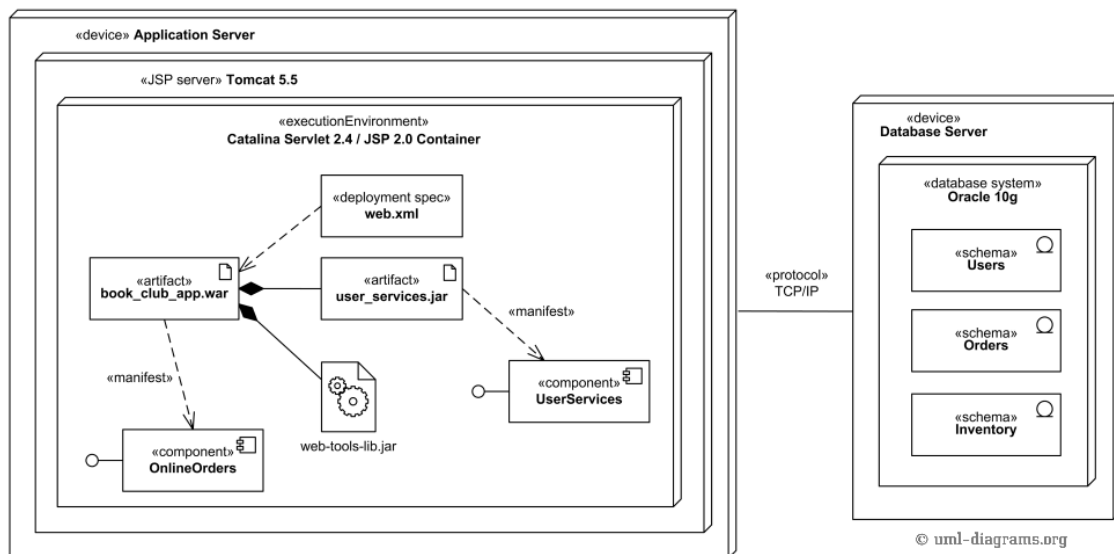


Figure 8.4 Use case: book club app – platforms designed using UML node elements [179]

Even though multiple nodes are utilised, the OSes are not depicted; more nodes are required to include the OSes in the design. Using multiple nested nodes to denote platforms makes the diagram less readable, mainly when the system contains many *Application* elements and databases running on dedicated platforms; for example, consider the architecture diagram with 4 *Application* elements in Figure 8.11 in section 8.3.1.

RiWAsML's *Platform* element is more abstract compared to the UML's *node* element and can denote the platform details of 3 levels using a single element. This feature facilitates keeping the diagram tidy and much readable. Consider the diagram given in Figure 8.5, which designs the platforms of the book club app and its database using RiWAsML's *Platform* element.

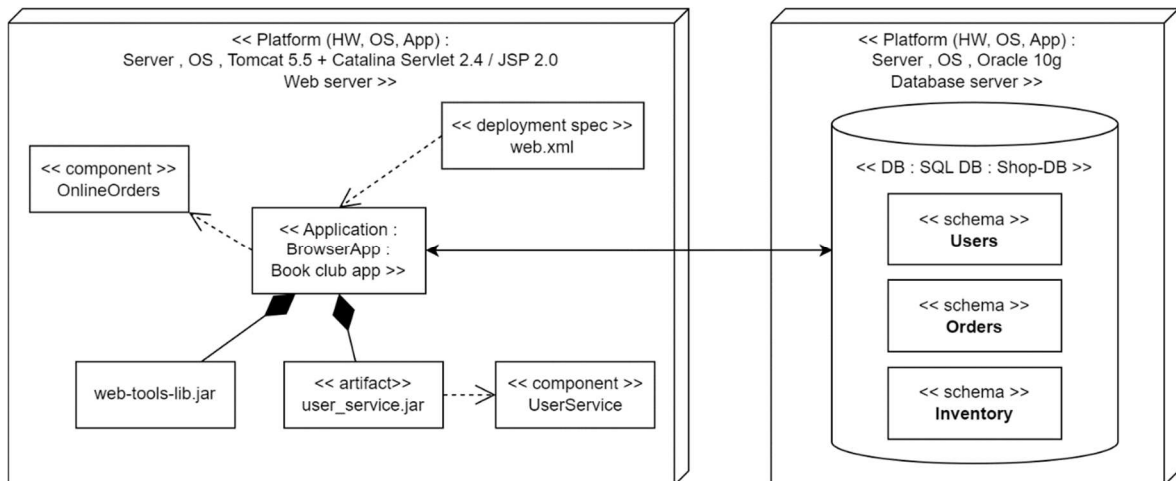


Figure 8.5 Use case: book club app – platforms designed with RiWAsML

The diagram in Figure 8.5 includes all the details of the diagram in Figure 8.4; however, the reduced number of elements diminishes the complexity of the diagram, improving its readability. Also, in Figure 8.5, the communication channel is drawn between the *Application* element and the database, which is semantically precise compared to Figure 8.4’s communication channel, which is depicted between the nodes.

RiWAsML’s focus is to capture the *Application* elements’ platform details, and the deployment details are not fully addressed as in UML; therefore, elements like *deployment specs* and *artefacts* are not provided by the RiWAsML. They are used in Figure 8.5 only to compare it with Figure 8.4. Anyhow, if these details are required to be included, RiWAsML advises using the UML model-elements or *Note* elements where necessary.

### 8.1.3. MiCADO-Edge – A Cloud-to-Edge Computing Architecture [180]

The MiCADO-Edge [180] is an architectural style that extends the cloud orchestration solution named MiCADOscale [181] to realise cloud-to-edge resource orchestration. The MiCADO-Edge research’s architectural diagrams are sketched using the box-and-line approach, and a legend is given to assist in understanding the different elements. However, even with the legend, the architecture is difficult to comprehend since it is incomplete, which leads to many questions discussed later in this section. A series of conversations were conducted with the research lead of the MiCADO-Edge research to verify the concerns. Then, I attempted to re-design the MiCADO-Edge architectural style [180] using the RiWAsML even though the context differs from the core RiWAs characteristics and features in this thesis. This activity aimed to verify that the RiWAsML can produce more readable and complete architectural diagrams using its formal syntax and the provided rules and guidelines. The original MiCADO-Edge architectural style [180] is given in Figure 8.6.

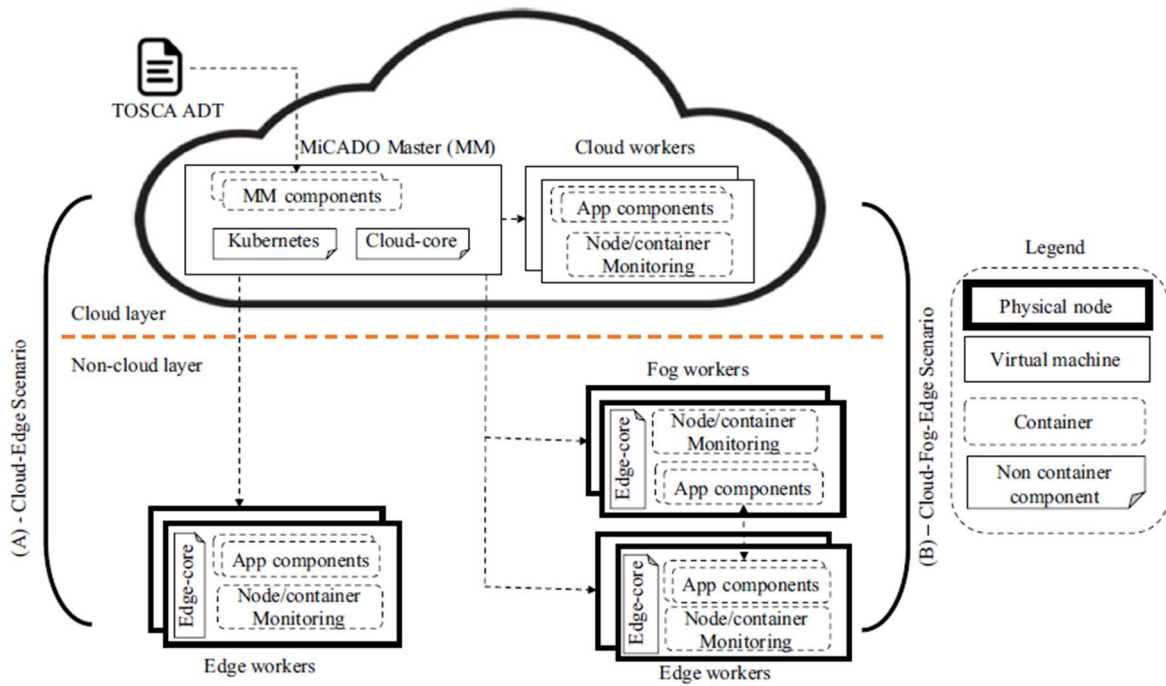


Figure 8.6 Use case: MiCADO-Edge – original architecture [180]

Preliminary questions, which require answers regarding this diagram are as follows.

- What is the type of the TOSCA ADT element?
- Do the container elements represent *Application* elements running in containers?
- Why are the non-container components, like *App components*, isolated without communicating with the other elements?
- With what elements do the non-container components communicate?

After studying the MiCADOscale and MiCADO-Edge architectures and finding answers to the above questions, combined with the knowledge obtained from the MiCADO-Edge lead researcher, I produced the following diagram in Figure 8.7 using the RiWAsML; a larger version is given in Appendix B.4. This diagram denotes the tiers and platforms of all the *Application* elements in the system; it can be seen as an *L1 Applications* diagram of the RiWAsML.

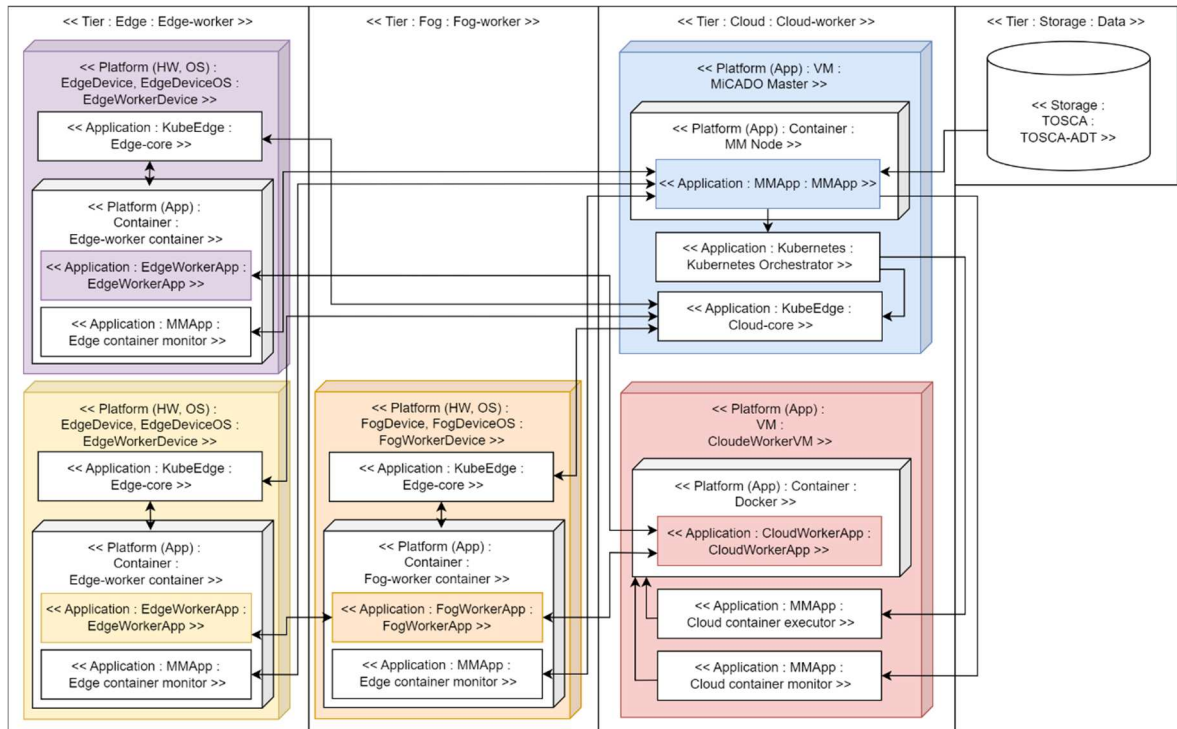


Figure 8.7 Use case: MiCADO-Edge – Architecture drawn using the RiWAsML

The diagram is compressed by reducing the spaces between the elements to ensure it fits into the page, which causes it to look compact. Anyhow, it answers the above questions and provides more details than the original diagram, making the architecture more readable and understandable. Some points to take into consideration are given below.

- TOSCA ADT element provides a description of infrastructure and policies. It is a configuration file written in YAML.
- Container platforms run *Application* elements in them.
- Non-container components in the original diagram are, in fact, *Application* elements, which are parts of the *KubeEdge* cloud orchestration infrastructure.
- *KubeEdge* elements communicate with each other to manage containers.

This diagram was sent to the MiCADO-Edge’s lead researcher, who verified the design’s accuracy and complimented the clarity. Since this is a complex diagram, providing separate *L2 View-Process* diagrams for the *Application* elements is recommended rather than producing an *L1+2 Architectural* diagram when it requires more high-level details. This use case is also a good example of a scenario for using separate *L1 Applications* diagram and *L2 View-Process* diagrams over a single *L1+2 Architectural* diagram.

## 8.2. Learning Management System

Smartest [182] is a learning management system (LMS) which uses a graph-based approach to present the learning resources to the students for a more resourceful navigational experience through the repositories in a clear, visual format that is easy to follow and understand. When I joined the Smartest project in March 2023, a working web application with all the core functions had been developed. The project's next step was to convert the web application into a RiWA with a web service in the back end. A team of two developers were working on the web service, and I joined as a front-end developer. My main task was to convert the web application into a RiWA, detaching the browser-based client application from its server application in order to communicate with the new web service. While trying to understand the system to apply the required conversion, I produced some design diagrams using the RiWAsML, which are discussed in this section and Section 8.3. The Smartest project largely contributed to improving the RiWAsML to its current version.

The core development technology stack of the Smartest system is as follows.

- Views – HTML, CSS, Bootstrap
- Front-end development – JavaScript, jQuery, vis.js for graphs editor
- Back-end development – Node, Express, Mongoose, EJS template engine
- Database – MongoDB
- Deployment – Heroku [183]

This section first provides the system's scenario with a use case diagram and then discusses the architecture of the Smartest web application and its conversion to a RiWA. Section 8.3 discusses the integration of the web service and low-level design aspects.

### 8.2.1. LMS – The Use Case Diagram

This section discusses the scenario of the Smartest system to understand its essential functions. The use case diagram is given in Figure 8.8, and some non-trivial use cases are described in Appendix C.1.

The main registered user types are Admin, Teacher, and Student; for the moment, the Researcher user type is considered a type of Student and shares the student's functions, and later, the Researcher is expected to provide some Teacher-level functions to manage entries.

In Smartest, the admins and teachers create the learning materials in a repository, which are stored and presented as graphs, where a graph is known as an "Entry". The registered users can browse and view the graph-based content in the repository or repo in short.





Figure 8.8 Use case: LMS – use case diagram

### 8.2.2. LMS – High-level Design

The architecture diagram, which was available when I joined the Smartest project, is shown in Figure 8.9. Even though the system’s functionalities are straightforward, understanding the development by looking at the code was stressful. I first created the UML use case diagram to gain a strong understanding of the functions and familiarized myself with the terminologies; then, I got it verified and finalized with the tech lead. After that, I started examining the original architectural diagram in Figure 8.9 to realise the implementation of the system.

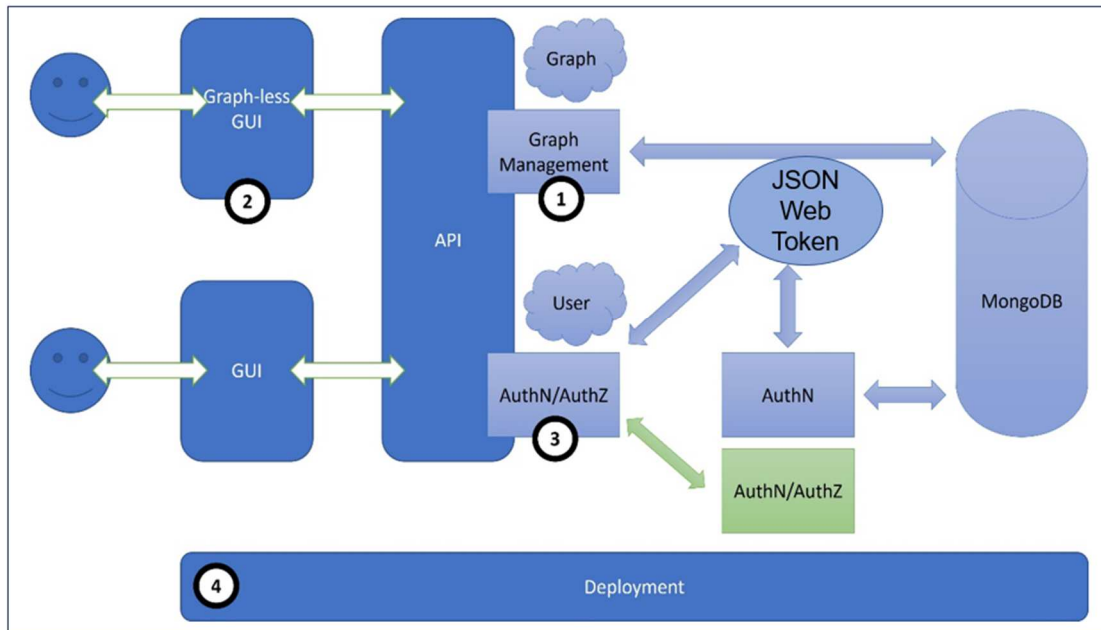


Figure 8.9 Use case: LMS – original box-and-line architecture

The questions I had while studying this architecture are stated below.

- Are *Graph Management* and *AuthN/AuthZ* elements (the blue elements on the left numbered 1 and 3) part of the API?
- What are the types of *Graph* and *User* elements shown as clouds?
- What is the type of the *JSON Web Token* element (components, data objects, etc.)?
- What are the types of *AuthN* and *AuthN/AuthZ* elements (on the right)?

After receiving the answers to the above questions from the tech lead and understanding the elements in this architecture, I drafted the architecture given in Figure 8.10 using the RiWAsML. A scaled-up version of the diagram is given in Appendix C.2. The *User*, *AuthN/AuthZ*, *Graph*, and *Graph Management* are internal modules of the server-model; therefore, they are not shown as elements in the new architecture.

The client with *Graph-less GUI* in the original architecture was not developed, and it was not planned to be developed in the next version (the RiWA with the web service); hence, it was not included in the new architecture. The HTTP and DC APIs are implemented in the *HTTPBus* and *DCBus*, which substitute the *API* element in the original architecture in Figure 8.9. The *JSON Web Token* (JWT) is used as a data element to implement the authentication, which is not an architectural element; it is included in the low-level design in Section 8.3.

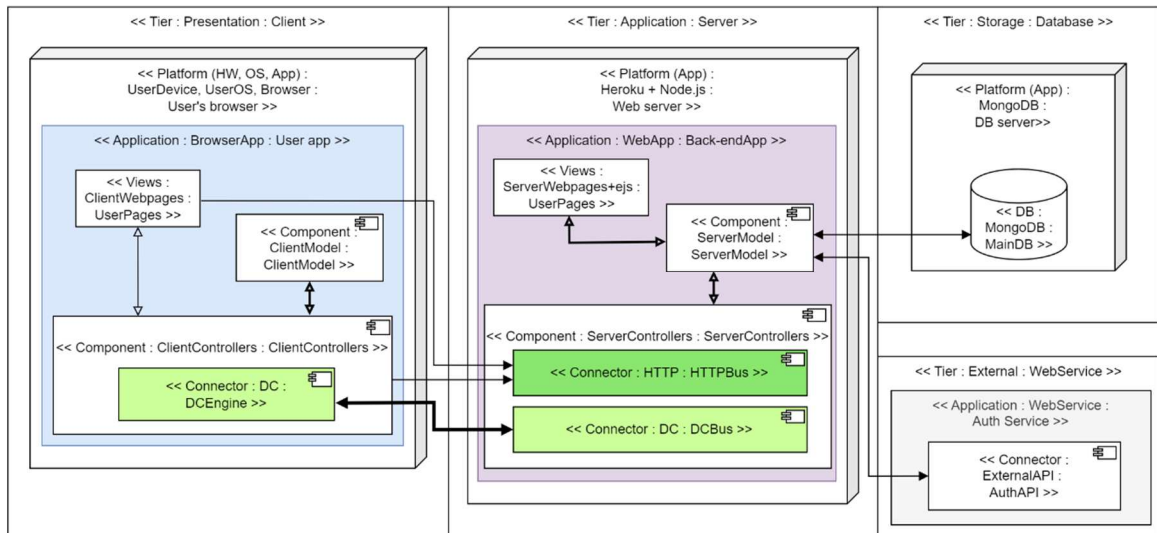


Figure 8.10 Use case: LMS – architecture designed with the RiWAsML

The web application is deployed on the Heroku [183] cloud platform, and the database is deployed on MongoDB [184]; thus, the *Web server* and *DB server* platforms only specify the application-level platforms without the hardware and OS levels. It was planned to use external authentication services like Google authentication in future; to assist that function, the *Auth Service Application* element is depicted in the *External* Tier.

This architecture provided sufficient knowledge to realise how the Smartest web application’s high-level elements are developed and deployed. Further, this new architecture helped plan how these architectural elements can be reused when converting the web application into a RiWA. The development aspects of the high-level elements, which align with the RiWAArch style, are elaborated below.

- The *Back-endApp* is developed using *NodeJS* and related technologies.
- The RESTful APIs in *HTTPBus* and *DCBus* are developed using the *NodeJS* library named *Express*.
- The views are developed using HTML, CSS, and Bootstrap. The *NodeJS*’s *EJS* template engine is used to dynamically generate content from the data retrieved by the views from the *ServerModel* and structure them within the web server before a view is sent to the user’s browser.
- The *ServerModel* uses the *Mongoose* library to use – and communicate with – the *MongoDB*.
- The *ClientControllers* and *ClientModel* elements are developed using JavaScript and related technologies.

### 8.3. Learning Management System With a Web Service

The Smartest project's [182] next step was to detach the server model logic from the web application and implement it in a web service, then convert the web application to a RiWA client of the web service. The back-end team was working on implementing the web service with RESTful DC APIs, and I was working on developing the browser-based client using the web application's web pages. This section discusses some issues we faced while implementing the Smartest RiWA with a web service and the solutions we came up with through the help of RiWAsML diagrams.

We used the top-down approach; we started with designing the architectural aspects and decided how to manage the development and deployment. The web service was already deployed into Heroku for testing; the web app's *Application* element was then deployed in a new Heroku instance, and I was working with that to build the client and server-side *Application* elements of the web app, which communicate with the web service. In the initial iteration, I designed how the login function should be implemented (refer to Section 8.3.2.2) and developed it since the *authN* and *authZ* functions are required to build and test the rest of the functions. After that, in each and every iteration, the team leader and I selected a function and then designed, developed, tested the function and deployed the new version of the web app. When we came across communication issues with the web service, discussions took place with the web service developers to identify the problems and come up with solutions.

#### 8.3.1. LMS with a Web Service – High-level Design

The architecture of the Smartest web application, as shown in Figure 8.10, was updated for conversion to a RiWA. The Smartest RiWA architecture is presented in Figure 8.11, and a larger diagram is given in Appendix C.3.

This Smartest RiWA architecture applies the following changes to the system in Figure 8.10.

- The core business logic is removed from the *ServerModel* and implemented in the *ServiceModel*, which is exposed to the clients via the *APIDCBus*.
- The *Mobile app* and the *Browser app* utilise the *APIService Application* element to process business logic.
- The *APIService Application* element utilises the database, and the *Back-endApp* does not.
- The *APIService Application* element utilises the *Auth Service* application, and the *Back-endApp* does not.

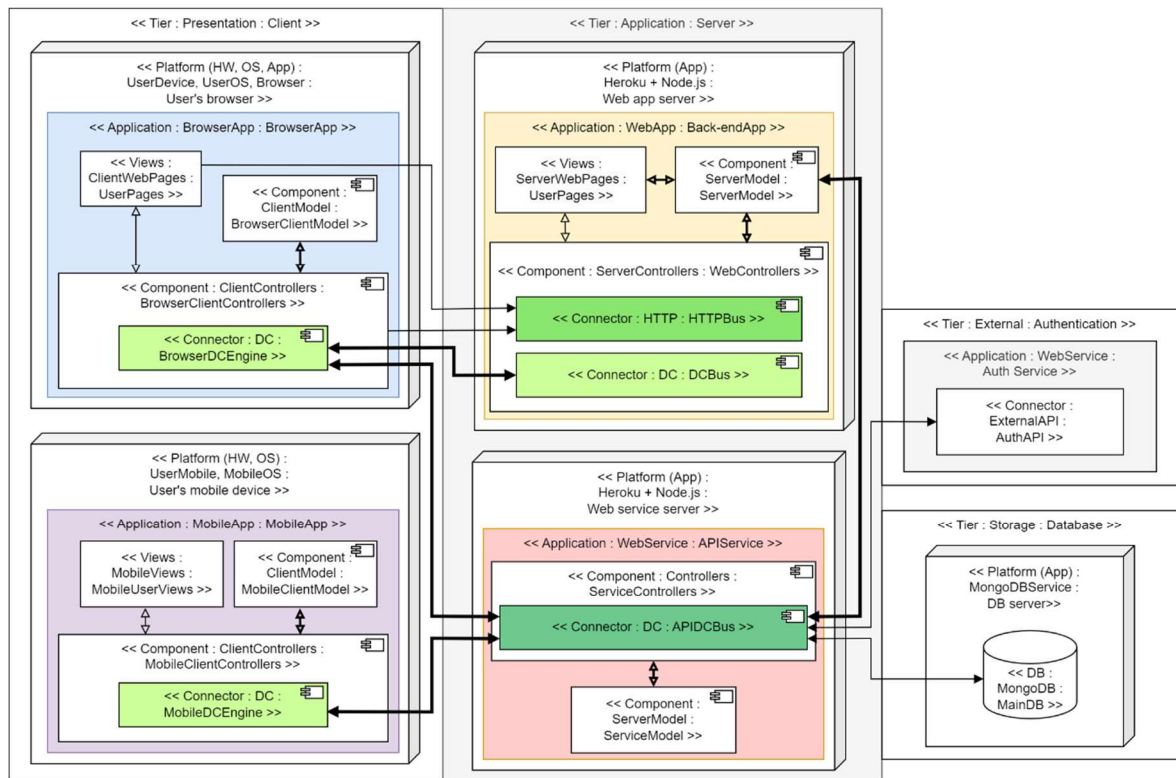


Figure 8.11 Use case: LMS with web service – L1+2 Architectural diagram

The low-level design and development are expected to address the following aspects.

- The *ServerModel* should only handle the authentication-related features for the *Browser app*'s web pages.
- *UserPages* should only utilise the *ServerModel* for authentication and session management-related functions.
- The *ServerModel* should use the *APIService Application* element for user authentication.
- Since the *ServerModel* does not contain business logic and does not utilise the database, *EJS* templating should be removed from the *UserPages*.
- Since the *UserPages* of the *Browser app* retrieve data from the *APIService* application, a client-side templating technology should be used to generate and structure dynamic content using the data retrieved from the *APIService Application* element.

The following section elaborates on how these aspects are addressed in low-level design and development with some examples.

### 8.3.2. LMS with a Web Service – Low-level Design

This section has selected some functions of Smartest RiWA, which required much attention in development due to their complexity and needed design assistance to understand them clearly. The selection of the functions to be discussed in this section is also influenced by the need to demonstrate all of RiWAsML's low-level design diagrams.

### 8.3.2.1. LMS with a Web Service – View-Navigation Diagram

Smartest’s *View-Navigation* diagram is depicted in Figure 8.12, and a larger version is given in Appendix C.4.

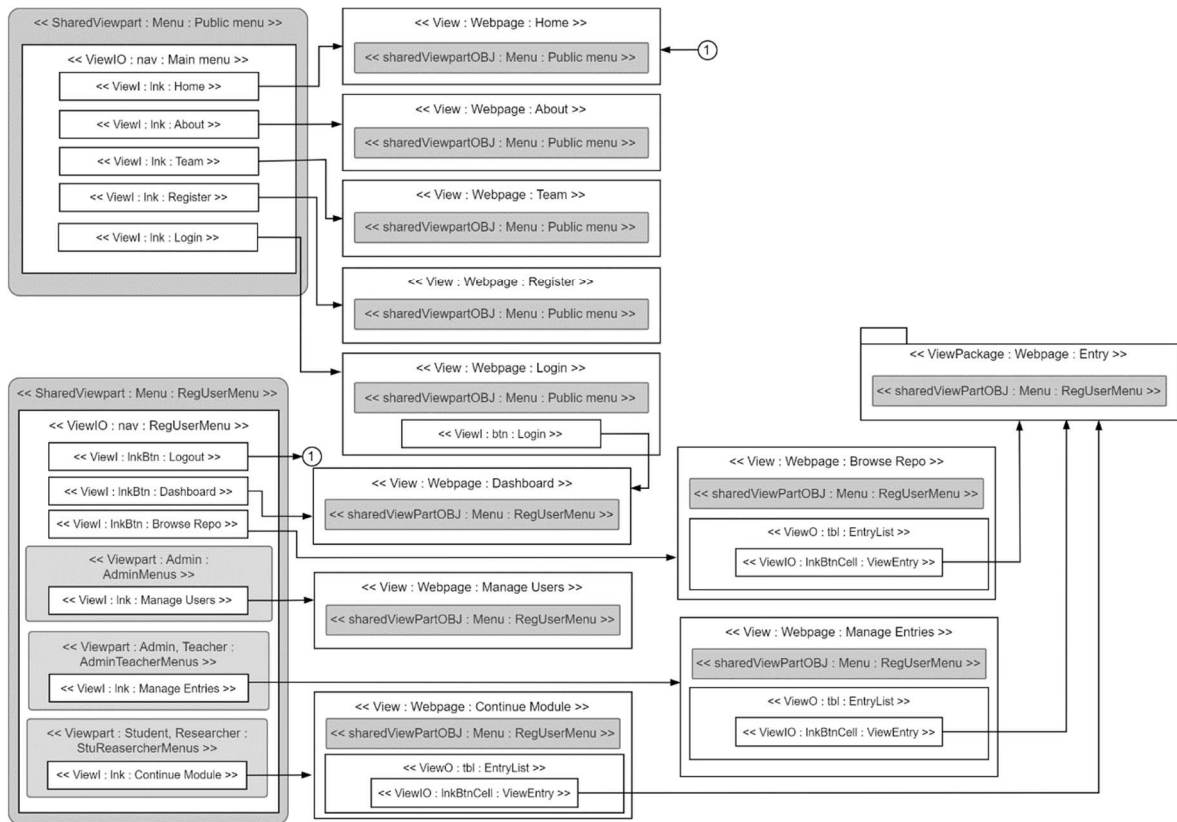


Figure 8.12 Use case: LMS – View-Navigation diagram

The Smartest system has two menus: a public menu for the general public on public web pages and the main menu for the registered users.

- A registered user can access the login page via the public menu, and once logged in, the user is navigated to the dashboard page.
- The registered users’ menu contains three common menu items: *Logout*, *Dashboard*, and *Browse Repo*; other menu items vary according to the user type.
- The logout menu item on the registered user’s menu logs out the user and navigates to the public home page.

This *View-Navigation* diagram does not contain all the views; only a subset of the menu items and their views are included to maintain the diagram’s size and make it fit into this thesis. Some key points to note are as follows.

- Admin users can access the *Manage Users* view to perform all the CRUD operations on their organisation’s users.

- The *Entry* web page is a complex view that implements many related features for different types of users, and it can be accessed via three different paths by different types of users.
  - All the user types can see the available public entries of the organization on the *Browse Repo* view, where the admins can also view private entries, and the teachers can view collaborating entries even if they are private. Users can see a selected entry by clicking on the entry's *ViewEntry* link button in the *EntryList* table. The entry will be then opened on the *Entry* view for viewing. Admins and the entry's owning and collaborating teachers can also edit the entry on the *Entry* view.
  - Admins and teachers are given permission to perform CRUD operations of the entries on the *Manage Entries* view, and they are provided with a menu item, *Manage Entries*, on the main menu to navigate to the view. Similar to the *Browse Repo* view, they are given an *EntryList* table on the view where admins can see all the organisation's entries, and the teachers can see all the owning and collaborating entries within the organization. Admins and teachers can click on an entry and navigate to the *Entry* view to edit the entry.
  - Students are given a menu item named *Continue Module* to navigate to the *Continue Module* view, where they can see a list of entries they have started to follow, which they can continue. Similar to the *Browse Repo* and *Manage Entries* views, the list of entries the student has started to follow is given in the *EntryList* table, and the student can click a selected entry's link button to open the entry on the *Entry* view.

By looking at the three navigation pathways to the *Entry* view, we can understand that the *Browse Repo*, *Manage Entries*, and *Continue Module* views share some features, and they can be implemented on a common view instead of developing three different views. Identifying such development-related possibilities is an excellent insight the *View-Navigation* diagram provides. In Smartest, the *Browse Repo* view contains some advanced filtering and searching functions (refer to Section 8.3.2.3), and the *Manage Entries* view contains features for performing CRUD operations on the entries; therefore, combining these views and implementing all these features on a shared view will increase the complexity of the view. In that case, the view should be carefully designed to accommodate all the features and enable/disable or show/hide them based on the user type and navigation path while maintaining a higher user experience.

### 8.3.2.2. LMS with a Web Service – View-Process Sequence Diagram for the Login Use Case

After the business logic is transferred to the *WebService* from the *WebApp*, the authentication is handled by the *WebService*. However, the *WebApp* still needs to identify the user to authorize access to the views; for example, the *WebApp* should not provide access to the admin's views for other types of users. Hence, there is a requirement to pass the user's authentication details to the *WebApp* after the *WebService* authenticates the user. This authentication function's process is designed using a *View-Process Sequence* diagram to realise it strongly, which is given in Figure 8.13.

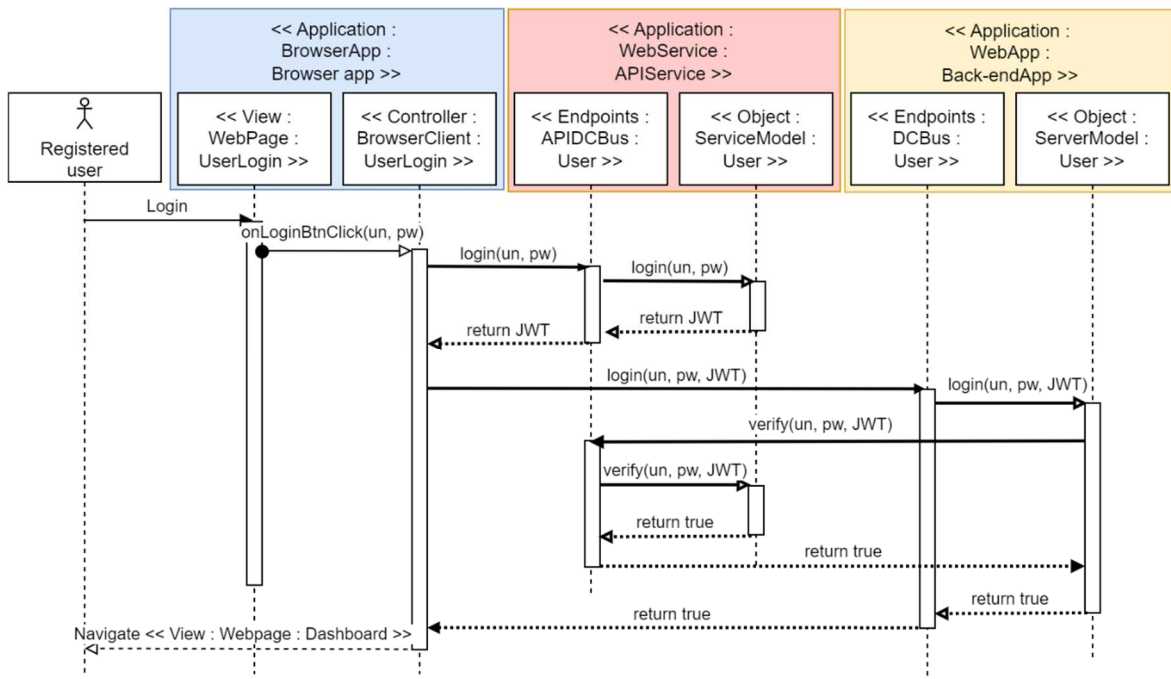


Figure 8.13 Use case: LMS with web service – View-Process Sequence diagram

The authentication process involves all three *Application* elements, and the steps are elaborated below with justifications where necessary.

- A registered user navigates to the *UserLogin* view, enters the username (*un*) and password (*pw*), and then clicks the login button to initiate the process.
- The *UserLogin* view triggers the login button's click event, and the *UserLogin* controller's *onLoginBtnClick()* event handler is invoked, which will read the *un* and *pw* from the view.
- The *UserLogin* controller sends a DC request, including the *un* and *pw*, to the *User EndpointsCollection* of the *APIDCBus* of the *APIService* *Application* element.
- The *APIDCBus* calls the *login()* method of the *ServiceModel*'s *User* class, passing the *un* and *pw* as parameters.
- The *User* class validates the user and returns a JWT object to the *APIDCBus*, which is then sent to the *UserLogin* controller as the DC response.
- Upon receiving a successful DC response from the *APIDCBus*, the *UserLogin* controller sends the next DC request to the *Back-endApp*'s *DCBus* element's *User EndpointCollection*, with *un*, *pw*, and the JWT-object received from the *Webservice*.
- The *User EndpointCollection* calls the *login()* method of the *User* class in the *ServerModel* by passing the *un*, *pw*, and the JWT-object as parameters.
- The *Back-endApp* cannot validate the user as it does not contain the required logic and does not have access to the database. Therefore, the *Back-endApp* sends a DC request with *un*, *pw*, and the JWT-object to the *APIDCBus* of the *Webservice* to verify the user.



- The *APIDCBus*'s *User EndpointsCollection* calls the *verify()* method of the *ServiceModel*'s *User* class by passing the *un*, *pw*, and the JWT-object as parameters.
- The *User* class verifies if the user is a valid and already logged-in user and returns *true* (a success message) to the *APIDCBus*, which is then forwarded to the *Back-endApps*'s *User* class as the DC response.
- Upon receiving a success message from the *APIService*, the *User* class of the *Back-endApp* sets the session-related data on the *WebApp* and returns *true* (a success message) to the *DCBus*, which is then sent to the *UserLogin* controller as the DC response.
- Upon receiving a success message from the *Back-endApp*, the *UserLogin* controller redirects the user to the next view, which is the user's *Dashboard* view in this case.
- Note that the error cases are not denoted on this diagram. If any error occurs during this process, an error message should be displayed on the *UserLogin* view instead of redirecting the user.

### 8.3.2.3. LMS with a Web Service – View-Controller Diagram of Browse Repo

This section discusses the *View-Controller* diagram of the *Browse Repo* function, which includes a *View* diagram and the related *ControllerClass* diagram. The *View-Controller* diagram of the *Browse Repo* function is illustrated in Figure 8.14.

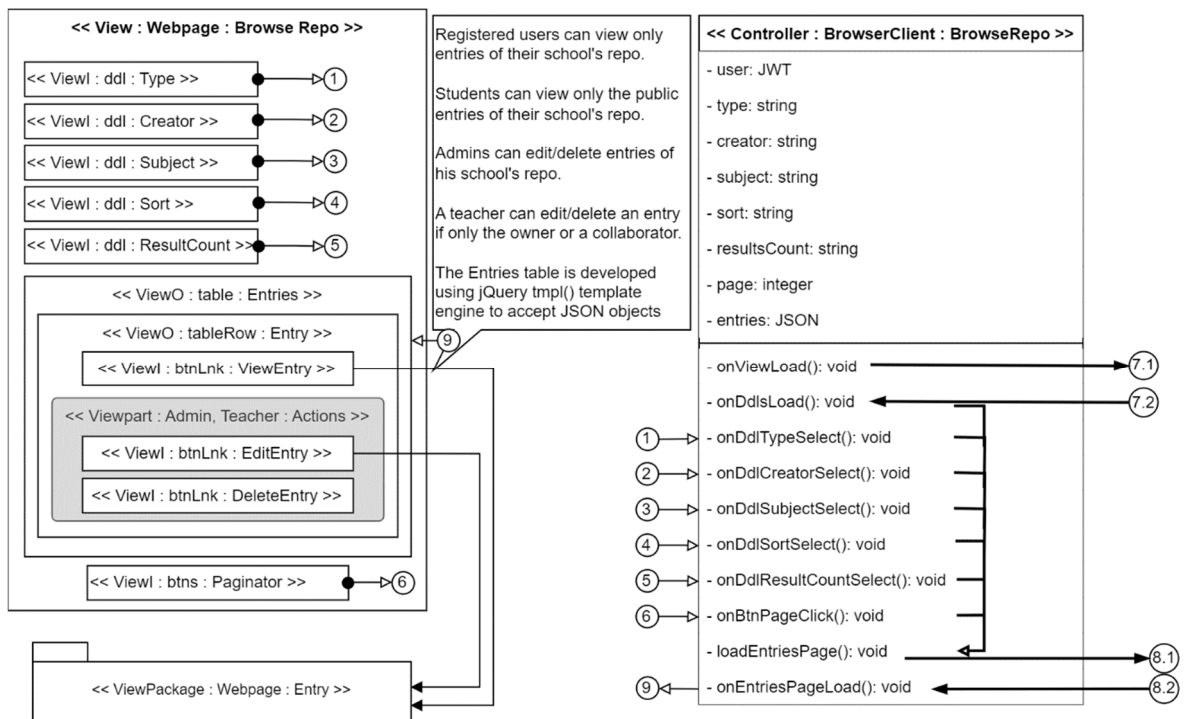


Figure 8.14 Use case: LMS with web service – browse repo View-Controller diagram

The *View-Controller* diagram answers the event handling and DC-related development concerns based on the RiWAArch style. The functions included in this diagram are explained below.

- The controller's properties are initialized with default values, and only the *user* property, which contains the JWT object, is dynamically assigned at the server when the view is requested from the *Web app server*. The JWT object is sent to the *APIService* with each DC request for user authentication. The *entries* property stores the set of entries as a JSON object sent by the *APIService*, as discussed later in this section.
- When the view is loaded to the browser, the controller's *onViewLoad()* event handler is invoked, which sends DC requests to the *APIService* to fetch data for the *Type*, *Creator*, and *Subject* drop-down lists.
- The *onDdlsLoad()* event handler is invoked upon receiving the DC responses to the above request, which reads the JSON data from the DC response and loads them to the drop-down lists (DDLs). Note that in actual development, three DC requests are sent to load data to DDL, and three separate DC response handlers are set to receive and process the responses. The diagram in Figure 8.14 is simplified to fit into this thesis's page.
- Once the DDL data is loaded, the *onDdlsLoad()* calls the *loadEntriesPage()*, which sends a DC request to the *APIService* with the default values of the controller's properties.
- The *onEntriesPageLoad()* is invoked upon receiving the DC response, which reads the JSON object sent by the *APIService* with the set of matching entries for the page (page number 1 is the default to start with) and sets the JSON object to the *entries* property of the controller.
- The table named *Entries* on the view is developed using the jQuery template library [185], which is denoted in the diagram's note. The *onEntriesPageLoad()* assigns the *entries* JSON object to the template, which dynamically generates the table data (denoted by connector 9).
- After the initial data load, when the user changes the DDLs or clicks on a page number on the *Paginator*, the relevant event handler on the controller is invoked, and the event handlers of the DDLs and paginator update the controller's appropriate property and call the *loadEntriesPage()* to send a new DC request to the *APIService*. When the DC response is received, it's processed again, as explained above.

On the table *Entries* of the view, the admin and teacher users are provided with *EditEntry* and *DeleteEntry* buttons to perform additional actions on the entries in the table. As mentioned in the view's note on the diagram, an admin can edit/delete entries of his school's repo, and a teacher can edit/delete an entry if only the owner or a collaborator.

The users are directed to the *Entry* view to view or edit a selected entry, as discussed in Section 8.3.2.1, and it is not to be included in the *Browse Repo* design. The delete entry feature requires more event handlers and DC-engine use; these artefacts and related discussions are kept out of the scope of this section to maintain the discussion's length.

### 8.3.2.4. LMS with a Web Service – DC-bus Diagram and EndpointsCollection Diagram

This section discusses the DC-bus and endpoints designing of the *Back-endApp*'s *DCBus* connector. The *DC-bus* diagram of the *DCBus* connector and the *EndpointsCollection* diagram of the *DCBus*'s endpoints collection named *RoutesRegUser* are given in Figure 8:15.

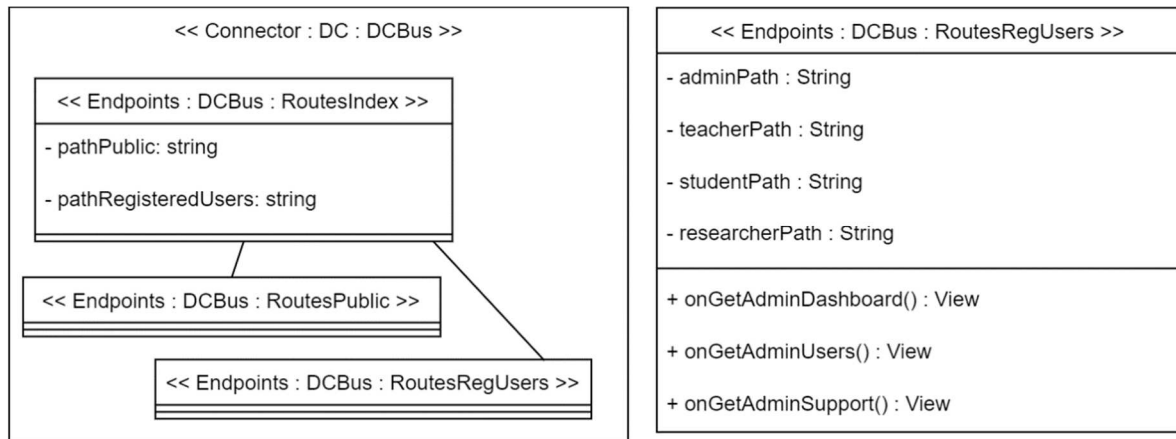


Figure 8.15 Use case: LMS with web service – DC-bus and an EndpointsCollection diagrams

The Smartest LMS with a web service has moved the business logic from the *Back-endApp* to the *APIService*; thus, the *Back-endApp*'s *DCBus* only implements the endpoints to serve the web pages to the *BrowserApp*. The DC requests are first served by the *RoutesIndex* endpoints collection, which checks if the request is for a public view or a registered user's view and forwards it to the appropriate endpoints collection.

The *DC-bus* diagram conceptually looks like it contains the *RoutesIndex* parent class and *RoutesPublic* and *RoutesRegUsers* children classes; actually, the relationships are associations, not inheritance. These are developed as separate *Express* controllers [173], and there is no parent-child relationship as in OOP. Furthermore, the *RoutesRegUsers* endpoints collection could be further decomposed based on the actors; however, in this case, since only view request handling endpoints are implemented in these, they do not contain many endpoints; hence, the endpoints for all the registered users are implemented in a single *EndpointsCollection* element.

The *RoutesRegUsers EndpointsCollection* diagram explains that four properties are available to hold paths for each registered user type. The actions/methods represent the request handlers for the registered users' views; only three request handlers for the admin's views are given in this diagram to maintain the diagram size for the thesis. The *Express* controllers implement the request handlers for RESTful requests, and the name of each action/method indicates the HTTP verb as "Get" to specify that the requests to be handled are HTTP GET requests, and the return is in the type of view. Note that the request handlers may use appropriate HTTP verbs, as required, to indicate the type of the request to be handled.

The *DC-bus* diagram helps to realise the overall arrangement of the DC handling of the *Application* element and, together with the *EndpointsCollection* diagram(s), provides enough guidance for the actual development.

### 8.3.2.5. LMS with a Web Service – AppModel Diagram and ModelClass diagram

This section briefly discusses the *AppModel* diagram of the *APIService*'s *ServiceMode*, which is illustrated in Figure 8.16.

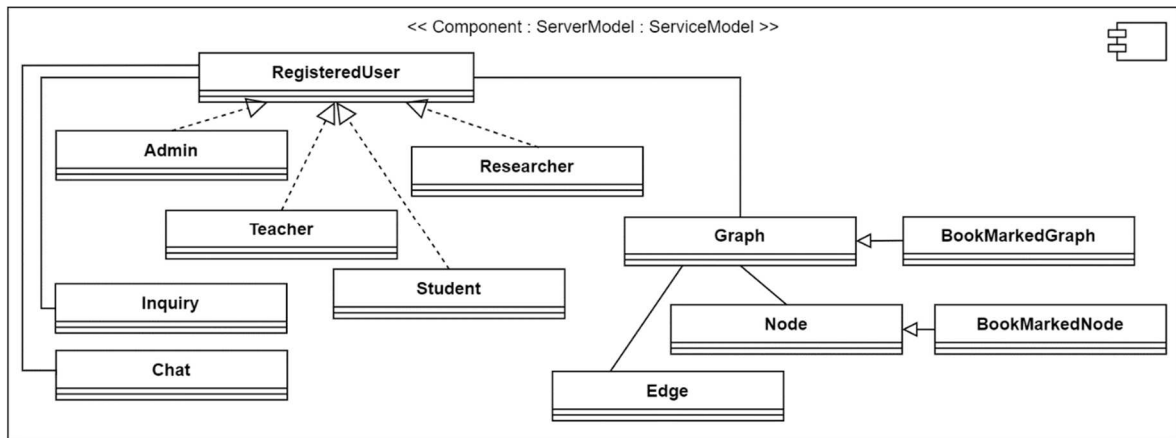


Figure 8.16 Use case: LMS with web service – an *AppModel* diagram

This is a simplified *AppModel* diagram without the properties and actions in the classes, and the model classes use only the RiWAsML label's name segment. Since the *AppModel* diagram is similar to the standard UML class diagram, in-depth discussions are avoided in this section. Each model class in the *AppModel* diagram can be independently designed as required, similar to a controller class. Also, if needed, the model classes can be designed together with *EndpointsCollection* elements to understand the model class's methods called by the endpoints. An example is given in Figure 8.17.

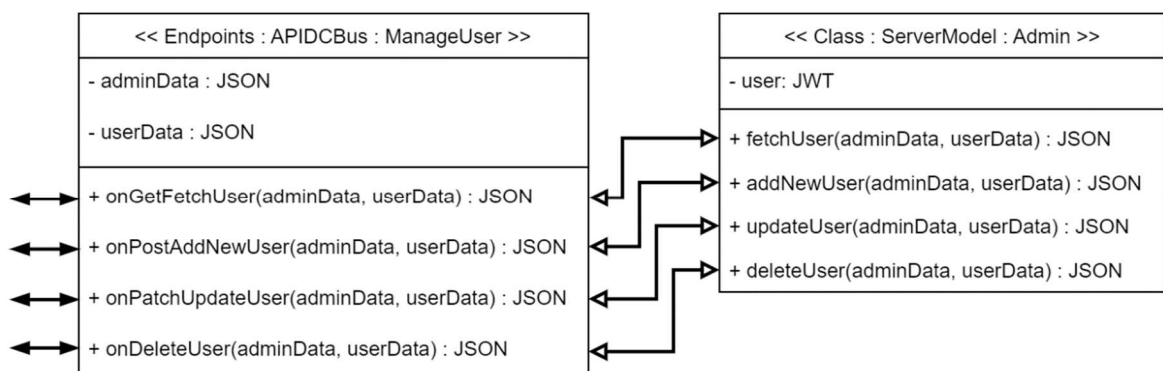


Figure 8.17 Use case: LMS with web service – an *EndpointsCollection* and *ModelClass* elements in one diagram

A simplified *Admin* class is given with only the methods required by the *ManageUser* endpoints collection. The endpoints collection's *adminData* property is required to authenticate and authorise the system's current user's JWT object, and the *userData* property is the user's data that the admin manages in the manage user function.

In the Smartest system, the *ServiceModel* is developed using NodeJS's features: *controllers*, *models*, *middleware*, and *services*.

## 8.4. Chapter Summary

This chapter demonstrates the adoption of RiWAsDM, including RiWAsML, through the following use cases, most of which are real-world systems.

- **A shopping system** demonstrates the improvement of the simplicity and usability of high-level designs using the *Tier* element.
- **Book club app**: This is a hypothetical use case taken from the *uml-diagrams.org* [179] to demonstrate how RiWAsML's *Platform* element improves the design's usability while preserving the simplicity compared to the UML's *node*.
- **MiCADO-Edge**: This is an architectural style for cloud-to-edge computing in a research publication [180], originally drawn as an incomplete architecture using the box-and-line approach. The style is reproduced using the RiWAsML to demonstrate the language's usability.
- **Smartest Learning Management System**: this use case demonstrates the conversion of a web application into a RiWA with the help of RiWAsDM/RiWAsML. The RiWA engineering of this use case utilises the top-down design approach in an AMDD environment. Both high-level and low-level design aspects are discussed to demonstrate the RiWAsML's models and model-elements, and their development is also discussed where necessary.

## Chapter 9. Evaluation

This chapter evaluates the RiWAsDM using three different methods and then draws the final evaluation results by triangulating the outcomes of these methods (refer to Section 1.5.5). The evaluation of the RiWAsDM fulfils research objective 5 (refer to Section 1.4).

### 9.1. Self-Evaluation

This section initiates the evaluation of the introduced RiWAsDM by discussing the satisfaction of the requirements within the context (refer to Section 1.5.5.1 for the self-evaluation method). The evaluation scale, which assesses the requirements satisfaction level, is given in Table 9.1.

Table 9.1 Evaluation scale (refer to Table 1.2 in Section 1.5.5.2)

Symbol	Interpretation
++	Very high effect
+	High effect
-+	Moderate effect
-	Less effect
--	Very low or no effect
NA	Not/None Applicable

The self-evaluation analysis is presented in Table 9.2. The facts in Table 9.2 are elaborated based on the use cases presented in Chapter 8 as proof of concepts. The effect of the RiWAsML/RiWAsDM elements/modules is examined against the requirements set in Chapter 4, and the overall effect of each requirement is considered the cumulative effect of the corresponding criteria.

#### R1 Element Names (refer to Section 4.2.1)

The RiWAsML primarily uses rectangles for model-element syntax instead of providing a dedicated symbol per element, and the model-elements are identified using a new descriptive *Label* element in the direction of improving learnability and readability. The RiWAsML's label with the three-segment format helps identify the element's class and type, which expresses some technical aspects, and a custom name can be assigned to identify the element within the designed RiWA uniquely. The details provided by the three-segment label format improve learnability, readability, and understandability while supporting development by stating technical information. Refer to Section 5.1.1 for a detailed discussion of RiWAsML's *Label* element. This new label is successfully used on all the new model-elements provided by the RiWAsML in use cases. These features together highly satisfy the R1 with a very high effect.

Table 9.2 The analysis of the self-evaluation results

Criteria/ Requirement	R1 Element names	R2 Comm channels	R3 Processing elements	R4 Views (high-level)	R5 Additional elements	R6 High-level models	R7 Views (low-level)	R8 Components	R9 Connectors	R10 Low-level models	Attr 1 Simplicity	Attr 2.1 Comprehensive	Attr 2.2 Usability	Attr 2.3 Dev support	Attr 2.4 Integrability
Label	++	NA	NA	NA	NA	NA	NA	NA	NA	NA	++	++	++	+	NA
Comm channels	NA	++	NA	NA	NA	NA	NA	NA	NA	NA	++	++	++	NA	NA
L1 App. Model -Tier -Platform -Application	++	++	++	NA	+	++	NA	NA	NA	NA	++	++	++	+	NA
L2 View- Proc. Model -Views -Components -Connectors	++	++	++	++	NA	++	NA	+	+	NA	++	++	++	NA	NA
L1+2 Arch. Model	++	++	++	++	+	++	NA	+	+	NA	++	++	++	NA	NA
Additional elements	++	NA	NA	NA	+	NA	NA	NA	NA	NA	+	++	++	NA	NA
View- Navigation Model	++	++	NA	NA	NA	NA	++	NA	NA	++	NA	++	++	++	NA
View Model	++	NA	NA	NA	NA	NA	++	NA	NA	++	++	++	++	++	NA
App-Cont. and Cont. Models	++	NA	++	NA	NA	NA	NA	++	NA	++	++	++	++	++	NA
View-Cont Model	++	++	NA	NA	NA	NA	++	++	NA	++	NA	++	NA	NA	NA
AppModel and ModelClass Models	++	NA	++	NA	NA	NA	NA	++	NA	++	++	++	++	++	NA
DC-bus Model and Endp. Coll. Model	++	NA	++	NA	NA	NA	NA	NA	++	++	++	++	++	++	NA
View-Proc. Seq. Model	++	++	++	NA	NA	NA	+	++	++	++	NA	++	++	++	NA
RiWAsML architecture	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	++	++	++	++	NA
RiWAsML rules and Guidelines	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	++	++	++	NA
RiWAsDM Process	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	++	+	+	++
<b>Overall effect</b>	++	++	++	++	+	++	++	++	++	++	++	++	++	++	++

### **R2 Communication Channels (refer to Section 4.2.2)**

The RiWAsML categorizes communication channels in RiWAs into 5 types and realises them based on the request-response model. An arrow denotes the communication channel and its direction, where different line and arrowhead styles are used to recognise the communication channel type (refer to Section 5.1.2). Further, the RiWAsML provides some more techniques like the use of numbered flow connectors for complex diagrams (refer to Section 6.1.2.), the use of a dot at the beginning of the arrow to indicate the event triggers (refer to Section 6.1.1.1), and use of stereotype labels (refer to Section 6.1.2.4) in low-level design to depict additional details on the communication channels. The communication channels are effectively utilised in high-level and low-level models: The *View-Navigation model*, *View-Controller model*, and *View-Process Sequence model*. These features of the communication channels satisfy the R2 with a very high effect.

### **R3 Processing Elements (refer to Section 4.3.1)**

RiWAsML separates the model-elements based on the RiWAArch style into tiers, platforms, applications, models, controllers, and connectors in high-level designing (refer to Section 5.2), which are utilised in the *L1 Applications model*, *L2 View-Process model*, and *L1+2 Architectural model* (refer to Sections 5.3.3, 5.3.4, and 5.3.5). Further, the internal elements of the high-level elements: *Views*, *AppControllers*, *AppModel*, and *Connector* are identified, and the following models are provided to design them: *View-Navigation model*, *View model*, *AppControllers model*, *ControllerClass model*, *View-Controller model*, *AppModel model*, *ModelClass model*, *DC-bus model*, *EndpointsCollection model*, and *View-Process Sequence model*. These processing elements and the models which use them satisfy the R3 with a very high effect.

### **R4 Views (High-level) (refer to Section 4.3.2)**

The RiWAsML provides a model-element to denote the high-level views on the *L2 View-Process model* and *L1+2 Architectural model* (refer to Sections 5.2.6, 5.3.4 and 5.3.5) aligning with the RiWAArch style, which satisfies the R4 with a very high effect.

### **R5 Additional Elements (refer to Section 4.3.3)**

The RiWAsML provide additional model-elements to show *actors*, *databases*, *files*, *external web services*, *networks*, *EBSs*, and *notes* on the *L1 Applications model* and *L1+2 Architecture model*, which satisfies the R5 (refer to Section 5.2.7). However, since a deep study, analysis, or discussions on the required additional elements were not done during this study, the R5 satisfaction level is considered (+) *high effect* rather than (++) *very high effect*.



### **R6 High-level Design Models (refer to Section 4.3.4)**

The RiWAsML provides the *L1 Applications* model to capture the *Application* elements within a RiWA, their platforms, and distribution in tiers (refer to Section 5.3.3). The *L2 View-Process* model is offered to design the high-level *Views*, *AppControllers*, *AppModel*, and *Connector* elements within an *Application* element (refer to Section 5.3.4). Further, the *L1+2 Architecture* model is provided to combine the L1 and L2 diagrams for less complex RiWAs to illustrate all the high-level aspects within a single diagram (refer to Section 5.3.5). Sections 8.1, 8.2.2, and 8.3.1 demonstrate the use of these high-level models using real-world development examples. These models satisfy the R6 with a very high effect.

### **R7 Views (low-level) (refer to Section 4.4.1)**

The RiWAsML provides a set of elements to model the views-related low-level aspects: *Views*, *ViewI*, *ViewO*, *ViewIO*, *Container*, *Section*, *Popup*, *Toggle*, *Viewpart*, *ViewpartOBJ*, *SharedViewpart*, *SharedViewparOBJ*, and *ViewPackage*, which are utilised in *View-Navigation* model (Section 6.2.1), *View* model (Section 6.2.2), and *View-Controller* model (Section 6.2.6). Moreover, the *View-Process Sequence* model employs the *Views* element to denote the GUIs with which the user interacts and initiates the process (Section 6.2.7). The provision of these models that utilise the views and related elements satisfies the R7 with a very high effect.

### **R8 Components (low-level) (refer to Section 4.4.2)**

The RiWAsML provides models and model-elements to design the low-level aspects of both high-level *AppControllers* and *AppModel* elements – which are the components included in the high-level *L2 View-Process* diagram and *L1+2 Architecture* diagram – mapping them into the low-level design.

The controller is an essential element in RiWAs; the RiWAsML's *AppControllers* model helps capture all the controllers in an *Application* element, and the *ControllerClass* model assists in the detailed designing of individual controllers. The *View-Controller* model assists in capturing the details of the events, events-handling, reading data from its view, and producing output on the view. Further, the *View-Controller* model can denote the invocation of popups and toggles. The *ControllerClass* model can depict communication with client-model and DC-bus, including DC response handling, which are unique features of the RiWAsML (refer to Sections 6.2.3 and 6.2.4).

The *AppModel* model and the *ModelClass* model of the RiWAsML provide additional details over the standard UML class diagram to state the model's communication with its server *Application* element's views and DC-bus's *EndpointsCollection* elements (Section 6.2.5).

Moreover, how the components in different *Application* elements are involved in a single process can be captured using RiWAsML's *View-Process Sequence* diagram (Section 6.2.7).

These features ensure the satisfaction of R8 with a very high effect.

**R9 Connectors (low-level) (refer to Section 4.4.3)**

The RiWAsML captures high-level connector details using the *L2 View-Process* model and *L1+2 Architectural* model. The low-level details of these connectors can be further explained using the *DC-bus* model and the *EndpointsCollection* model, using the elements *DC-Bus* and *EndpointsCollection* (refer to Section 6.2.5). Designing RiWAs' connectors is another unique feature of the RiWAsML, and it can ensure the satisfaction of R9 with a very high effect.

**R10 Low-level Design Models (refer to Section 4.4.4)**

The RiWAsML offers models to design the low-level aspects of the high-level elements *Views*, *Controllers*, *Model*, and *Connector*, through the *View-Navigation* model, *View* model, *AppControllers* model, *ControllerClass* model, *AppModel* model, *ModelClass* model, *DC-bus* model, and *EndpointsCollection* model. Also, the *View-Controller* model is provided to capture the interactions between a view and its controller. Further, some utility elements are given to denote more interactions between the elements in models; for example, numbered flow connectors can be used with *ModelClasse* elements to show the communication with *EndpointsCollection* elements. Section 8.3.2 demonstrates the utilisation of RiWAsML's low-level models using an actual RiWA engineering project. All these models and model-elements offer a comprehensive toolkit to design all the aspects of the RiWAs aligning with the RiWAArch style. Considering these points, the R10 can be regarded as satisfied with a very high effect.

**Attr 1 Simplicity (refer to Section 4.1.1)**

The RiWAsML's selection of model-elements is based on the RiWAArch style [12], and the RiWAsML inherits the RiWAArch style's simplicity by providing model-elements for all the RiWAArch style's architectural elements. This is further explained by the RiWAsML architecture as briefed below (refer to Figure 7.1 in Section 7.2.1 for a detailed discussion).

- For the high-level design, RiWAsML provides model-elements in two levels:
  - **Level 1:** The *Tier*, *Platform*, and *Application* elements capture the **deployment** of the *Application* elements. Additional high-level elements like databases are given to denote the elements required by the *Application* elements.
  - **Level 2:** The *Views*, *Controllers*, *Model*, and *Connector* elements – which are the internal elements of the *Application* elements – denote the **packaging** details to assist the development of the low-level elements.
- For the low-level design, RiWAsML provides model-elements to design the internal aspects of the views, controller, models, and DC-bus, supporting their **development**.

The RiWAsML's label helps identify the model-elements separately based on the element's class and the type without having to remember visual symbolic notations. The communication channels

provide different syntaxes to identify the 5 different communication channel types in RiWAs; further, their semantics can be easily understood by the elements engaged in the communication.

Considering the identification and provision of these elements, the simplicity of the RiWAsML can be marked as very high.

#### **Attr 2.1 Adoptability – Comprehensiveness (refer to Section 4.1.2.1)**

The discussion in Section 9.1.11 shows that the RiWAsML offers the model and model-elements to satisfy the features expected from the RiWAsML, which are set in Section 4.1.2.1.

The RiWAsDM provide the following modules to assist RiWAs engineering (refer to Chapter 7), satisfying the features expected from the RiWAsDM, which are set in Section 4.1.2.1.

- **The RiWAsML** provides models and model-elements to design all the aspects of the RiWAArch style, covering the general characteristics and essential features of the RiWAs (refer to Section 2.3.5) as discussed above under R1 to R10, which are listed below.
  - **Rich GUIs:** The *View-Navigation* model captures the views in a RiWA and their navigation aspects, focusing on implementing rich features on views. The *View* model helps design the GUI aspects to implement the rich features.
  - **A collection of applications and databases:** The high-level *L1 Applications* model and *L1+2 Architectural* model can capture the *Application* and *Database* elements and their configuration within a RiWA.
  - **Client-side events handling, Split business logic between client and server, and MVC-based modularization:** Aligning with the RiWAArch style, the RiWAsML offers models to capture these aspects based on BAW-MVC [105] using the models: *View* model, *AppControllers* model, *ControllerClass* model, *View-Controller* model, *AppModel* model, and *ModelClass* model.
  - **Use of DC and DC handling connectors:** At the high level, the RiWAsML provides the *Connector* element to denote the DC-engine and DC-bus in the *L2 View-Process* model and *L1+2 Architectural* model, and at the low level, the *DC-Bus* model and *EndpointsCollection* model are given for the detailed design of the DC-bus.
- **The RiWAsML architecture** (refer to Figure 7.1 in Section 7.2.1) ensures the provision of model-elements for designing all the RiWAArch style's elements and their development aspects. Further, it helps understand how the high-level elements are mapped to low-level elements and further mapped to development-level models and elements.
- **The RiWAsML rules and guidelines** (refer to Section 7.2.3) assist in adopting the RiWAsML into RiWAs designing and also guide some development aspects of the low-level models.
- **The RiWAsDM process** (refer to Section 7.3) discusses adopting the RiWAsML into RiWAs engineering, enabling agile model-driven development (AMDD) (refer to Section 7.3.3).

These modules make the RiWAsDM a complete methodology for RiWAs designing with a very high effect in the context of *Attr 2.1 comprehensiveness* (refer to Section 4.1.2.1).

**Attr 2.2 Adoptability – Usability (Learnability, Readability/Understandability) (refer to Section 4.1.2.2)**

The RiWAsML satisfies the features set in Section 4.1.2.2 as follows.

1. The RiWAsML is an extension of the UML therefore, inherits many features from the UML; for example, the *Platform* element is similar to the UML *Node* element, and *ControllerClass*, *ModelClass*, and *EndpointsCollection* elements are similar to UML *Class*. Therefore, engineers with UML knowledge may find it easy to learn the RiWAsML.
2. The RiWAsML is based on the RiWAArch style and realises the high-level views, controllers, models, DC-engine, and DC-bus elements and their configuration. Thus, for engineers who know the RiWAArch style, learning, reading, and understanding the RiWAsML designs would be easier. However, other than the DC-related aspects, the high-level tiers, platforms, *Application* elements, views, controllers, and models are generally known by RiWAs engineers; therefore, RiWAsML-based high-level designing would engage a low learning curve for them. Additionally, the RiWAsML architecture (refer to Figure 7.1 in Section 7.2.1) helps understand mapping the RiWAArch style-based high-level elements to development-level elements.
3. The RiWAsML uses a new *Label* element to distinguish between the model-elements over using many graphical symbols in the direction of assist in learning and understanding the language (refer to Section 4.2.1 for the *Label* element's requirements and Section 5.1.1 for the introduction of the *Label* element). Also, the RiWAsML encourages the use of notes to include more textual details on the designs to improve readability/understandability.
4. The low-level elements *ControllerClass*, *ModelClass*, and *EndpointsCollection* are primarily based on UML classes, which the software engineers generally know; thus, RiWAsML-based low-level designs would be easy to learn, read, understand, and directly use in development.
5. The RiWAsDM offers rules and guidelines for learning and understanding the proper use of RiWAsML in RiWAs' design and development (refer to Section 7.2.3).
6. Furthermore, the RiWAsDM process (refer to Section 7.3) assists in understanding the use of RiWAsML in RiWAs engineering by explaining the design approach, engineering approach, and integration of the RiWAsDM with agile methodologies.

Considering these aspects, the usability of the RiWAsML can be marked as satisfied with a very high effect in the context of learning, reading, and understanding.

### **Attr 2.3 Adoptability – Development Support (refer to Section 4.1.3)**

The RiWAsDM provides the following elements to support the development of the RiWAsML models and adopt RiWAs designing activities in engineering, satisfying the features set in Section 4.1.2.3 towards supporting development.

1. The RiWAsML models and model-elements align with the RiWAArch style's abstract formalism, realising the RiWAs' general characteristics and essential features as discussed in Section 9.1's Attr 2.1. Further, the RiWAsML is not based on any development technology/technique and is abstract, allowing it to be adopted into RiWAs engineering regardless of the development technology/technique.
2. The RiWAsML's low-level elements like *View*, *ControllerClass*, *ModelClass*, and *EndpointsCollection* are similar to the UML class and can be directly mapped into development. The RiWAsML is flexible, and some model-elements, such as GUI elements, events, and data types, are defined abstractly, allowing them to include details related to the development environment based on the languages and frameworks used. Also, the RiWAsML recommends including more development-related information using notes to improve the development support.
3. The RiWAsDM provides some rules and guidelines for the RiWAsML-based design to effectively map the low-level diagrams into development (refer to Section 7.2.3).

In addition to these features, the RiWAsDM presents the following in the direction of strengthening the development support.

- **The RiWAsML architecture** (refer to Figures 7.1, 7.2, and Table 7.1 in Section 7.2.1) assists in understanding the mapping of the high-level elements to low-level and development-level elements.
- **The RiWAsML's *Label* element** enables denoting some valuable details for development, especially the platform details, on which the development technologies depend. Also, it suggests selecting names using the Pascal case, which can then be used as package, module, or class names.
- **The high-level designs** provide details of the tiers and platforms to assist decision-making in selecting the development technologies and tools.
- **The views and related models** offer details required for implementing the functions for them; for example, which elements trigger events, which elements need templating, and which functions can be developed together on a view (refer to Sections 8.3.2.1 and 8.3.2.3 for use cases).

- **A real-world use case** is employed to demonstrate the use of the RiWAsML, and the development details are discussed to understand how the RiWAsML models can be developed (refer to Section 8.3).
- **The RiWAsDM process** discusses the design approach, engineering approach, and guidelines for integrating the RiWAsDM (refer to Section 7.3) to assist in understanding the adoption of the RiWAsDM in actual RiWAs engineering.

These points help understand that the RiWAsDM highly support RiWAs development without being limited to a conceptual solution.

**Attr 2.4 Adoptability – Integrability (refer to Section 4.1.4)**

The RiWAsDM provides a process that discusses how the RiWAsML-based design activities can be integrated into the RiWAs engineering life cycle in the direction of having an agile model-driven engineering (AMDD) methodology (refer to Section 7.3). The RiWAsDM process provides insight into making it a highly integrable solution in the context of the required level.

**9.2. Contextualised Comparisons**

Section 1.5.5.2 states the use of contextualised comparison as an evaluation method of the RiWAsDM. Only the UML-based available solutions are selected for the comparison (which are reviewed in Chapter 3), and the solutions limited to research publications are not chosen, except the IAML, which is in the same context and is much related (refer to Section 3.4). The analysis of the contextualised comparison is given in Table 9.3.

*Table 9.3 The analysis of the contextualised comparison*

Criteria/ Requirement	R1 Element names	R2 Comm channels	R3 Processing elements	R4 Views (high-level)	R5 Additional elements	R6 High-level models	R7 Views (low-level)	R8 Components	R9 Connectors	R10 Low-level models	Attr 1 Simplicity	Attr 2.1 Comprehensive	Attr 2.2 Usability	Attr 2.3 Dev support	Attr 2.4 Integrability
Solutions															
MDA+UML	-+	-+	-+	--	+	-	NA	-+	NA	-+	-	-	+	-+	+
Arc42	-+	-+	-+	--	+	-+	NA	NA	NA	-+	-	-	+	-+	-+
TAM	-+	-+	-+	-+	+	+	NA	-+	NA	-+	-	-	+	-+	--
ArchiMate	-+	-+	-+	--	+	-	NA	-+	NA	-+	-	-	-	-+	+
C4 model	-+	-+	-+	--	+	-+	NA	-+	NA	-	-+	-	-+	-+	--
UWE	-+	-+	NA	NA	NA	NA	+	-+	NA	-+	-+	-	+	-+	--
IFML	-+	-+	NA	NA	NA	NA	-+	-+	NA	-	-	-	+	-+	--
SysML	-+	-+	NA	NA	NA	NA	NA	-	NA	-	-	-	+	-+	++
IAML	-+	-+	NA	NA	NA	NA	-+	+	NA	+	-+	-	+	-+	--
RiWAsDM	++	++	++	++	+	++	++	++	++	++	++	++	++	++	++

Under each requirement, first, the effect of the UML or RiWAsML is discussed within the requirement's context and then used as a benchmark for other solutions. Since the available solutions' models and model-elements are already reviewed in detail in Chapters 3, 5, and 6, they are only discussed briefly in this section.

### **R1 Element Names**

The UML-based software modelling languages generally use a single string value to label the model-elements, which indicates the element's class, type, or name. Usually, graphical symbols are given to identify the element's class; therefore, the element's label particularly states the element's type or name. UML's stereotype labels provide an additional naming level to identify the element type. The main issue with the general element naming convention is that it does not assist in the language's and/or its designs' learnability and readability/understandability. Considering this limitation, this evaluation ranks the general UML-based naming convention as satisfying the R1 with a moderate effect.

The **RiWAsML**'s element naming label aids in learning the language elements and reading and understanding the diagrams; hence, the R1 is satisfied with a very high effect compared to the other solutions' naming labels.

### **R2 Communication Channels**

The available solutions do not explicitly realise the RiWA's communication channel types or request-response model. However, they can still denote the communication between the elements using unidirectional/bi-directional paths; hence, we can think the R2 is satisfied with a high effect. Still, the available solutions primarily do not realise the DC, which pulls down the impact of the known solution on the R2 and can be marked as a moderate effect within the context.

The **RiWAsML** consistently identifies the expected types of communication paths based on the request-response model across all the diagrams, satisfying the R2 with a very high effect compared to the other solutions.

### **R3 Processing Elements (High-level)**

The available high-level design solutions provide some of the RiWAs' high-level elements, and some of the available solutions' elements are likely to be exploited for the RiWAs' high-level elements. Anyhow, none of them offer elements for high-level connectors. Considering the elements provided and not provided, they are generally marked as satisfying the R3 with a moderate effect.

The **RiWAsML** delivers all the expected high-level processing elements: *Tier*, *Platform*, *Application*, *Component* (Model and Controller), and *Connector*, satisfying the R3 with a very high effect compared to the other solutions.

### **R4 Views (High-level)**

Only **TAM** provides a high-level view element on the *component/block diagram* [82]; nevertheless, this element is not included in the specification and is only shown on sample diagrams; thus, the TAM's effect on the R4 is considered moderate instead of high. Since other high-level design solutions do not provide architectural elements to denote views, they are marked as having no effect on R4.

The **RiWAsML** provides a high-level *Views* element, which supports multiple view types in different platforms, and the effect on the R4 is very high considering the other solutions.

### **R5 Additional Elements**

RiWAsML and other available solutions provide some additional supportive elements to be included in the architectural design. This thesis generally considered that all the solutions satisfy the R5 with a high effect.

### **R6 High-level Design Models**

**UML**'s *package* diagram can be exploited for the tiers, and UML provides the *deployment* diagram to design the deployment of the system elements. The UML *component* diagram can realise the high-level components. However, these diagrams are disconnected and do not provide an overall architectural realisation. Therefore, UML's effect on R6 is considered less effective.

**Arc**<sup>42</sup> and the **C4 model** provide a practical hierarchical high-level designing approach to realise the architectural formalism of the systems sufficiently. Nevertheless, they are not specified for the RiWAs, do not use formal notations, and lack formal guidelines; hence, they are measured as having a moderate effect on R6 compared to the UML.

**TAM**'s *component/block diagram* is an effective architectural diagram with formal syntax, potentially satisfying R6 with a high effect. Anyhow, the *component/block* diagram is not specialized for RiWAs and cannot realise DC and related aspects; thus, the effect on R6 cannot be considered very high.

**RiWAsML**'s *L1 Applications* diagram, *L2 View-Process* diagram, and *L1+2 Architectural* diagram strongly realise the RiWAs by aligning with the RiWAArch style, satisfying the R6 with a very high effect.

### **R7 Views (low-level)**

**UWE**'s presentation model provides a limited set of GUI elements to design views, and the *navigation* model captures link-based and process-based navigation between views. These features cause the UWE to have a substantial effect on the R7.



**IFML** provides some GUI elements that are just enough to capture the interaction flows of the views. IFML's effect on R7 can be considered moderate compared to the UWE.

Even though **IAML** discusses some view-related aspects, it lacks examples and discussions, lowering its effect on R7 compared to UWE and marked as a moderate effect.

**RiWAsML** provides a set of abstract GUI element classes to design views' functional aspects using the *View* model, and the *View-Navigation* model is given to capture the related functions to be implemented on the common views and different navigation paths to them. These features help the RiWAsML to satisfy R7 with a very high effect compared to the other solutions.

### **R8 Components**

**UML**, **TAM**, **UWE**, **C4 model**, and **UWE** assist in designing models using class diagrams; however, the controller is overlooked. Hence, the effect on R8 is considered moderate.

**Arc42** does not discuss domain modelling; therefore, it is marked as not applicable for R8.

**IFML** does not use the class diagram, yet it captures the event handling, which is a part of the controller, and is marked as having a moderate effect on R8.

**IAML** discusses domain modelling as well as the client-side events and related aspects; hence, its effect on the R8 can be considered high. The lack of model-elements and examples reduces the IAML's effect on R8 to a moderate effect.

**RiWAsML** explicitly provides the required elements to capture RiWA's model and controller elements, having a very high effect on R8 compared to the other solutions.

### **R9 Connectors**

The connector element is a unique feature of the RiWAsML, and none of the other available solutions look into connector design aspects for RiWAs. Because of this, the RiWAsML can be considered to satisfy the R9 with a very high effect compared to the other solutions.

### **R10 Low-level Design Models**

Most of the available solutions, including **UML**, **Arc<sup>42</sup>**, **TAM**, **ArchiMate**, and **UWE**, provide low-level design models. However, they are general and do not cater to the specificity of the RiWAs; they can be used to design some aspects of the RiWA; for example, the RiWAs' model can be designed using the UML's class diagram. Hence, the effect on the R10 is considered to be moderate.

Even though the **C4 model** is for the RiWAs, it mainly focuses on high-level modelling, and the effect on the R10 is less compared to the other solutions stated earlier.

**IFML**'s scope is limited to designing the interaction flows, and the effect on R10 is less.

Compared to the other solutions mentioned before, **SysML** provides fewer low-level models, which can be used for RiWAs, causing less effect on R10.

**IAML**'s context is RIAs/RiWAs, and its models are provided for low-level designing; compared to other solutions, IAML has a higher effect on R10. However, it lacks models for DC-related aspects, which limits the effect on R10 to stay at a high level instead of a very high level.

**RiWAsML** provides low-level design models for all view, controller, model, and DC-bus elements, having a very high satisfactory level on R10.

### **Attr 1 Simplicity**

**UML**, **Arc**<sup>42</sup>, **TAM**, **Archimate**, and **SysML** are general solutions, and they do not provide RiWAs-specific tools; therefore, they have less effect on Attr1.

**IFML**'s scope is limited to the interaction flows and has less effect on Attr1.

The **C4 model** appropriately identifies some high-level elements of the RiWAs, so the simplicity is higher than the other solutions. Nevertheless, it lacks support for low-level design aspects, which limits the effect on Attr 1 and can be considered moderate.

**UWE** and **IAML** capture some low-level elements of the RiWAs and can expect a high effect on Attr1. Still, they do not identify high-level elements; hence, they are considered to have only a moderate impact on Attr 1.

**RiWAsML** identifies all the required high-level elements of the RiWAs based on the RiWAArch style and further separates the high-level elements' internal elements in low-level designing in the direction of mapping them to development. The simplicity of the RiWAsML is very high in the context of this research compared to the other solutions.

### **Attr 2.1 Adoptability – Comprehensiveness**

Most available solutions are general and do not support the RiWAs context. Further, the available solutions only provide either high-level or low-level modelling tools. Therefore, none of them can be seen as comprehensive solutions in the context of RiWAsML and have less effect on Attr 2.1.

The **RiWAsML** addresses both high-level and low-level design aspects of the RiWAs and provides models and model-elements to satisfy R1 – R10; hence, the RiWAsML can be considered a highly comprehensive design solution compared to the available solutions.

### **Attr 2.2 Adoptability – Usability (Learnability, Readability/Understandability)**

**ArchiMate** is not based on UML and engages a higher learning curve; thus, the usability is less compared to the UML-based languages within the context of this research.

The **C4 model** is not difficult to use; however, it uses the informal box-and-line approach; thus, the usability is considered moderate in the context of the usability of UML-based methods.

The **RiWAsML** is UML-based and can be expected to be highly usable for software engineers in general, as the UML is the standard GPML. Moreover, the RiWAsML's label helps include more details in the designs to improve readability/understandability. The sufficient information provided by the new label format also helps RiWAs engineers learn the language with less effort. These features make the RiWAsML have a very high effect on Attr 2.2.

Other solutions are based on the UML and considered to have a good effect on Attr 2.2, which is still lower than the RiWAsML.

### **Attr 2.3 Adoptability – Development Support**

Available solutions primarily aid in designing systems using their models and model-elements; however, they use some design diagrams, for example, class diagrams, which can map into the development, so we can think they support development. Still, they do not intensely discuss how these designs should be developed and/or provide rules and guidelines to help the development. Further, since most of the available solutions are generic and do not support designing view, controller, and connector elements, they do not support the development of these elements. From the RiWAs-related solutions, the **C4 model** does not address views and connectors, **IFML** only focuses on the interaction flows, and **IAML** does not realise connectors and DC. Considering all these points, the development support provided by the available solutions is generally ranked moderate.

The **RiWAsML** provides models to design the views, controllers, models, and DC-bus, which are similar to the class diagram, and further detailed down to the level of individual detailed classes (refer to Section 9.1.14). Additionally, the RiWAsML/RiWAsDM provides rules and guidelines for development (refer to Section 7.2.3.3). These features make the RiWAsML/RiWAsDM very highly development-supportive compared to the other solutions.

### **Attr 2.4 Adoptability – Integrability**

**TAM**, **C4 model**, **UWE**, **IFML**, and **IAML** do not discuss how to integrate their designs into engineering and are marked as not affecting Attr 2.4.

**MDA** provides a guide [75] to integrate UML-based designing into MDSE, positively affecting Attr 2.4.

**Arc**<sup>42</sup>'s document template helps document some aspects like quality goals, architectural constraints, system designs, risks and technical details, which can assist in integrating the design into engineering. Taking them into account, **Arc**<sup>42</sup>'s effect on Attr 2.4 is considered moderate compared to MDA.

**ArchiMate**, as an independent solution and a framework, discusses many integration aspects, providing resources like guidelines, articles, and examples [186] [187]; therefore, it can be seen as a highly integrable solution.

**SysML** heavily discusses its integration into Agile MBSE [175], and it can be considered a highly integrable solution.

The **RiWAsDM** discusses integrating RiWAsML-based designing into RiWAs engineering (refer to Section 7.3) and can be seen as a highly integrable methodology.

### 9.3. Expert Evaluation

This section discusses the results of the expert evaluation, which was conducted using the method stated in Section 1.5.5.3. The experts' original feedback, including the experts' bios, is provided in Appendix D, and the summary is given in Table 9.4. In some criteria, the average of the related feedback sections is considered. For example, for R3 processing elements, the average of the *Tier*, *Platform*, *Application*, *Component*, and *Connector* elements' ranks is taken. The 2<sup>nd</sup> academic expert's feedback was not finalized by the submission time of the thesis; hence, not included in this section.

Table 9.4 The analysis of the expert evaluation

Criteria/ Requirement	R1 Element names	R2 Comm channels	R3 Processing elements	R4 Views (high-level)	R5 Additional elements	R6 High-level models	R7 Views (low-level)	R8 Components	R9 Connectors	R10 Low-level models	Attr 1 Simplicity	Attr 2.1 Comprehensive	Attr 2.2 Usability	Attr 2.3 Dev support	Attr 2.4 Integrability	Given overall rating
Solutions																
Expert 1	5	5	4.4	5	5	5	4.6	5	4.5	4.6	5	4	4.5	5	4	4
Expert 2	4.9	5	5	5	5	5	5	5	5	5	5	5	4.9	5	5	5
Expert 3	5	5	5	5	5	5	4.7	5	5	5	5	5	5	4	4	4.5
<b>Cumulative rating</b>	5	5	4.8	5	5	5	4.8	5	4.8	4.8	5	4.6	4.8	4.6	4.3	4.5

Overall, the experts have appreciated how the RiWAsDM satisfies most of the requirements. Some valuable points made by the experts are discussed below.

#### R3 Processing Elements

- **Expert 1** [*Application* element]: “it could be valuable to present a design in which a few executable applications run and interact within one platform. For instance, it would be interesting to see a diagram expanded for a solution including several microservices, which could potentially increase the overall learnability and comprehensiveness.”

- **Discussion:** It is indeed necessary to provide more use cases to demonstrate different criteria.
- **Expert 1** [*Connector* element]: “*With modern solutions such as message queuing services, it would be interesting to see how some alternative communication means could be represented on the diagram, too.*”
  - **Discussion:** Addressing advanced features and technologies is acknowledged as future work.

#### **R7 Views and Related Elements (low-level)**

- **Expert 1** [*SharedViewpartOBJ* element]: “*The main characteristics of the SharedViewpartOBJ elements were not always that easy to distinguish from the SharedViewpart elements for me, as they can share the same types and names, impacting the overall readability.*”
- **Expert 2** [*SharedViewpartOBJ* element]: “*Regarding ViewPartOBJ and SharedViewPartOBJ, to me, the “OBJ” wording seemed a bit too abstract and I thought another more descriptive word could be used instead, like “Region” or “section”; I may be wrong as I may not have as clear a picture behind the use of OBJ. I also think the use of capitalization and abbreviation of “Object” to “OBJ”, doesn’t quite fit in with the rest of the naming style used for the other elements.*”
- **Expert 3** [*SharedViewpartOBJ* element]: “*The purpose of this is not very clear.*”
- **Discussion:** The notations of the objects should be revised, and a better technique should be used to improve their usability.

#### **View-Navigation Model**

- **Expert 1:** “*usefulness could potentially be increased by providing more guidelines regarding the most effective selection of the views and their corresponding elements.*”
  - **Discussion:** This is a valid aspect that needs to be further studied and incorporated.

#### **DC-Bus Model and EndpointsCollents model**

- **Expert 1:** “*Comprehensiveness and development support could potentially be increased by introducing a higher level of detail for each endpoint, with some additional information regarding the request methods (as the method name might not always state it) or media types.*”
  - **Discussion:** This is an important point and should be incorporated.

#### **Development Support and Integrability**

- **Expert 3:** It is difficult to evaluate this attribute only by looking at the given examples.
  - **Discussion:** More use cases should be provided to demonstrate the RiWAsML.

## 9.4. Triangulating the Results

This section triangulates the results of the self-evaluation, contextualised comparison, and expert evaluations in the direction of drawing conclusions on the evaluation of the RiWAsDM. The analysis of the evaluation is given in Table 9.5.

Table 9.5 The analysis of the RiWAsDM evaluation

Criteria/ Requirement	R1 Element names	R2 Comm channels	R3 Processing elements	R4 Views (high-level)	R5 Additional elements	R6 High-level models	R7 Views (low-level)	R8 Components	R9 Connectors	R10 Low-level models	Attr 1 Simplicity	Attr 2.1 Comprehensive	Attr 2.2 Usability	Attr 2.3 Dev support	Attr 2.4 Integrability
Solutions															
RiWAsDM	++	++	++	++	+	++	++	++	++	++	++	++	++	++	++
MDA+UML	-+	-+	-+	--	+	-	NA	-+	NA	-+	-	-	+	-+	+
Arc42	-+	-+	-+	--	+	-+	NA	NA	NA	-+	-	-	+	-+	-+
TAM	-+	-+	-+	-+	+	+	NA	-+	NA	-+	-	-	+	-+	--
ArchiMate	-+	-+	-+	--	+	-	NA	-+	NA	-+	-	-	-	-+	+
C4 model	-+	-+	-+	--	+	-+	NA	-+	NA	-	-+	-	-+	-+	--
UWE	-+	-+	NA	NA	NA	NA	+	-+	NA	-+	-+	-	+	-+	--
IFML	-+	-+	NA	NA	NA	NA	-+	-+	NA	-	-	-	+	-+	--
SysML	-+	-+	NA	NA	NA	NA	NA	-	NA	-	-	-	+	-+	++
IAML	-+	-+	NA	NA	NA	NA	-+	+	NA	+	-+	-	+	-+	--
Expert evaluation	5	5	4.8	5	5	5	4.8	5	4.8	4.8	5	4.6	4.8	4.6	4.3

The self-evaluation shows that the RiWAsDM meets the requirements with a very high effect, based on the use cases as proof of concept and reasoning. The contextualized comparison verifies that the RiWAsDM satisfies the requirements with a higher-level impact than the available solutions. The expert evaluation states that the RiWAsDM delivers the features to satisfy the requirements with high effectiveness. Altogether, the RiWAsDM can be considered a simple and adoptable design methodology for the RiWAs.

## 9.5. Chapter Summary

This chapter evaluates the RiWAsDM, and the following results are observed.

- The **self-evaluation** finds that the RiWAsDM satisfies all the requirements set in Chapter 4 with a very high effect except for the *R5 Additional elements*. More elements to extend the RiWAs from the core 3-tier architecture should be studied in depth and considered a future work.

- The **contextualised comparison** reflects that the RiWAsDM satisfies all the requirements set in Chapter 4 with a very high effect, which is higher than the available solutions in all the criteria except two cases. (1) *R5 Additional elements* requirement is considered *high effect* similar to other high-level design solutions and requires more profound study. (2) in the case of *Attr 2.4 Integrability*, SysML also exhibits a very high effect. This thesis recommends conducting a study to examine SysML's integrability and identify potential aspects that could strengthen the RiWAsDM's integrability.
- The **expert evaluation** shows that the RiWAsDM meets the requirements with high effectiveness.

By triangulating the results of the self-evaluation, contextualised comparison, and expert evaluation, it can be concluded that the RiWAsDM is a simple and adoptable design methodology for the RiWAs.

## Chapter 10. Conclusion

---

This chapter first discusses the achievements of the research objectives, proof of the hypotheses, and the fulfilment of the research aim. Then, the research contributions are stated, the challenges faced are reflected, and identified limitations are discussed. Finally, possible future work is pointed out by concluding the thesis.

### 10.1. Achievements of Research Aim and Objectives

This section concludes the following aspects: the achievements of the objectives, proof of the research hypotheses, and the fulfilment of the research aim.

#### 10.1.1. Achievements of the Objectives

The research objectives set in Section 1.4 are revisited here, judging their achievements by the outputs produced by this thesis.

**Obj 1. Identify an abstract comprehensive architectural style for the RiWAs:** The RiWAArch style is identified in Section 3.1 as a potential style that can realise the RiWAs' general characteristics and essential features given in Section 2.3.5. The RiWAsML is introduced based on the formalism of the RiWAArch style.

**Obj 2. Introduce the RiWAsML:** Chapter 4 sets the requirements for the RiWAsML based on the RiWAArch style in the direction of addressing the RiWAs' general characteristics and essential features, and Chapters 5 and 6 introduce the RiWAsML to design high-level and low-level aspects of the RiWAs' respectively.

**Obj 3. Introduce the RiWAsDM:** Chapter 7 implements the RiWAsDM, based on RiWAsML, introducing a process to adopt RiWAsML-based designing into RiWAs engineering with the agile model-driven development (AMDD) approach.

**Obj 4. Demonstrate the utilisation of the RiWAsDM through use cases:** Chapter 8 demonstrates the adoption of the RiWAsML/RiWAsDM using real-world use cases.

**Obj 5. Evaluate the introduced RiWAsDM:** Chapter 9 evaluates the RiWAsDM and shows that the requirements are satisfied with high effectiveness, and the simplicity and adoptability are more effective than the available solutions. The expert evaluation verifies the RiWAsDM's simplicity and adoptability are in high rank. Overall, the ultimate triangulated evaluation results ensure that the RiWAsDM is a simple and adoptable solution for the RiWAs.

Considering these aspects, it can be concluded that all the research objectives are fulfilled within the context of this thesis.



### 10.1.2. Proving the Hypotheses

This section proves the research hypotheses set in Section 1.3 using reasoning based on the fulfilment of the research objectives discussed in the previous section.

- **Hypothesis 1** ( $H_01^1$ ): Achievement of research objective 1 ensures that an abstract comprehensive architectural style named the RiWAArch style, which realises the RiWAs' general characteristics and essential features, is identified. Chapter 4 identifies a complete set of models and model-elements to design RiWAs, covering all the general characteristics and essential features realised by the RiWAArch style and sets requirements for these models and model-elements. The identification of a set of models and model-elements required to completely design RiWAs based on the RiWAArch style, addressing all the RiWAs' general characteristics and essential features, proves the  $H_01$ .
- **Hypothesis 2** ( $H_02^2$ ): The fulfilment of research objective 2 confirms that a UML-based DSML named RiWAsML for RiWAs is implemented to satisfy the requirements set for the DSML's models and model-elements while proving  $H_01$ . Achievement of objective 5 guarantees that the introduced RiWAsML is evaluated and its simplicity and adoptability are recognized. These aspects prove the  $H_02$ .
- **Hypothesis 3** ( $H_03^3$ ): Research objective 3 ensures that the RiWAsDM is introduced based on the RiWAsML, which includes design and engineering approaches and guidelines for adopting RiWAs designing into AMDD. Research objective 5 is fulfilled by ensuring the RiWAsDM is simple and adoptable with a very high effect. These points confirm that the  $H_03$  is proven.

### 10.1.3. Fulfilment of the Research Aim

Proof of research hypotheses confirms that the proposed RiWAsDM is implemented. Research objectives 4 and 5 ensure the RiWAsDM's simplicity and adoptability by demonstrating it with real-world use cases and rigorously evaluating it with multiple methods to show its validity. These features prove that the research aim<sup>4</sup> is effectively achieved.

---

<sup>1</sup> **H<sub>01</sub>**: A comprehensive set of RiWAs design models and model-elements can be identified to design all the general characteristics and essential features of the RiWAs based on a solid abstract architectural style, maintaining higher simplicity and adoptability.

<sup>2</sup> **H<sub>02</sub>**: A simple and adoptable UML-based DSML for RiWAs can be implemented using the comprehensive set of models and model-elements identified while proving the  $H_01$ .

<sup>3</sup> **H<sub>03</sub>**: A simple and adoptable RiWA design methodology can be produced – utilising the UML-based DSML introduced while satisfying the  $H_02$  – which provides RiWAs design and engineering approaches and guidelines for adopting RiWAs designing into agile-SE.

<sup>4</sup> The aim of the research is to introduce a simple and adoptable novel design methodology for the RiWAs, named *RiWAs Design Methodology* (RiWAsDM), whose adoptability is demonstrated with use cases and evaluated with multiple methods to ensure validity.

## 10.2. Contributions

This thesis contributes the following artefacts to the domains of RiWAs engineering, UML-based designing, and DSML.

1. **A design methodology implementation process:** Even though some DSML implementing processes are available, they do not discuss extending the DSML into a design methodology. This thesis offers a new process with 3 steps to introduce a domain-specific design methodology with a new DSML (refer to Section 1.5.3).
2. **A new simple and adoptable Rich Web-based Applications Modelling Language:** This thesis introduces a new DSML for RiWAs named RiWAsML, which includes the following unique features.
  - 2.1. **A *Label* element with a new naming format convention:** The RiWAsML's *Label* element with a new format (refer to Sections 4.2.1 and 5.1.1) can be seen as the strength of the RiWAsML, which improves the **usability** by reducing the use of new graphical symbols as notations and increasing the text-based details on the diagrams.
  - 2.2. **The communication channels notation:** RiWAsML introduces a set of request-response model-based communication channel notations, which can be seen as an effective concept for improving the designs' clarity and **usability**. These communication channels provide syntax to denote the communication between the elements across the high-level to low-level diagrams in a consistent manner.
  - 2.3. **A comprehensive set of models and model-elements for RiWAs:** The models and model-elements of the RiWAsML cater to the specificity of the RiWAs by realising the general characteristics and essential features of the RiWAs based on the RiWAArch style, and these models offer the following unique features (refer to Section 4.1.2.1 for the criteria of Comprehensiveness).
    - 2.3.1. **RiWAs Architecture:** A comprehensive set of architectural designing tools, with 3 models and their model-elements, which can realise the following based on the RiWAArch style: RiWAs' *Application* elements, their deployments to the platforms, their logical arrangements into the tiers, and also the internal high-level elements – the views, controllers, models, and connectors – of the *Application* elements and their configuration.
    - 2.3.2. **View-Navigation model:** This model captures – not only the navigation, like the available solutions – but also the related functions to be implemented on common views and multiple paths to navigate to them by different actors. The information captured by a *View-Navigation* diagram helps make decisions on merging/splitting the views based on their functions in the direction of improving the user experience and better management of the development.

- 2.3.3. **View model:** This model provides abstract GUI element categories to design the function-related GUI elements towards support developing them with the target technologies.
- 2.3.4. **View-Controller model:** Capture the views' events handling and related processing to support the controller development based on OODD practices towards improving the user experience.
- 2.3.5. **DC-bus model and EndpointsCollection model:** Design the server-side APIs and their communication with the other elements based on OODD practices.
- 2.4. **The RiWAsML is simpler, more usable (learnable and readable/understandable), and more development-supportive:** The models and model-elements are based on the RiWAArch style [12]; hence, firmly maintain the principle of separation of concerns, assisting the DSML to be simpler compared to the available solutions. The RiWAsML's new *Label* element assists in learning the DSML without remembering/referring to various graphical symbols. The *Label* element also promotes the DSML's readability/understandability, ensuring that anyone with some RiWAs engineering knowledge can read and understand the RiWAsML designs even if the DSML is unknown. Further, based on the RiWAArch style, the RiWAsML's model elements are much related to the actual development elements that provide development support, which is further enhanced by the RiWAsML's rules and guidelines.
3. **A new UML extension for the RiWAsML:** A set of new UML profiles for the new models and model-elements are introduced to guarantee the RiWAsML is a UML extension.
4. **An integrable new design methodology for the RiWAs named RiWAsDM:** This methodology provides rules and guidelines to design RiWAs using the RiWAsML and map the designs to development, and most importantly, offers a process to integrate RiWAsML-based designing into RiWAs engineering enabling AMDD approach.
5. **Demonstration of the use of the RiWAsML/RiWAsDM through real-world use case:** Available solutions offer hypothetical use cases to demonstrate the use of their models; the RiWAsML models' utilisation is shown using some real-world use cases, which can be seen as a distinctive feature of this thesis. Further, the utilisation of the RiWAsML's models and model-elements is demonstrated using a dedicated single real-world use case, showing the connectivity of all the design diagrams based on the top-down design approach and AMDD approach (refer to Sections 8.2 and 8.3).

### 10.3. Reflections on Challenges

The challenges faced by this research are stated in this section.

- **Research methodology:** This is a conceptual research aiming to introduce a design methodology that satisfies some quality attributes such as simplicity, learnability, readability, and understandability. Constructing a methodology for conceptual and qualitative software engineering research was difficult due to the lack of related literature. At the core of this thesis's research methodology, first, the design language implementation method was created based on the waterfall method and literature, and then it was upgraded to the design methodology implementation level inspired by the RiWAArch style implementation method (refer to Section 1.5.3). The other research methods were incorporated to assist the core methods and produce a complete research methodology for this thesis.
- **Implementing the RiWAsML:** Even though it was decided to be based on the RiWAArch style, it wasn't easy to figure out how to initiate implementing the RiWAsML. It took longer than expected to finalize the high-level elements and their notations. When selecting notations, after many iterations, the decision to exploit the element label to identify different element classes and types was taken, and then it took more time to experiment and come up with the label format. When the RiWAsML grows from the high-level elements to the low-level aspects, some decisions taken at later levels affected the already completed work; in order to maintain the consistency between the already constructed and new work, there were numerous incidents to go back and update the completed work; consequently, redraw or amend many diagrams and text. For example, while working on the low-level *View* and *ControllerClass* models, it was noted that at the high level, the relevant elements should be plural (*Views* and *Controllers* elements), even though they are in singular form on the RiWAArch style. For another example, the communication channel notations were first experimented with many variations and finalized using labels. After that, when working on the Smartest system (refer to Sections 8.2 and 8.3), these communication channels with labels were tried, and it was noted that they make the diagrams untidy. Hence, better techniques were required, and then the communication channels were re-analysed, and the current arrow style-based notations were formalized. The new communication channel syntax led to revising old discussions on the communication channels and redrawing all the previously drawn diagrams with the new notations.
- **Examples and Use Cases:** It was tough selecting the examples and use cases to demonstrate the utilisation of the RiWAsML. The example scenarios were required to show the utilisation of the model-elements and models presented during the introduction of the RiWAsML in Chapters 5 and 6. These scenarios were hypothetical and had to be designed to match the context while exhibiting all the required features of the models and model-elements. Finding matching real-world use cases is phenomenal, and it could be an impossible task; I personally think I was

extremely fortunate to work on the use cases discussed in Chapter 8, especially the Smartest project, during the final year of the research, thanks to my director of studies, Dr Alexander Bolotove, who offered me a place to work in the project as an architect and a developer.

- **Implementing the RiWAsDM:** There is no literature discussing how to implement a design methodology or what should be exactly included in a design methodology. The available methodologies like MDA, ArchiMate, or SysML do not explicitly discuss the features of a design methodology, and they include some different features. The modules of the RiWAsDM are selected intuitively, considering the research context and the known contemporary status of the industry through personal experience. Validating these features is challenging, and this research is merely based on the feedback of the expert evaluation for supporting the RiWAsDM.
- **Evaluating the RiWAsML and RiWAsDM:** Evaluating conceptual and qualitative work is extremely challenging, especially with minimal literature support. One could argue that assessing a software engineering artefact without empirical evidence is subjective and cannot be accepted. I personally think that it's only a matter of time to observe the acceptance of a concept by the industry, and collect evidence over time to evaluate the actual validity of the concept.

#### 10.4. Limitations

The work presented by this thesis is mainly limited by the scope given in Section 1.5.2. Some other identified limitations are as follows.

- **Single-paged RiWAs:** RiWAs can be developed as single-paged applications (SPA), where all the functionalities are developed in a single web page. The SPAs may have specific general characteristics; this thesis does not look into the SPAs specifically.
- **Integration with UML notations:** Some UML diagrams, such as activity diagrams and state chart diagrams, can be helpful in RiWAs designing. Utilising them with the RiWAsML is not straightforward because of the RiWAsML's new label and model-elements. This thesis does not discuss the use of the available UML models and model-elements together with the RiWAsML.
- **Language specification:** This thesis does not provide an OMG-compatible, complete specification for the RiWAsML.
- **Tools and exchange format:** This thesis does not attempt to develop a case tool for the RiWAsML or give an XMI-based specification to support integrating the RiWAsML into the available CASE tools.

## 10.5. Future Work

Possible future work as the follow-up of this thesis is given below.

- **Complete RiWAsML specification:** The next step of this research should be to compile a comprehensive specification for the RiWAsML, identify the steps required and get approval from OMG to make it a recognized standard. The OMG assists in drafting ISO standard specifications and submitting them to ISO for approval. The OMG also publishes the finalized specifications.
- **Implement a CASE tool and produce XMI for CASE tool integration:** The practical use of a DSML would not be feasible without a CASE tool, and it is essential to develop a CASE tool for the engineers to start utilising the new DSML. Also, providing XMI specifications is vital for engineers using the RiWAsML/RiWAsDM in their regular CASE tools.
- **Study integrating UML meta-model diagrams and elements with RiWAsML:** It would be beneficial to utilise the UML meta-model's diagrams and elements when required rather than specifying a new version of them for the RiWAs; for example, the UML activity diagram can be used to model the processes in RiWAs. This aspect should be further studied to identify the possibilities, concerns, and solutions.
- **Expand the scope of the RiWAsML by addressing the limitations of the work:** Further research is required to expand the scope of the RiWAsML to assist larger and more complex RiWAs with cutting-edge functions engineered with advanced technologies (refer to the scope in Section 1.5.2). Primarily, cloud computing, DS/AI/ML, IoT, and micro-services-related functions are suggested for consideration.
- **Use cases:** The RiWAsML/RiWAsDM require more use cases to demonstrate its adoption in various types of RiWAs in different sizes in the direction of supporting RiWAsML/RiWAsDM's learning and utilisation. In the case of expanding the scope of the RiWAsML, adopting the new versions of the RiWAsML should be comprehensively demonstrated through use cases.

In order to put the RiWAsDM into practice, a complete language specification and CASE tool support are essential. Within the context of this thesis, the RiWAsDM is a simple and adoptable solution for RiWAs engineering. Even though the scope of the RiWAs is limited to the basic RiWAs in 3-tier architecture, the use cases evident that the RiWAsML/RiWAsDM is potential for supporting RiWAs with web services (refer to Section 8.3) and *Application* elements running in cloud services (refer to Section 8.1.3), and hopefully the RiWAsML/RiWAsDM can be utilized for n-tier RiWAs with the features beyond the specified scope limitations. Once the RiWAs engineers start utilising the RiWAsDM in real-world projects, the factual requirements for integrating the RiWAsML's models with standard UML models and expansions over the scope limitations could be precisely

identified. In parallel, studies can be conducted to understand the requirements for support designing the RiWAs with advanced functionalities with AI/ML/big-data-related features.

## 10.6. Chapter Summary

This chapter concludes the thesis by discussing the following and delivering some concluding knowledge.

- The final status of the research artefacts below is verified by reasoning.
  - **Research objectives:** It is shown that all the research objectives are strongly fulfilled.
  - **Research hypotheses:** All the research hypotheses are effectively proven.
  - **Research aim:** It is discussed that the aim of the research was successfully achieved.
- The following research contributions are explained.
  - A design methodology implementation process.
  - A new Rich Web-based Applications Modelling Language named RiWAsML.
  - A new UML extension for the RiWAsML.
  - A new design methodology for the RiWAs named RiWAsDM.
  - Demonstration of the use of the RiWAsML/RiWAsDM through real-world use cases.
- The challenges faced while working on the following research artefacts are mentioned.
  - Implementing the research methodology.
  - Implementing the RiWAsML.
  - Selecting examples and use cases.
  - Implementing the RiWAsDM.
  - Evaluating the RiWAsML and RiWAsDM.
- Identified limitations under the following aspects are discussed.
  - Single-paged RiWAs.
  - Integration with UML notations.
  - Language specification.
  - CASE tool and exchange format.
- Possible future work to follow the work in this thesis is acknowledged.
  - Complete RiWAsML specification.
  - Implement a CASE tool and provide XMI.
  - Study integrating UML meta-model diagrams and elements with RiWAsML.
  - Expand the scope of the RiWAsML by addressing the scope limitations.
  - Provide more use cases.

## References

---

- [1] L. Frankowski, *The Cross-Time Engineer*, Del Rey/Ballantine, 1986.
- [2] G. Engels and S. Sauer, "A Meta-Method for Defining Software Engineering Methods," *Graph Transformations and Model-Driven Engineering*, vol. 5765, p. 411–440, 2010.
- [3] E. Freeman, *DevOps For Dummies*, John Wiley & Sons, Inc., 2019.
- [4] P. P. Dingare, *CI/CD Pipeline Using Jenkins Unleashed: Solutions While Setting Up CI/CD Processes*, Apress, 2022.
- [5] K. Beck, M. Beedle, A. Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. Martin, S. Mellor, K. Schwaber, J. Sutherland and D. Thomas, "Manifesto for Agile Software Development," 2001. [Online]. Available: <http://agilemanifesto.org/>. [Accessed 20 Feb 2024].
- [6] I. The Institute of Electrical and Electronics Engineers, "IEEE Standard Glossary of Software Engineering Terminology," 2002.
- [7] OMG, "ABOUT THE UNIFIED MODELING LANGUAGE SPECIFICATION VERSION 2.5.1," Object Management Group, 2023. [Online]. Available: <https://www.omg.org/spec/UML/2.5.1/About-UML>. [Accessed 04 12 2023].
- [8] M. Brambilla, J. Cabot and M. Wimmer, *Model-Driven Software Engineering in Practice*, Morgan & Claypool Publishers, 2012.
- [9] L. Fuentes and A. Vallecillo, "An introduction to UML profiles," *UPGRADE The European Journal for the Informatics Professional*, vol. V, no. 2, pp. 6-13, 2004.
- [10] N. R. Dissanayake and K. Dias, "Rich Web-based Applications: An Umbrella Term with a Definition and Taxonomies for Development Techniques and Technologies," *International Journal of Future Computer and Communication*, vol. 7, no. 1, pp. 14-20, 2018.
- [11] N. R. Dissanayake and G. Dias, "Delta Communication: The Power of the Rich Internet Applications," *International Journal of Future Computer and Communication*, vol. 6, no. 2, pp. 31-36, 2017.
- [12] N. R. Dissanayake and K. Dias, "RiWAArch Style: An Architectural style for Rich Web-based Applications," in *Proceedings of the 2020 Future Technologies Conference (FTC)*, Canada, 2020.



## References

- [13] N. R. Dissanayake and G. K. A. Dias, "Abstract concepts: A contemporary requirement for Rich Internet Applications engineering," in *9th International Research Conference of KDU (KDU-IRC 9)*, Colombo, Sri Lanka, 2016.
- [14] R. Rodríguez-Echeverría, "Ria: more than a nice face," in *Proceedings of the Doctoral Consortium of the International Conference on Web Engineering*, 2009.
- [15] M. Busch and N. Koch, "Rich Internet Applications - State-of-the-Art," Ludwig-Maximilians-Universität, München, 2009.
- [16] J. Preciado, M. Linaje, F. Sanchez and S. Comai, "Necessity of methodologies to model Rich Internet Applications," in *Proceedings of the 2005 Seventh IEEE International Symposium on Web Evolution*, 2005.
- [17] U. Frank, "Outline of a Method for Designing Domain-Specific Modelling Languages," *ICB-Research Report*, vol. 42, pp. 1-62, 2010.
- [18] P. A. Laplante, *WHAT EVERY ENGINEER SHOULD KNOW ABOUT SOFTWARE ENGINEERING*, CRC Press, 2007.
- [19] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," University of California, Irvine, 2000.
- [20] D. Budgen, *SOFTWARE DESIGN*, Essex, England: Pearson Education Limited, 2003.
- [21] N. R. Dissanayake and G. K. A. Dias, "Essential Features a General AJAX Rich Internet Application Architecture Should Have in Order to Support Rapid Application Development," *International Journal of Future Computer and Communication*, vol. 3, no. 5, pp. 350-353, 2014.
- [22] M. Linaje, J. C. Preciado and F. Sanchez-Figueroa, "Engineering Rich Internet Application User Interfaces over Legacy Web Models," *Internet Computing, IEEE*, vol. 11, no. 6, pp. 53-59, November-December 2007.
- [23] UWE, "UWE – UML-based Web Engineering," UWE, 10 Aug 2016. [Online]. Available: <http://uwe.pst.ifi.lmu.de/index.html>. [Accessed 20 Feb 2024].
- [24] J. C. Preciado, M. Linaje, R. Morales-Chaparro, F. Sanchez-Figueroa, G. Zhang, C. Kroiß and N. Koch, "Designing Rich Internet Applications Combining UWE and RUX-Method," in *Eighth International Conference on Web Engineering*, 2008.

## References

- [25] H. Koning, C. Dormann and H. v. Vliet, "Practical guidelines for the readability of IT-architecture diagrams," in *SIGDOC '02: Proceedings of the 20th annual international conference on Computer documentation*, 2002.
- [26] M. Petre, "UML in Practice," in *ICSE '13: Proceedings of the 2013 International Conference on Software Engineering*, 2013.
- [27] H. Zuse, *Software Complexity Measures and Models*, New York: de Gruyter & Co., 1992.
- [28] Wikipedia, "Complex system," [Online]. Available: [https://en.wikipedia.org/wiki/Complex\\_system](https://en.wikipedia.org/wiki/Complex_system). [Accessed 20 Feb 2024].
- [29] J. M. SUSSMAN, *The New Transportation Faculty: The Evolution to Engineering Systems*, 1999.
- [30] J. C. Pereira and R. Russo, "Design Thinking Integrated in Agile Software Development: A Systematic Literature Review," *Procedia Computer Science*, vol. 138, pp. 775-782, 2018.
- [31] D. G. Gregg, U. R. Kulkarni and A. S. Vinzé, "Understanding the Philosophical Underpinnings of Software Engineering Research in Information Systems," *Information Systems Frontiers*, vol. 3, no. 2, pp. 169-183, 2001.
- [32] P. Gu, M. Hashemian, S. Sosale and E. Rivin, "An Integrated Modular Design Methodology for Life-Cycle Engineering," *CIRP Annals*, vol. 46, no. 1, pp. 71-74, 1997.
- [33] S. Kumar, A. Jantsch, J.-P. Soinen and M. Forsell, "A Network on Chip Architecture and Design Methodology," in *Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI'02)*, 2002.
- [34] S. Ahmed, S. Kim and K. M. Wallace, "A Methodology for Creating Ontologies for Engineering Design," *Transactions of the ASME*, vol. 7, no. 2, pp. 132-140, 2007.
- [35] S. Meliá, J. Gómez, S. Pérez and O. Díaz, "A Model-Driven Development for GWT-Based Rich Internet Applications with OOH4RIA," in *Eighth International Conference on Web Engineering*, 2008.
- [36] F. Rademacher, S. Sachweh and A. Zündorf, "Modeling Method for Systematic Architecture Reconstruction of Microservice-Based Software Systems," *Enterprise, Business-Process and Information Systems Modeling*, p. 311–326, 2020.

## References

- [37] B. G. Assefa and Ö. Özkasap, "RES DN: A Novel Metric and Method for Energy Efficient Routing in Software Defined Networks," *IEEE Transactions on Network and Service Management*, vol. 17, no. 2, pp. 736 - 749, Jun 2020.
- [38] O. Aldawud, T. Elrad and A. Bader, "UML profile for aspect-oriented software development," in *Proceedings of Third International Workshop on Aspect-Oriented Modeling*, 2003.
- [39] S. Graf, I. Ober and I. Ober, "A real-time profile for UML," *International Journal on Software Tools for Technology Transfer*, vol. 8, p. pages113–127, 2006.
- [40] A. Gómez, J. Merseguer, E. D. Nitto and D. A. Tamburri, "Towards a UML profile for data intensive applications," in *QUDOS 2016: Proceedings of the 2nd International Workshop on Quality-Aware DevOps*, 2016.
- [41] H. Marouane, C. Duvallet, A. Makni, R. Bouaziz and B. Sadeg, "An UML profile for representing real-time design patterns," *Journal of King Saud University - Computer and Information Sciences*, vol. 30, no. 4, pp. 478-497, 2018.
- [42] J. M. Wright, "A Modelling Language for Rich Internet Applications," Massey University, Turitea, New Zealand, 2011.
- [43] uml-diagrams.org, "UML Diagrams Examples," uml-diagrams.org, 2024. [Online]. Available: <https://www.uml-diagrams.org/index-examples.html>. [Accessed 20 Feb 2024].
- [44] arc42.org, "144 tips and 33 examples how to use the arc42 template," arc42.org, [Online]. Available: <https://docs.arc42.org/examples/>. [Accessed 20 Feb 2024].
- [45] SAP, Object-Oriented Architecture - SAP PowerDesigner Documentation Collection, 16.7.07 – 2023-05-29 ed., SAP, 2023.
- [46] visual-paradigm, "CH 12 – ArchiMate: Learn by Examples," visual-paradigm, [Online]. Available: <https://archimate.visual-paradigm.com/category/archimate-concepts/ch-11-archimate-learn-by-examples/>. [Accessed 20 Feb 2024].
- [47] SysML.org, "SysML Diagram Tutorial," SysML.org, 2024. [Online]. Available: <https://sysml.org/tutorials/sysml-diagram-tutorial/>. [Accessed 20 Feb 2024].
- [48] V. Pattyn, A. Molenveld and B. Befani, "Qualitative Comparative Analysis as an Evaluation Tool: Lessons From an Application in Development Cooperation," *American Journal of Evaluation*, vol. 40, no. 1, pp. 55-74, 2017.

## References

- [49] CECAN, "Qualitative Comparative Analysis: a pragmatic method for evaluating intervention," CECAN, 2016.
- [50] A. Mesbah and A. v. Deursen, "An Architectural Style for AJAX," in *Software Architecture, 2007. WICSA '07. The Working IEEE/IFIP Conference*, Mumbai, 2007.
- [51] D. Renmans and V. C. Pleguezuelo, "Methods in realist evaluation: A mapping review," *Evaluation and Program Planning*, vol. 97, 2023.
- [52] F. C. Mukumbang, B. Marchal, S. V. Belle and B. v. Wyk, "Using the realist interview approach to maintain theoretical awareness in realist studies," *Qualitative Research*, vol. 20, no. 4, p. 485–515, 2020.
- [53] J. M. Morse, "Data Were Saturated . . .," *Qualitative Health Research*, vol. 25, no. 5, pp. 587-588, 2015.
- [54] S. Elsie, Baker and R. Edwards, "How many qualitative interviews is enough," National Centre for Research Methods (NCRM), 2021.
- [55] B. Randell, "SOFTWARE ENGINEERING IN 1968," in *Proc. of the 4th Int. Conf. on Software*, Munich, 1979.
- [56] M. S. Mahoney, "The roots of software engineering," *CWI Quarterly*, vol. 3, pp. 325-334, 1990.
- [57] A. Peck, "Software Development Glossary: 88 Essential Terms," Clutch, 29 Mar 2023. [Online]. Available: <https://clutch.co/resources/software-development-glossary-88-essential-terms>. [Accessed 20 Feb 2024].
- [58] E. Mnkandla, "About Software Engineering Frameworks and Methodologies," in *IEEE AFRICON 2009*, Nairobi, Kenya, 2009.
- [59] R. S. Pressman, *Software Engineering A PRACTITIONER ' S APPROACH*, New York, United States: McGraw-Hill, Inc., 2009.
- [60] Scrum.org, "Welcome to the Home of Scrum!™," Scrum.org, 2024. [Online]. Available: <https://www.scrum.org/>. [Accessed 20 Feb 2024].
- [61] K. Schwaber and J. Sutherland, "The Scrum Guide," Scrum.org, 2020.
- [62] Prince2, "PRINCE2 Methodology," Prince2, 2024. [Online]. Available: <https://www.prince2.com/uk/prince2-methodology>. [Accessed 20 Feb 2024].

## References

- [63] H. Zhu, *Software Design Methodology: From Principles to Architectural Styles*, Butterworth-Heinemann, 2005.
- [64] T. Kapteijns, S. Jansen, S. Brinkkemper, H. Houët and R. Barendse, "A Comparative Case Study of Model Driven Development vs Traditional Development: The Tortoise or the Hare," in *4th European Workshop on "From code centric to model centric software engineering: Practices, Implications and ROI"*, 2009.
- [65] S. Al-Saqqa, S. Sawalha and H. AbdelNabi, "Agile Software Development: Methodologies and Trends," *International Journal of Interactive Mobile Technologies*, vol. 14, no. 11, pp. 246-269, 2020.
- [66] "Agile Methodology: Advantages and Disadvantages," University of Minnesota, 11 Feb 2022. [Online]. Available: <https://ccaps.umn.edu/story/agile-methodology-advantages-and-disadvantages>. [Accessed 20 Feb 2024].
- [67] T. Tran, "The Desirable Benefits of Agile Methodology in Software Development," Orient, 16 Jun 2022. [Online]. Available: <https://www.orientsoftware.com/blog/benefits-of-agile-methodology/>. [Accessed 20 Feb 2024].
- [68] Segue-Technologies, "8 Benefits of Agile Software Development," Segue Technologies, 25 Aug 2015. [Online]. Available: <https://www.seguetech.com/8-benefits-of-agile-software-development/>. [Accessed 20 Feb 2024].
- [69] Kissflow, "The 9 Key Benefits of Using the Agile Methodology," Kissflow, 21 Dec 2023. [Online]. Available: <https://kissflow.com/project/agile/benefits-of-agile/>. [Accessed 20 Feb 2024].
- [70] agilemanifesto.org, "Principles behind the Agile Manifesto," agilemanifesto.org, 2001. [Online]. Available: <https://agilemanifesto.org/principles.html>. [Accessed 20 Feb 2024].
- [71] J. D. Haan, "Why there is no future for Model Driven Development," 25 Jan 2011. [Online]. Available: <http://www.theenterprisearchitect.eu/blog/2011/01/25/why-there-is-no-future-for-model-driven-development/>. [Accessed 20 Feb 2024].
- [72] B. Ross, "Can model-driven architecture be used on Agile development projects?," 24 Nov 2015. [Online]. Available: <https://www.equinox.co.nz/blog/model-driven-architecture-on-agile-development-projects>. [Accessed 20 Feb 2024].
- [73] G. Tremblay, "CHAPTER 3 SOFTWARE DESIGN," 2002.
- [74] P. Bourque and R. Fairley, "SWEBOK v3.0," IEEE, 2014.

## References

- [75] OMG, "Model Driven Architecture (MDA) - MDA Guide rev. 2.0," OMG, 2014.
- [76] D. Hough, "Rapid Delivery: An evolutionary approach for application development," *IBM SYSTEM JOURNAL*, vol. 32, no. 3, pp. 397-419, 1993.
- [77] D. M. Selfa, M. Carrillo and M. d. R. Boone, "A Database and Web Application Based on MVC Architecture," in *Electronics, Communications and Computers, 2006. CONIELECOMP 2006. 16th International Conference, 2006*.
- [78] D. Garlan, "Software architecture: a travelogue," in *FOSE 2014: Future of Software Engineering Proceedings*, Hyderabad India, 2014.
- [79] S. Brown, "The C4 model for visualising software architecture," <https://simonbrown.je/>, 2023. [Online]. Available: <https://c4model.com/>. [Accessed 20 Feb 2024].
- [80] P. Hruschka and G. Starke, "arc42," arc42, 2024. [Online]. Available: <https://arc42.org>. [Accessed 20 Feb 2024].
- [81] iso-architecture.org, "Welcome to the ISO/IEC/IEEE 42010 Website," iso-architecture.org, 18 Apr 2023. [Online]. Available: <http://www.iso-architecture.org/42010/index.html>. [Accessed 20 Feb 2024].
- [82] SAP, Standardized Technical Architecture Modeling - Conceptual and Design Level, SAP, 2007.
- [83] J. Ingeno, Software Architect's Handbook, Packt Publishing, 2018.
- [84] N. Koch, A. Knapp, G. Zhang and H. Baumeister, "UML-BASED WEB ENGINEERING - An Approach Based on Standards," in *Web Engineering: Modelling and Implementing Web Applications*, Springer, 2008, pp. 157-191.
- [85] O. 2. E. Team, "Meta-Modeling and the OMG Meta Object Facility (MOF)," OMG, 2017.
- [86] J. Pearce, "The UML Meta-Model," [Online]. Available: <https://www.cs.sjsu.edu/~pearce/modules/lectures/uml2/index.htm>. [Accessed 20 Feb 2024].
- [87] IBM, "Exchanging model data by using XMI," IBM, 20 Feb 2024. [Online]. Available: <https://www.ibm.com/docs/en/engineering-lifecycle-management-suite/design-rhapsody/9.0.2?topic=tools-exchanging-model-data-by-using-xmi>. [Accessed 20 Feb 2024].
- [88] OMG, "XML Metadata Interchange (XMI) Specification," 2015.

## References

- [89] OMG, "OMG PROCESS INTRODUCTION," OMG, 2024. [Online]. Available: <https://www.omg.org/gettingstarted/processintro.htm>. [Accessed 20 Feb 2024].
- [90] OMG, "POPULAR OMG STANDARDS," OMG, 2024. [Online]. Available: <https://www.omg.org/about/omg-standards-introduction.htm>. [Accessed 20 Feb 2024].
- [91] I. Malavolta, H. Muccini and M. Sebastiani, "Automatically bridging UML profiles to MOF Metamodels," in *41st Euromicro Conference on Software Engineering and Advanced Applications*, Funchal, Portugal, 2015.
- [92] uml-diagrams.org, "UML Stereotype," 2024. [Online]. Available: <https://www.uml-diagrams.org/stereotype.html>. [Accessed 20 Feb 2024].
- [93] uml-diagrams.org, "UML, Meta Meta Models and Profiles," 2024. [Online]. Available: <https://www.uml-diagrams.org/uml-meta-models.html>. [Accessed 20 Feb 2024].
- [94] uml-diagrams.org, "UML Profile," 2024. [Online]. Available: <https://www.uml-diagrams.org/profile.html>. [Accessed 20 Feb 2024].
- [95] uml-diagrams.org, "UML Profile Diagrams," 2024. [Online]. Available: <https://www.uml-diagrams.org/profile-diagrams.html>. [Accessed 20 Feb 2024].
- [96] M. v. Steen and A. S. Tanenbaum, *Distributed Systems*, 2017.
- [97] W3C, "Architecture of the World Wide Web, Volume One," 15 Dec 2004. [Online]. Available: <http://www.w3.org/TR/webarch/>. [Accessed 20 Feb 2024].
- [98] N. R. Dissanayake and G. Dias, "Web-based Applications: Extending the General Perspective of the Service of Web," in *10th International Research Conference of KDU (KDU-IRC 2017) on Changing Dynamics in the Global Environment: Challenges and Opportunities*, Rathmalana, Sri Lanka, 2017.
- [99] J. J. Garrett, "Ajax: A New Approach to Web Applications," adaptive path, 18 February 2005. [Online]. Available: [https://designftw.mit.edu/lectures/apis/ajax\\_adaptive\\_path.pdf](https://designftw.mit.edu/lectures/apis/ajax_adaptive_path.pdf). [Accessed 20 Feb 2024].
- [100] W3C, "XMLHttpRequest Level 1," 06 Oct 2016. [Online]. Available: <http://www.w3.org/TR/2014/WD-XMLHttpRequest-20140130/>. [Accessed 20 Feb 2024].
- [101] J. Allaire, "Macromedia Flash MX—A next-generation rich client," Macromedia, San Francisco, 2002.

## References

- [102] N. R. Dissanayake and G. Dias, "A Comparison of Delta-Communication Technologies and Techniques," in *10th International Research Conference of KDU (KDU-IRC 2017) on Changing Dynamics in the Global Environment: Challenges and Opportunities*, Rathmalana, Sri Lanka, 2017.
- [103] I. Fette, Google, Inc., A. Melnikov and Isode Ltd., "The WebSocket Protocol," Internet Engineering Task Force, 2011.
- [104] T. J. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, Vols. SE-2, no. 4, pp. 308-320, 1979.
- [105] N. R. Dissanayake and K. Dias, "Balanced Abstract Web-MVC Style: An Abstract MVC Implementation for Web-based Applications," *GSTF Journal on Computing*, vol. 5, no. 3, pp. 27-41, 2017.
- [106] A. Ginige and S. Murugesan, "Web engineering: an introduction," *IEEE MultiMedia*, vol. 8, no. 1, pp. 14-18, 2001.
- [107] WebML.org, "The web modeling language," WebML.org, [Online]. Available: <http://www.webml.org>. [Accessed 27 05 2018].
- [108] A. Mesbah and A. v. Deursen, "A component and push-based architectural style for AJAX applications," *The Journal of Systems and Software*, vol. 81, no. 12, p. 2194–2209, 2008.
- [109] J. Li and C. Peng, "jQuery-based Ajax General Interactive Architecture," in *Software Engineering and Service Science (ICSESS), 2012 IEEE 3rd International Conference*, Beijing, 2012.
- [110] A. Avritzer, D. Paulish, Y. Cai and K. Sethi, "Coordination implications of software architecture in a global software development project," *The Journal of Systems and Software*, vol. 83, no. 10, pp. 1881-1895, 2010.
- [111] M. Ozkaya and C. Kloukinas, "Are We There Yet? Analyzing Architecture Description Languages for Formal Analysis, Usability, and Realizability," in *2013 39th Euromicro Conference on Software Engineering and Advanced Applications*, Santander, Spain, 2013.
- [112] University of California, "xADL3.0," University of California, [Online]. Available: <http://isr.uci.edu/projects/xarchuci/index.html>. [Accessed 20 Feb 2024].
- [113] E. M. Dashofy, A. v. d. Hoek and R. N. Taylor, "An Infrastructure for the Rapid Development of XML-based Architecture Description Languages," in *Proceedings of the 24th International Conference on Software Engineering (ICSE2002)*, Orlando, Florida, 2002.



## References

- [114] ABLE, "The Acme Project," Carnegie Mellon University, 2011. [Online]. Available: <https://www.cs.cmu.edu/~acme/index.html>. [Accessed 20 Feb 2024].
- [115] B. Schmerl, "xAcme: CMU Acme Extensions to xArch," Carnegie Mellon University, 2001.
- [116] P. h. Feiler, D. p. Gluch and J. J. Hudak, The Architecture Analysis & Design Language (AADL): An Introduction, USA: Carnegie Mellon University, 2006.
- [117] Carnegie-Mellon-University, "Architecture Analysis and Design Language (AADL)," Carnegie Mellon University, Feb 2022. [Online]. Available: [https://www.sei.cmu.edu/our-work/projects/display.cfm?customel\\_datapageid\\_4050=191439,191439](https://www.sei.cmu.edu/our-work/projects/display.cfm?customel_datapageid_4050=191439,191439). [Accessed 20 Feb 2024].
- [118] OMG, "MISSION & VISION," OMG, 2024. [Online]. Available: <https://www.omg.org/about/index.htm>. [Accessed 20 Feb 2024].
- [119] OMG, "Unified Modeling Language 2.5.1," OMG Unified Modeling Language, 2017.
- [120] P. Hruschka and G. Starke, "Download arc42," arc42, 202. [Online]. Available: <https://arc42.org/download>. [Accessed 20 Feb 202].
- [121] The Open Group, "ArchiMate® 3.2 Specification," The Open Group, 03 Jan 2023. [Online]. Available: <https://pubs.opengroup.org/architecture/archimate32-doc/>. [Accessed 20 Feb 2024].
- [122] M. Lankhorst and ArchiMate team, "ArchiMate Language Primer," TELEMATICA INSTITUUT, 2004.
- [123] L. MACHADO, O. FILHO and J. RIBEIRO, "UWE-R: an extension to a Web Engineering methodology for Rich Internet Applications," *WSEAS TRANSACTIONS on INFORMATION SCIENCE and APPLICATIONS*, vol. 6, no. 4, pp. 601-610, 2009.
- [124] I. Object Management Group, "IFML: The Interaction Flow Modeling Language," Object Management Group, Inc, 2018. [Online]. Available: <http://www.ifml.org>. [Accessed 20 Feb 2024].
- [125] OMG, "ABOUT THE INTERACTION FLOW MODELING LANGUAGE SPECIFICATION VERSION 1.0," OMG, 2023. [Online]. Available: <https://www.omg.org/spec/IFML>. [Accessed 17 10 2023].

## References

- [126] SysML.org, "SysML Open Source Project - What is SysML? Who created SysML?," PivotPoint Technology Corp., 2024. [Online]. Available: <https://sysml.org/>. [Accessed 20 Feb 2024].
- [127] OMG, "OMG Systems Modeling Language version 1.6," OMG SysML, 2018.
- [128] OMG, "OMG Systems Modeling Language Version 2.0 Beta 1," OMG, 2023.
- [129] J. Conallen, "Modeling Web Application Architectures with UML," *Communications of the ACM*, vol. 42, no. 10, pp. 63-70, 1999.
- [130] R. Hennicker and N. Koch, "Systematic Design of Web Applications with UML," in *Unified Modeling Language: Systems Analysis, Design and Development Issues*, Idea Group Publishing, 2001, pp. 1-20.
- [131] R. Hennicker and N. Koch, "Modeling the User Interface of Web Applications with UML," in *Practical UML-Based Rigorous Development Methods - Countering or Integrating the eXtremists, Workshop of the pUML-Group held together with the «UML»*, 2001.
- [132] V. E. S. Souza, R. d. A. Falbo and G. Guizzardi, "A UML Profile for Modeling Framework-based Web Information Systems," in *Proc. of the 12th International Workshop on Exploring Modeling Methods in Systems Analysis and Design*, 2007.
- [133] P. Dolog and J. Stage, "Designing Interaction Spaces for Rich Internet Applications with UML," in *7th International Conference Web Engineering, ICWE 2007*, Como, Italy, 2007.
- [134] J. Gómez and C. Cachero, "Chapter VIII," in *OO-H Method: Extending UML to Model Web Interfaces*, 2003, pp. 144-173.
- [135] T. Černý and E. Song, "A Profile Approach to Using UML Models for Rich Form Generation," in *International Conference on Information Science and Applications (ICISA)*, 2010.
- [136] S. A. Mubin and A. H. Jantan, "A UML 2.0 profile web design framework for modeling complex web application," in *Proceedings of the 6th International Conference on Information Technology and Multimedia*, Putrajaya, Malaysia, 2014.
- [137] uml-diagrams.org, "Node," uml-diagrams.org, 2024. [Online]. Available: <https://www.uml-diagrams.org/deployment-diagrams.html#node>. [Accessed 20 Feb 2024].
- [138] M. Richards, *Software Architecture Patterns*, O'Reilly Media, Inc., 2022.

## References

- [139] Laravel, "Controllers," Laravel, 2023. [Online]. Available: <https://laravel.com/docs/10.x/controllers>. [Accessed 20 Feb 2024].
- [140] M. Model, "Model View Controller History," 26 Dec 2014. [Online]. Available: <http://c2.com/cgi/wiki?ModelViewControllerHistory>. [Accessed 20 Feb 2024].
- [141] S. Burbeck, *Applications Programming in Smalltalk-80™: How to use Model-View-Controller (MVC)*, Softsmarts, Incorporated, 1987.
- [142] UWE, "Tutorial - Presentation Model," UWE, 10 Aug 2016. [Online]. Available: <https://uwe.pst.ifi.lmu.de/teachingTutorialPresentation.html>. [Accessed 20 Feb 2024].
- [143] J. Preciado, M. Linaje, F. Sanchez and S. Comai, "Necessity of methodologie to model Rich Internet Applications," in *Proceedings of the 2005 Seventh IEEE International Symposium on Web Evolution*, 2005.
- [144] W3C, "SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)," 27 Apr 2007. [Online]. Available: <https://www.w3.org/TR/soap12/>. [Accessed 20 Feb 2024].
- [145] L. Gupta, "REST API URI Naming Conventions and Best Practices," 04 Nov 2023. [Online]. Available: <https://restfulapi.net/resource-naming/>. [Accessed 20 Feb 2024].
- [146] uml-diagrams.org, "UML Deployment Diagrams," uml-diagrams.org, 2024. [Online]. Available: <https://www.uml-diagrams.org/deployment-diagrams.html>. [Accessed 20 Feb 2024].
- [147] UWE, "Tutorial - Navigation Model," UWE, 10 Aug 2016. [Online]. Available: <https://uwe.pst.ifi.lmu.de/teachingTutorialNavigation.html>. [Accessed 20 Feb 2024].
- [148] uml-diagrams.org, "Multi-Layered Web Architecture," uml-diagrams.org, 2024. [Online]. Available: <https://www.uml-diagrams.org/multi-layered-web-architecture-uml-package-diagram-example.html>. [Accessed 20 Feb 2024].
- [149] uml-diagrams.org, "Deployment Diagrams Overview," uml-diagrams.org, 2024. [Online]. Available: <https://www.uml-diagrams.org/deployment-diagrams-overview.html>. [Accessed 20 Feb 2024].
- [150] Visual-Paradigm, "ArchiMate Notation: Part 3 – Technology Layer," Visual-Paradigm, 20 Feb 2018. [Online]. Available: <https://archimate.visual-paradigm.com/archimate-notation-part-3-technology-layers/>. [Accessed 20 Feb 2024].

## References

- [151] uml-diagrams.org, "UML Artifact," uml-diagrams.org, 2024. [Online]. Available: <https://www.uml-diagrams.org/artifact.html>. [Accessed 20 Feb 2024].
- [152] Visual-Paradigm, "ArchiMate Notation: Part 1 – Business Layer," Visual-Paradigm, 21 Feb 2018. [Online]. Available: <https://archimate.visual-paradigm.com/archimate-notation-part-1-business-layers/>. [Accessed 20 Feb 2024].
- [153] uml-diagrams.org, "Component," uml-diagrams.org, 2024. [Online]. Available: <https://www.uml-diagrams.org/component.html>. [Accessed 20 Feb 2024].
- [154] J. Pearce, "The Entity-Control-Boundary Pattern," [Online]. Available: <https://www.cs.sjsu.edu/~pearce/modules/lectures/ooa/analysis/ecb.htm>. [Accessed 20 Feb 2024].
- [155] Wikipedia, "Entity-control-boundary," Wikipedia, [Online]. Available: <https://en.wikipedia.org/wiki/Entity-control-boundary>. [Accessed 20 Feb 2024].
- [156] IBM, "How to display the Entity, Boundary, Control stereotypes on class diagrams," IBM, 10 Sep 2020. [Online]. Available: <https://www.ibm.com/support/pages/how-display-entity-boundary-control-stereotypes-class-diagrams>. [Accessed 20 Feb 2024].
- [157] uml-diagrams.org, "Namespace," uml-diagrams.org, 2024. [Online]. Available: <https://www.uml-diagrams.org/namespace.html>. [Accessed 20 Feb 2024].
- [158] uml-diagrams.org, "UML Package Diagrams Notation," uml-diagrams.org, 2024. [Online]. Available: <https://www.uml-diagrams.org/package-diagrams.html>. [Accessed 20 Feb 2024].
- [159] uml-diagrams.org, "UML Classifier," uml-diagrams.org, 2024. [Online]. Available: <https://www.uml-diagrams.org/classifier.html>. [Accessed 20 Feb 2024].
- [160] uml-diagrams.org, "Class," uml-diagrams.org, 2024. [Online]. Available: <https://www.uml-diagrams.org/class.html>. [Accessed 20 Feb 2024].
- [161] uml-diagrams.org, "Information Flow Elements," uml-diagrams.org, 2014. [Online]. Available: <https://www.uml-diagrams.org/information-flow-elements.html#information-flow>. [Accessed 20 Feb 2024].
- [162] uml-diagrams.org, "Object Flow Edge," uml-diagrams.org, 2024. [Online]. Available: <https://www.uml-diagrams.org/activity-diagrams.html#object-flow-edge>. [Accessed 20 Feb 2024].

## References

- [163] uml-diagrams.org, "UML Message," uml-diagrams.org, 2024. [Online]. Available: <https://www.uml-diagrams.org/interaction-message.html>. [Accessed 20 Feb 2024].
- [164] uml-diagrams.org, "Communication Path," uml-diagrams.org, 2024. [Online]. Available: <https://www.uml-diagrams.org/deployment-diagrams.html#communication-path>. [Accessed 20 Feb 2024].
- [165] uml-diagrams.org, "UML Component Diagrams," uml-diagrams.org, 2024. [Online]. Available: <https://www.uml-diagrams.org/component-diagrams.html>. [Accessed 20 Feb 2024].
- [166] uml-diagrams.org, "UML Composite Structure Diagrams," uml-diagrams.org, 2024. [Online]. Available: <https://www.uml-diagrams.org/composite-structure-diagrams.html>. [Accessed 20 Feb 2024].
- [167] M. González, J. Casariego, J. J. Bareiro, L. Cernuzzi and O. Pastor, "A MDA Approach for Navigational and User Perspectives," *CLEI Electronic Journal*, vol. 14, no. 1, pp. 1-12, 2011.
- [168] Mozilla, "Using the Fetch API," Mozilla, 18 Aug 2023. [Online]. Available: [https://developer.mozilla.org/en-US/docs/Web/API/Fetch\\_API/Using\\_Fetch](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch). [Accessed 20 Feb 2024].
- [169] jQuery, "jQuery.ajax()," jQuery, [Online]. Available: <https://api.jquery.com/jQuery.ajax/>. [Accessed 20 Feb 2024].
- [170] Mozilla, "Express Tutorial Part 3: Using a Database (with Mongoose)," Mozilla, 20 Feb 2024. [Online]. Available: [https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express\\_Nodejs/mongoose#designing\\_the\\_locallibrary\\_models](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/mongoose#designing_the_locallibrary_models). [Accessed 20 Feb 2024].
- [171] Laravel, "Eloquent: Getting Started," Laravel, [Online]. Available: <https://laravel.com/docs/10.x/eloquent>. [Accessed 20 Feb 2024].
- [172] Laravel, "Routing," Laravel, [Online]. Available: <https://laravel.com/docs/10.x/routing>. [Accessed 20 Feb 2024].
- [173] Mozilla, "Express Tutorial Part 4: Routes and controllers," Mozilla, 18 Oct 2023. [Online]. Available: [https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express\\_Nodejs/routes](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/routes). [Accessed 20 Feb 2024].

## References

- [174] Spring, "Mapping Requests," Spring, [Online]. Available: <https://docs.spring.io/spring-framework/reference/web/webmvc/mvc-controller/ann-requestmapping.html>. [Accessed 20 Feb 2020].
- [175] PivotPoint-Technology, "Custom Agile MBSE™ + SysML Training & Certification," PivotPoint Technology, 2024. [Online]. Available: <https://pivotpt.com/training/mbse-sysml/>. [Accessed 20 Feb 2024].
- [176] TheScrumMaster.co.uk, "The Simple Guide To Scrum - 1 Pager," TheScrumMaster.co.uk, 2021.
- [177] Synopsys, "CI CD," Synopsys, 2024. [Online]. Available: <https://www.synopsys.com/glossary/what-is-cicd.html>. [Accessed 20 Feb 2024].
- [178] Apache, "Learning To Rank," Apache Software Foundation, 04 Nov 2020. [Online]. Available: [https://solr.apache.org/guide/8\\_7/learning-to-rank.html](https://solr.apache.org/guide/8_7/learning-to-rank.html). [Accessed 20 Feb 2024].
- [179] uml-diagrams.org, "Web Application - UML Deployment Diagram Example," uml-diagrams.org, 2024. [Online]. Available: <https://www.uml-diagrams.org/web-application-uml-deployment-diagram-example.html?context=depl-examples>. [Accessed 20 Feb 2024].
- [180] A. Ullah, H. Dagdeviren, R. C. Ariyattu, J. DesLauriers, T. Kiss and J. Bowden, "MiCADO-Edge: Towards an Application-level Orchestrator for the Cloud-to-Edge Computing Continuum," *Journal of Grid Computing*, vol. 19, no. 47, 2021.
- [181] MiCADO, "Scale virtual machines and containers at runtime," MiCADO, [Online]. Available: <https://micado-scale.eu/>. [Accessed 20 Feb 2024].
- [182] A. P. G. C. Y. F. D. W. D. I. G. T. C. M. W. W. T. K. A. a. Y. N. Bolotov, "SMARTTEST - knowledge and learning repository," University of Westminster, 2020. [Online]. Available: <https://westminsterresearch.westminster.ac.uk/item/v2x13/smartest-knowledge-and-learning-repository>. [Accessed 20 Feb 2024].
- [183] Salesforce, "Heroku," Salesforce, 2024. [Online]. Available: <https://www.heroku.com/>. [Accessed 20 Feb 2024].
- [184] MongoDB, "MongoDB," MongoDB, 2024. [Online]. Available: <https://www.mongodb.com/>. [Accessed 20 Feb 2024].
- [185] B. Moore, "BorisMoore/jquery-tmpl - jQuery Templates plugin vBeta1.0.0," GitHub, 2018. [Online]. Available: <https://github.com/BorisMoore/jquery-tmpl>. [Accessed 20 Feb 2024].

## References

- [186] visual-paradigm, "Harmonizing Horizons: The Seamless Integration of ArchiMate and TOGAF ADM for Comprehensive Enterprise Architecture," visual-paradigm, 12 Oct 2023. [Online]. Available: <https://archimate.visual-paradigm.com/2023/10/12/harmonizing-horizons-the-seamless-integration-of-archimate-and-togaf-adm-for-comprehensive-enterprise-architecture/>. [Accessed 20 Feb 2024].
- [187] visual-paradigm, "Bookmarks & Resources," visual-paradigm, [Online]. Available: <https://archimate.visual-paradigm.com/category/resources/>. [Accessed 20 Feb 2024].

# Appendices

## Appendix A. Example: Shopping App – Level 1+2 Architecture Diagram (large)

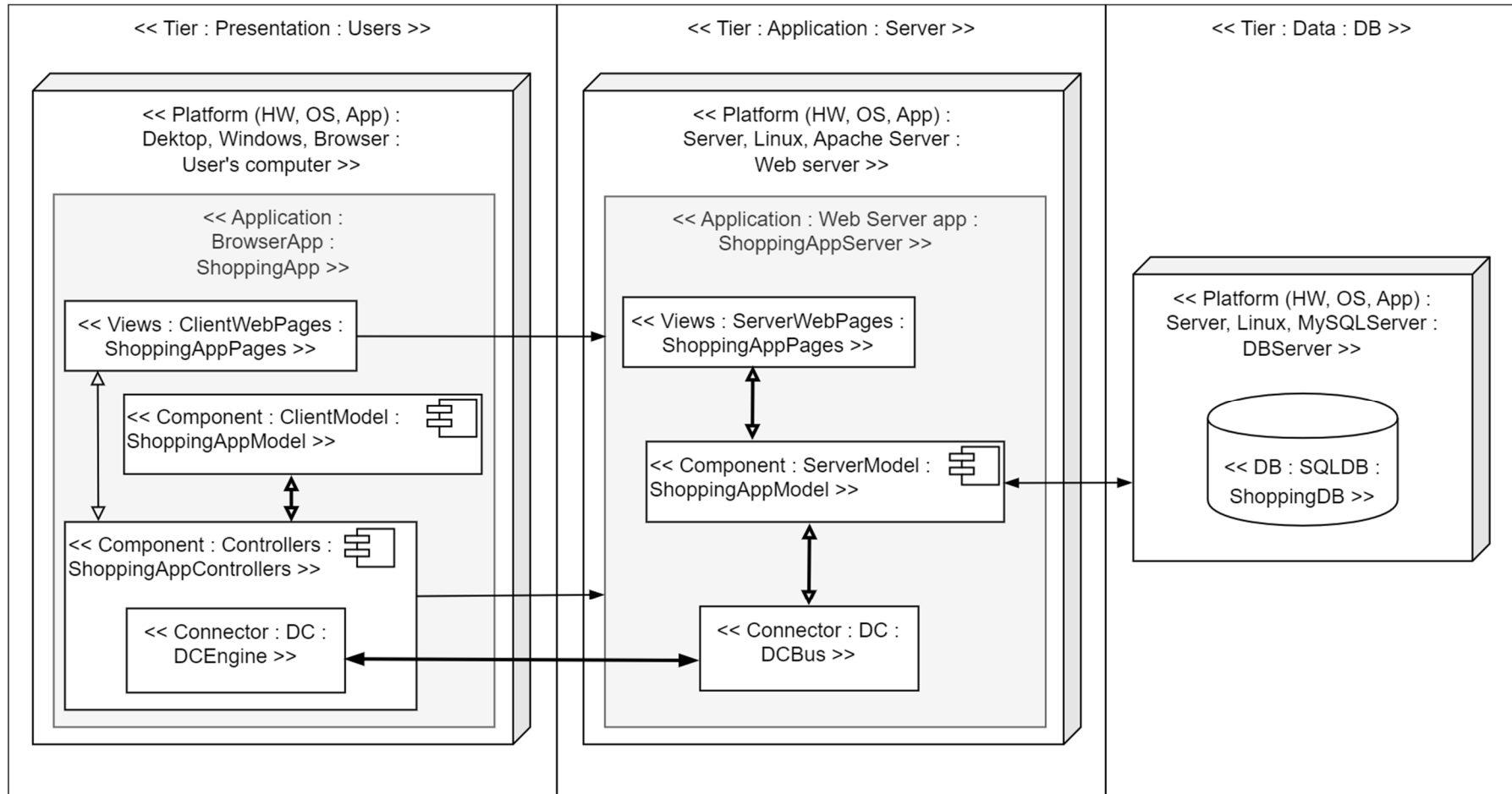


Figure Appendix A.1 Example: shopping app – level 1+2 architecture (large)



### Appendix B. Use Cases: High-level Designing with RiWAsML

#### Appendix B.1. Shopping System – Original Architecture Without Using Tiers (large)

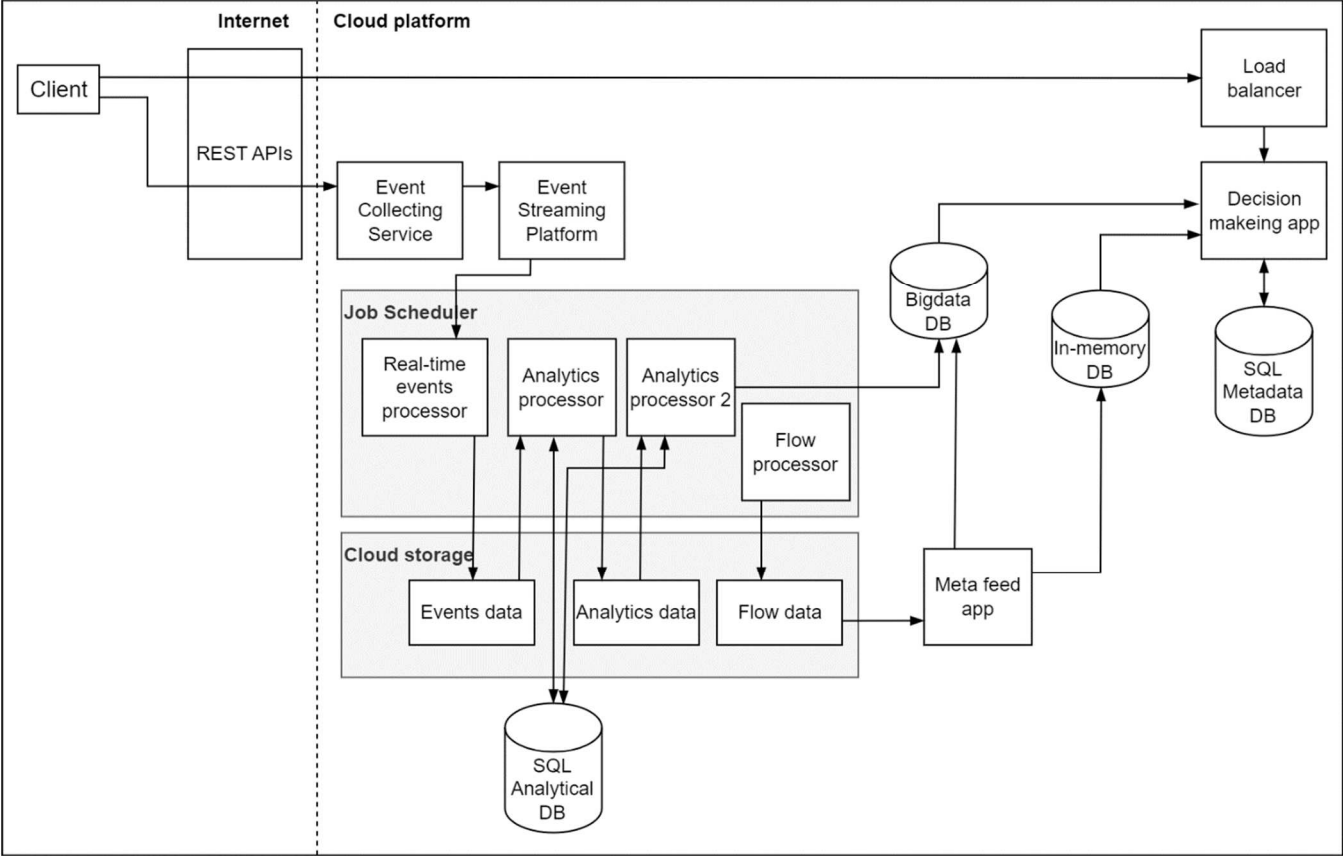


Figure Appendix B.1 Use case: shopping system – original architecture (large)

Appendix B.2. Shopping System – Architecture With Tiers (large)

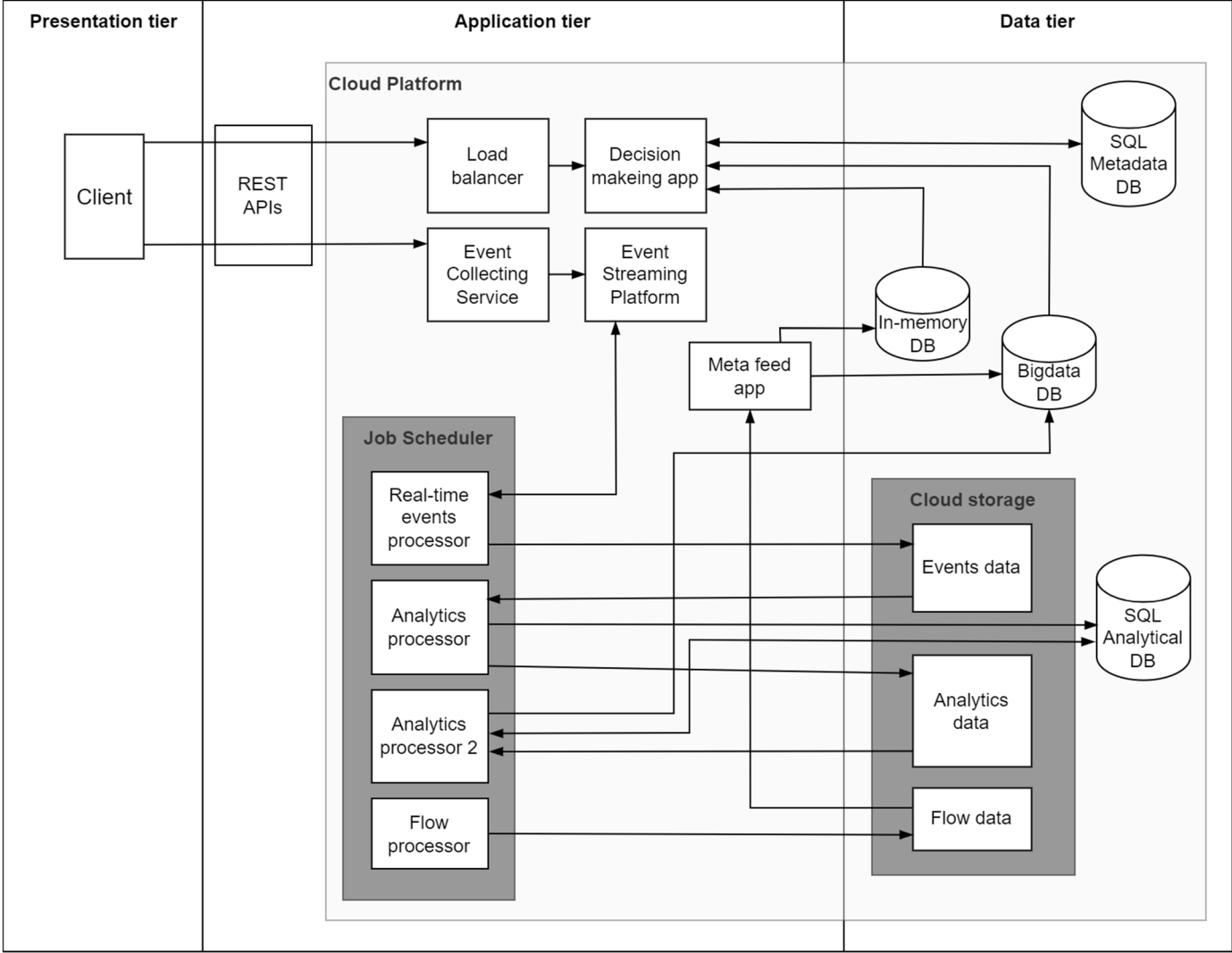


Figure Appendix B.2 Use case: shopping system – architecture with tiers (large)

### Appendix B.3. Shopping System – Level 1 Applications Diagram (large)

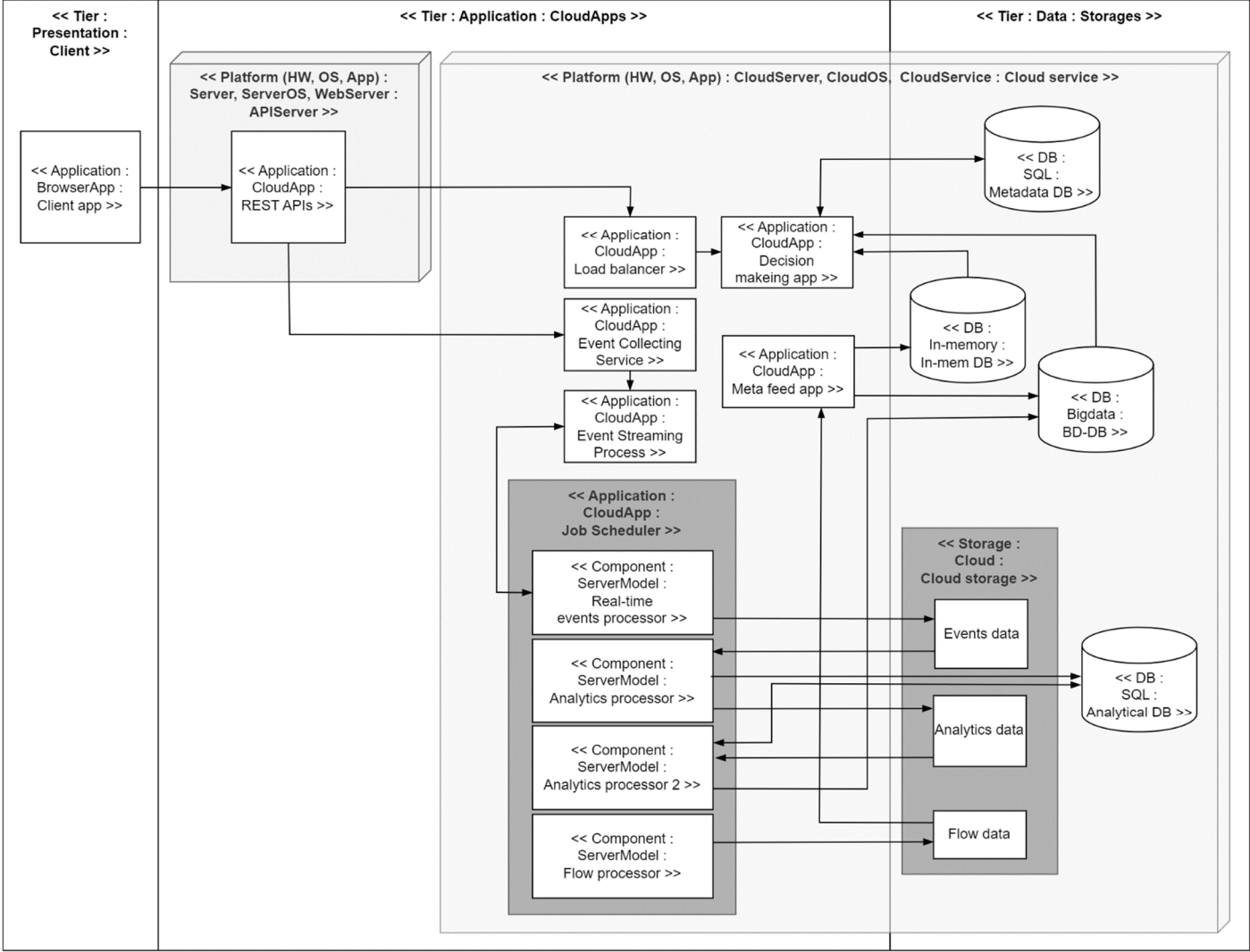


Figure Appendix B.3 Use case: shopping system – level 1 applications diagram (large)

**Appendix B.4. MiCADO-Edge [180] Architecture drawn using the RiWAsML**

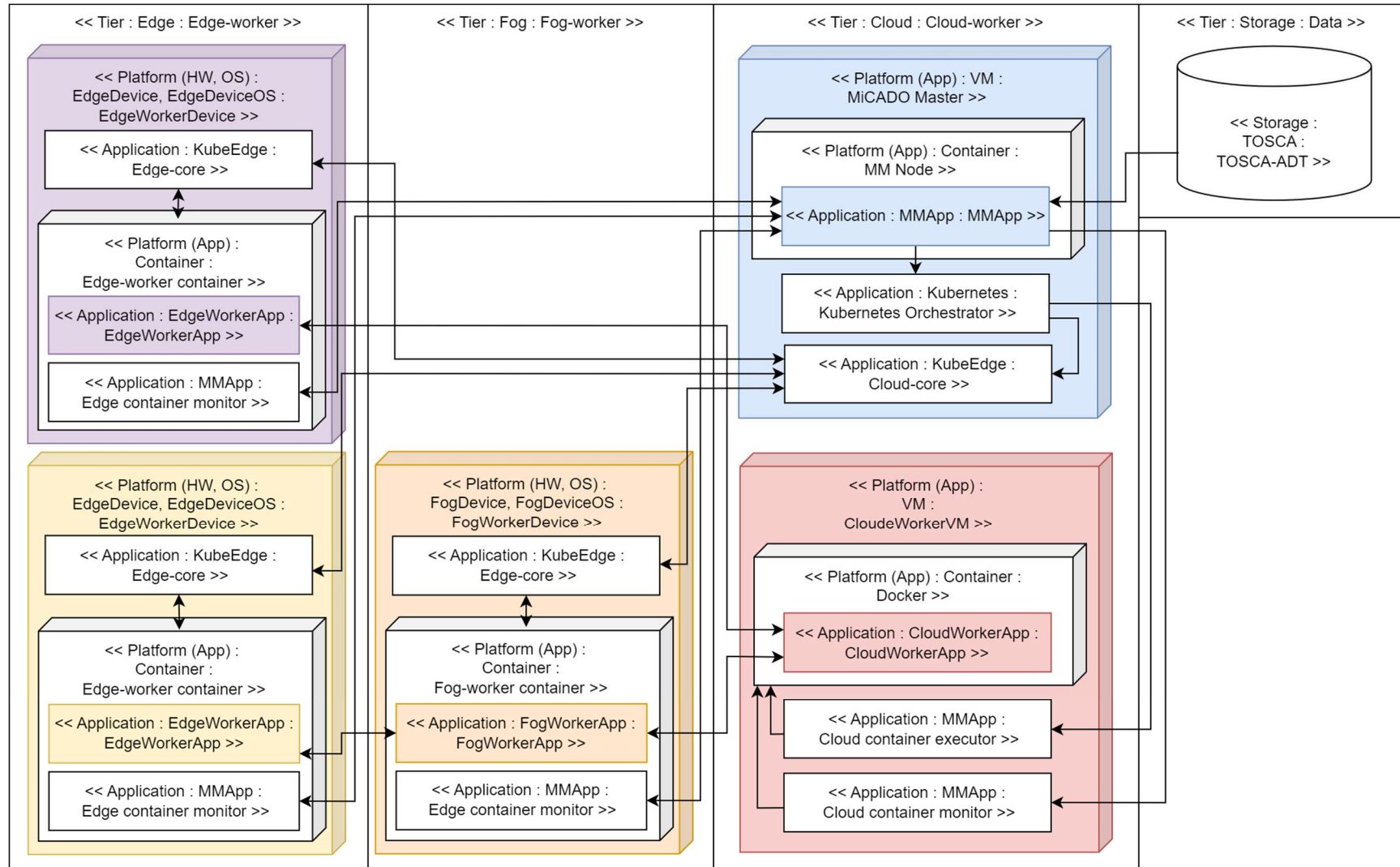


Figure Appendix B.4 Use case: MiCADO-Edge Architecture drawn using the RiWAsML (large)

## Appendix C. Use Case: Learning Management System (LMS)

### Appendix C.1. Use Case: LMS – Use Case Descriptions

This appendix only provides descriptions of the non-trivial use cases to understand the core functions of the Smartest system.

#### Browse and Search Use Case

<b>Use case name:</b> <u>Browse and Search</u>
<b>Actor:</b> Registered user
<b>Associated GUI:</b> Browse repo (refer to section 8.3.2.3)
<b>Description:</b> Registered users can browse and search for the available repositories using filters. They can view a selected repository from the list. Students can bookmark repositories from the list.
<b>Preconditions:</b> The user should be logged in.
<b>Post conditions:</b> None.

#### Manage Users Use Case

<b>Use case name:</b> <u>Browse and Search</u>
<b>Actor:</b> Registered user
<b>Associated GUI:</b> Browse repo (refer to section 8.3.2.3)
<b>Description:</b> Registered users can browse and search for the available repositories using filters. They can view a selected repository from the list. Students can bookmark repositories from the list.
<b>Preconditions:</b> The user should be logged in.
<b>Post conditions:</b> None.

### Manage Support Categories Use Case

<b>Use case name:</b> <u>Browse and Search</u>
<b>Actor:</b> Registered user
<b>Associated GUI:</b> Browse repo (refer to section 8.3.2.3)
<b>Description:</b> Registered users can browse and search for the available repositories using filters. They can view a selected repository from the list. Students can bookmark repositories from the list.
<b>Preconditions:</b> The user should be logged in.
<b>Post conditions:</b> None.

### Provide Support Use Case

<b>Use case name:</b> <u>Browse and Search</u>
<b>Actor:</b> Registered user
<b>Associated GUI:</b> Browse repo (refer to section 8.3.2.3)
<b>Description:</b> Registered users can browse and search for the available repositories using filters. They can view a selected repository from the list. Students can bookmark repositories from the list.
<b>Preconditions:</b> The user should be logged in.
<b>Post conditions:</b> None.

### Manage Entries Use Case

<b>Use case name:</b> <u>Browse and Search</u>
<b>Actor:</b> Registered user

<b>Associated GUI:</b> Browse repo (refer to section 8.3.2.3)
<b>Description:</b> Registered users can browse and search for the available repositories using filters. They can view a selected repository from the list. Students can bookmark repositories from the list.
<b>Preconditions:</b> The user should be logged in.
<b>Post conditions:</b> None.

### Check Progress Use Case

<b>Use case name:</b> <u>Browse and Search</u>
<b>Actor:</b> Registered user
<b>Associated GUI:</b> Browse repo (refer to section 8.3.2.3)
<b>Description:</b> Registered users can browse and search for the available repositories using filters. They can view a selected repository from the list. Students can bookmark repositories from the list.
<b>Preconditions:</b> The user should be logged in.
<b>Post conditions:</b> None.

**Appendix C.2. Use Case: LMS – Architecture (large)**

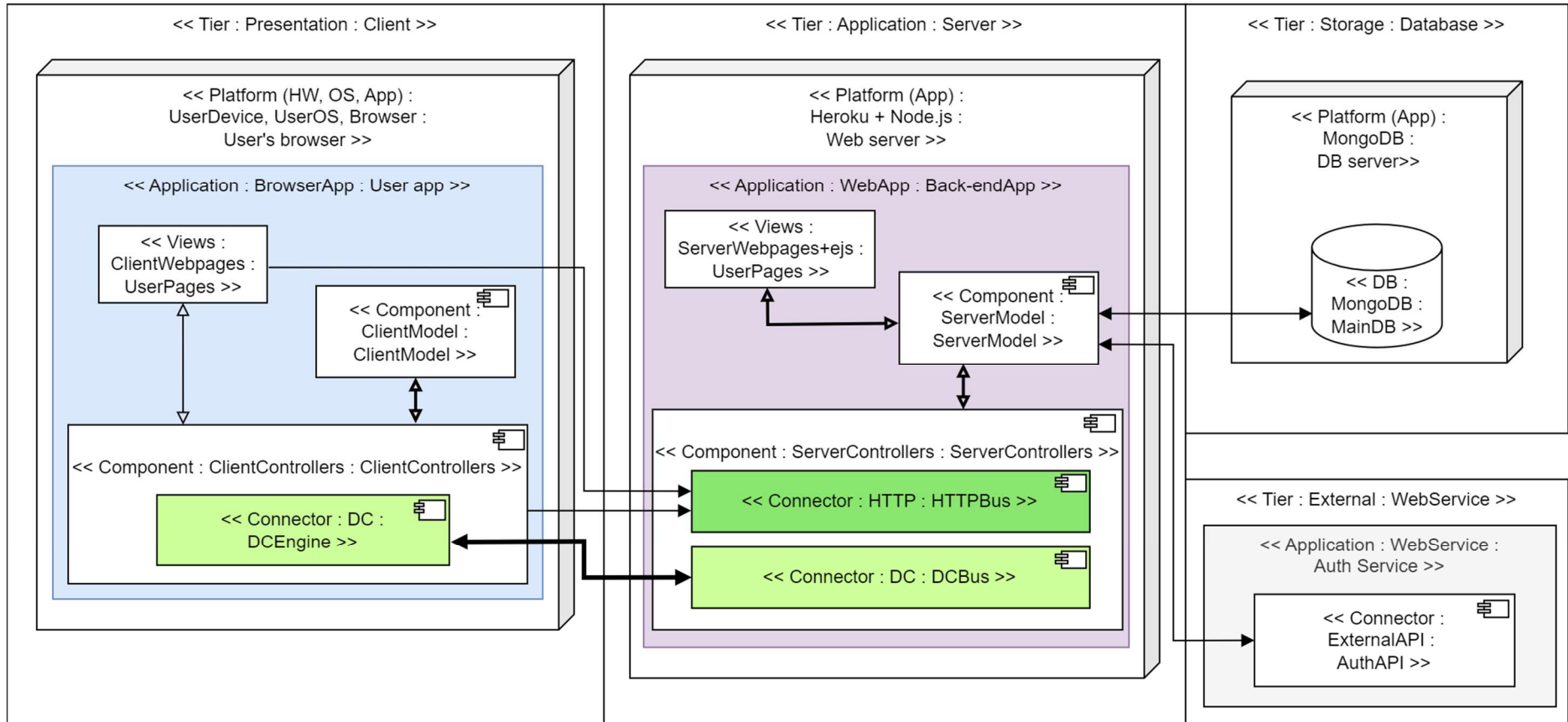


Figure Appendix C.2 Use case: LMS – architecture (large)



**Appendix C.3. Use Case: LMS with Web Service – Architecture (large)**

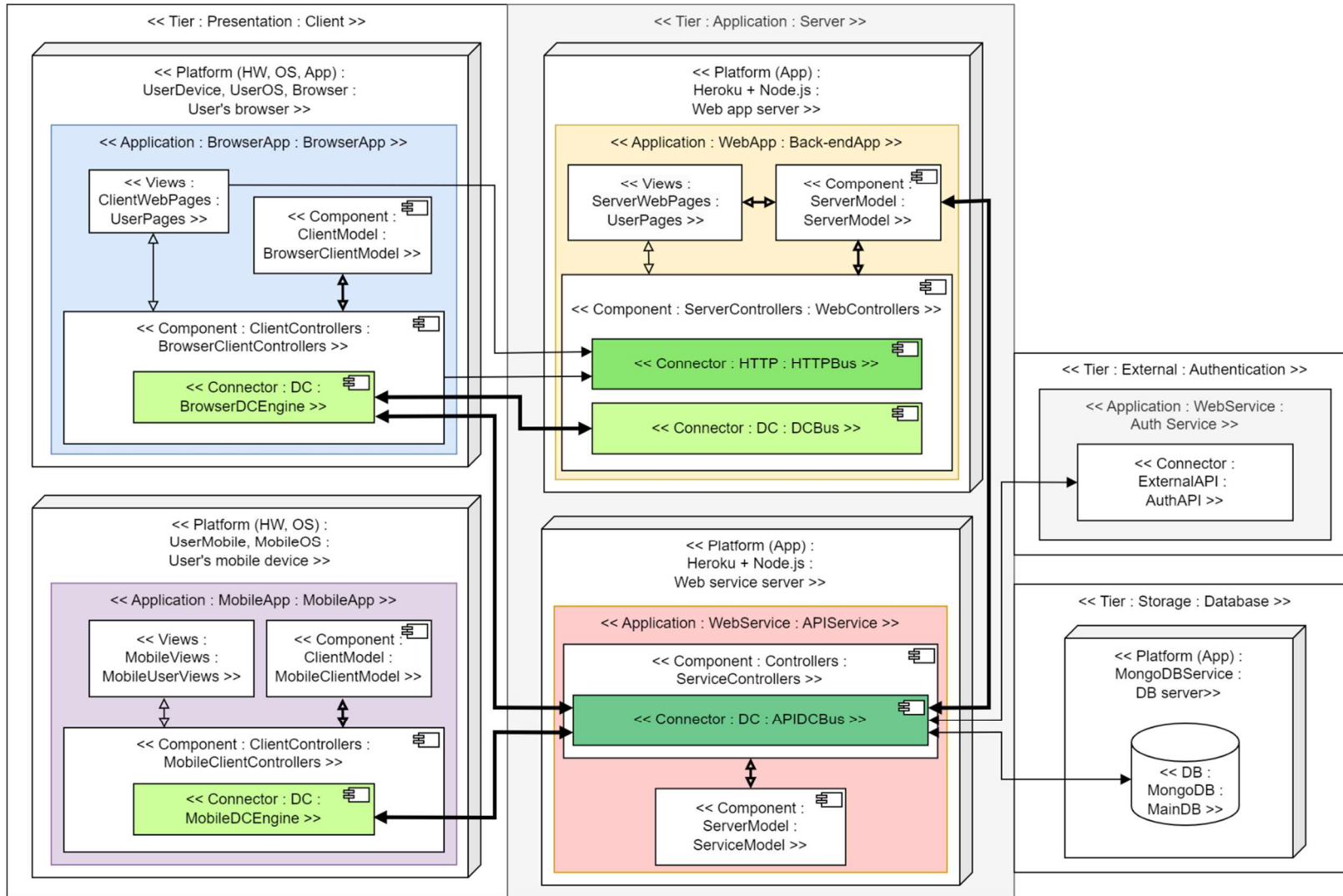


Figure Appendix C.3 Use case: LMS with web service – architecture (large)

### Appendix C.4. Use Case: LMS – View-Navigation Diagram (large)

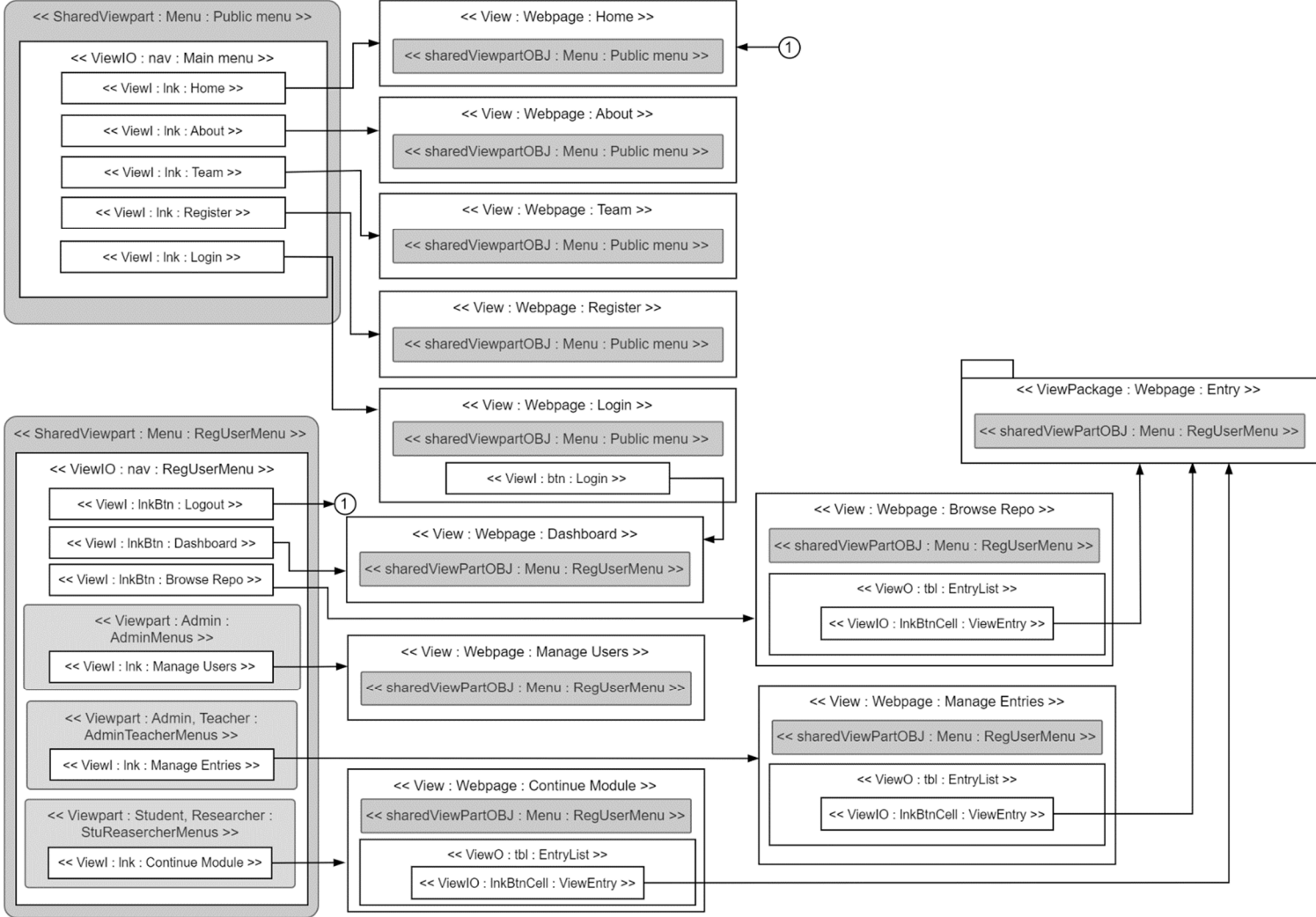


Figure Appendix C.4 Use case: LMS – view-navigation diagram (large)

## **Appendix D. Expert Evaluation**

### **Appendix D.1. Expert Evaluation Form**

---

**Rich Web-based Applications Modelling Language**

**Rich Web-based Applications Design Methodology**

# **RiWAsML and RiWAsDM**

**-An Evaluation-**

---

### **Section 1 – Evaluator Details**

1. **Name:**
2. **Email:**
3. **Bio:**

Please provide a short bio indicating your expertise in the domain of rich web-based application design and development.

4. **LinkedIn (or similar professional) profile:**
5. **Do you give consent to publish the details provided in this section? (yes/no)** (if there is any detail you don't want to publish, please mention):

## **Section 2 – Instructions**

1. Use MS Word's navigation pane to navigate between sections easily.
  - View -> Show -> Navigation Pane
2. To learn the scenario of the LMS
3. Refer to Appendix 1
4. You need some knowledge of the following. However, the evaluation tries to verify the **readability** and **understandability** of the new modelling language; therefore, I request that you attempt the evaluation first and refer to the links below to gain the required knowledge as needed.
  - **RiWAArch style** - Refer to Appendix 8
  - **UML** - <https://www.uml-diagrams.org/uml-25-diagrams.html>
  - **RiWAsML notations** - Refer to Appendix 9
5. Refer to the **General Notations** table in Appendix 9 to understand the syntax of Label and Communication channels.
6. If you need further assistance, please schedule an online meeting with me for a discussion.
7. Please justify your answers by providing a statement in the given space.

## **Section 3 – Quality Attributes**

**Consider the following quality attributes to justify your evaluation.**

**Simplicity** – In software engineering, simplicity refers to the separation of concerns, which appreciates decomposing a system and identifying and separating the modules for greater realisation, thus, management. A design language must separate and identify enough models and model-elements to provide enough realisation of the target systems.

**Comprehensiveness** – A RiWAs design methodology is considered comprehensive when it provides the following: models and model-elements to design all the aspects realised by the RiWAArch style, rules and guidelines for designing RiWAs and mapping the designs into development, and design and engineering approaches and guidelines to adopt into agile engineering environments.

**Learnability** – A design language is based on UML and the RiWAArch style; hence, it is easy to learn.

**Readability/understandability** – The models and model-elements contain enough details, making them easy to read and understand.

**Development support** – how easily the designs can be mapped into the actual development?

**Integrability** – how to integrate the RiWAsML-based design activities into RiWAs engineering by enabling agile model-driven development?

## **Section 4 – High-level Model and Elements Evaluation**

1. See the **architecture diagram** in Appendix 2. This architecture attempts to capture the high-level Application elements within the system, the platforms they run, and the tiers they belong to. Within applications, the views, controllers, models, DC-engine, and DC-bus and their communication are identified.
2. Explain your experience and opinions by rating using the scale below and justifying.

<b>Symbol</b>	<b>Interpretation</b>
5	Very high effect
4	High effect
3	Moderate effect
2	Less effect
1	Very low or no effect

3. Rate the usefulness of the architecture and its model-elements in the table below. Please justify your answer.

<b>Diagram and its elements</b>	<b>Usefulness</b>	<b>Justification</b>
<u>Architecture diagram</u>		
Tier element		
Platform element		
Application element		
Views element		
Components element		
Connectors element		
Other elements (database, external services)		

## **Section 5 – Low-level Models and Elements Evaluation**

### **5.1 View-Navigation Diagram**

- The **view-navigation diagram** is used to capture the following.
  - All the views in the system.
  - Related features to implement on common views.
  - Different paths for different user types to navigate to views.
- Consider the **view-navigation diagram** of the browser app in [Appendix 3](#).
- Explain your experience and opinions by rating using the same scale given in Section 4 and justifying.

<b>Diagram and its elements</b>	<b>Usefulness</b>	<b>Justification</b>
<u>View-navigation diagram</u>		
View element		
View package element		
Viewpart element		
SharedViewpart		
SharedViewpartOBJ		
ViewI/ViewO/ViewIO		

### **5.2 View Diagram and Controller Diagram**

- The **view diagram** is used to design the internal elements of the views, which help understand the implementation of the user functionalities.
- A view has a dedicated **controller**, which implements the event handlers.
- Consider the **view-controller diagram** of the **Browse Repo function** in [Appendix 5](#).
- Explain your experience and opinions by rating using the same scale given in Section 4 and justifying.

<b>Diagram and its elements</b>	<b>Usefulness</b>	<b>Justification</b>
<u>View diagram</u>		
-View element		
-View package element		
-Viewpart element		
-ViewI/ViewO/ViewIO		
<u>Controller diagram</u>		
-ControllerClass element		
<u>View-Controller diagram</u> (as a single diagram)		

### 5.3 DC-bus Diagram and EndpointsCollection Diagram

- The **EndpointsCollection** class helps understand the API design using the OOP practices.
- The **DC-bus diagram** is used to understand the APIs by structuring them into classes/groups.
- Consider the **DC-bus diagram** of the **DCBus** element of the **Back-endApp** application and its **RoutesRegUsers EndpointsCollection** in [Appendix 6](#).
- **Note:** only admin APIs are included in the endpoints diagram for demonstration.
- Explain your experience and opinions by rating using the same scale given in Section 4 and justifying.

Diagram and its elements	Usefulness	Justification
<u>DC-bus diagram and EndpointsCollection diagram</u>		
-Connector element		
-Endpoints class element		

### 5.4 AppModel Diagram and ModelClass Diagram

- An **AppModel diagram** is utilised to design a domain model of a particular Application element's model using a class diagram.
- A **ModelClass diagram** can be used to design a given class of an AppModel diagram.
- Consider the diagrams in [Appendix 7](#).
- Explain your experience and opinions by rating using the same scale given in Section 4 and justifying.

Diagram and its elements	Usefulness	Justification
<u>AppModel diagram and ModelClass diagram</u>		
-Component element		
-Class element		

### 5.5 View-Process Sequence Diagram

- The **view-process sequence diagram** is similar to the UML sequence diagram and is used to design the execution flow of a function.
- A **view-process sequence diagram** can capture the Application elements and their processing elements, which take part in the execution.
- Consider the **view-process sequence diagram** for the **login function** in [Appendix 4](#).
- Explain your experience and opinions by rating using the same scale given in Section 4 and justifying.

Diagram and its elements	Usefulness	Justification
<u>View-process sequence diagram</u>		
-Application element		
-View element		
-Controller element		
-Endpoints element		
-ModelClass object element		

### 5.6 Element Names and Communication Channels

- Consider the **label** used for the design elements and **communication channels** syntax.
- Refer to the **General Notations** table in [Appendix 9](#) to understand the syntax of Label and Communication channels.
- Explain your experience and opinions by rating using the same scale given in Section 4 and justifying.

Diagram and its elements	Usefulness	Justification
<u>Element names (label)</u>		
<u>Communication channels</u>		
-Standard communication		
-Delta-Communication		
-View-controller communication		
-Method call and return		
-ModelClass object element		



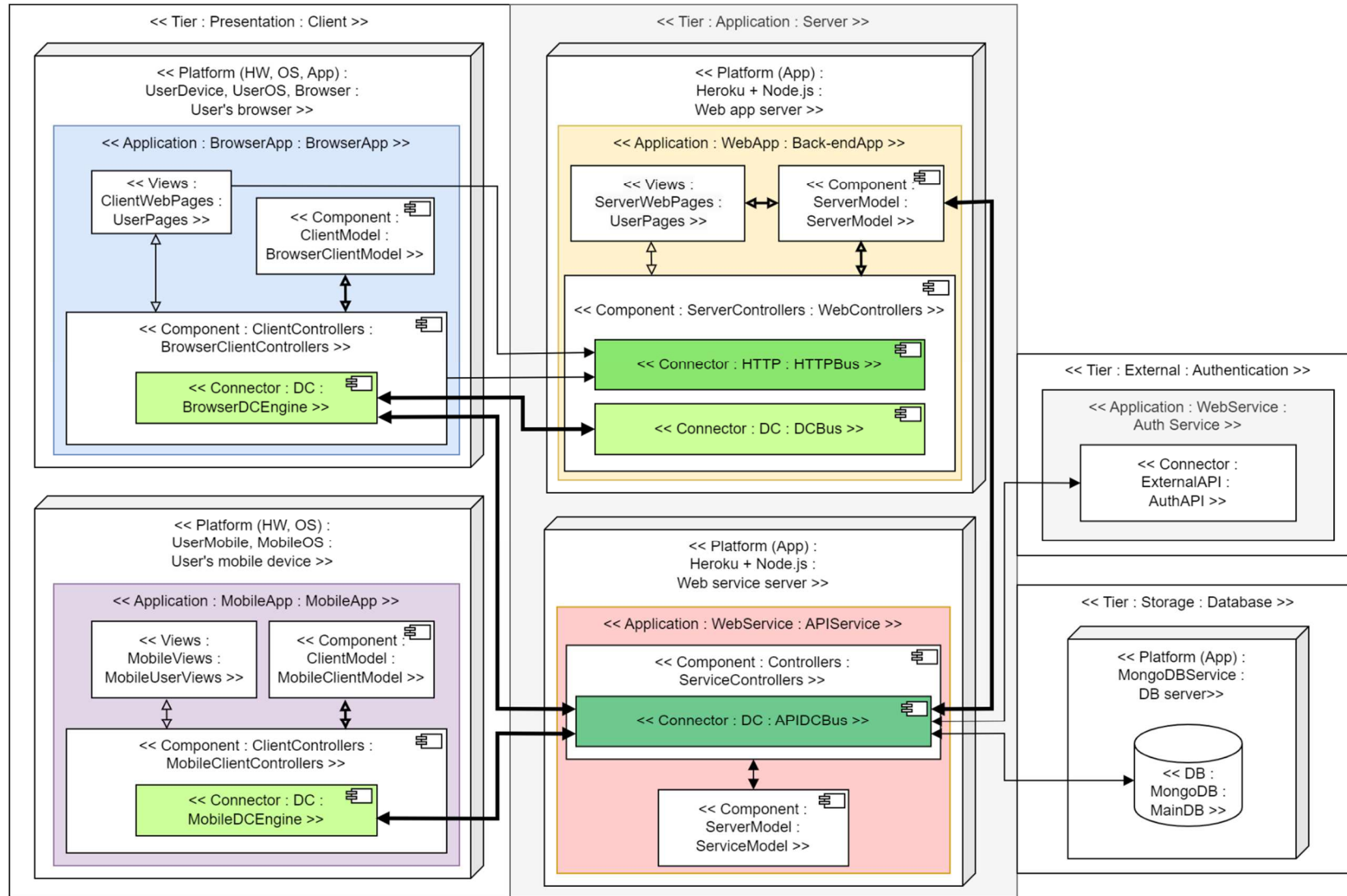


**Appendix 1 – Learning Management System Use Case Diagram**

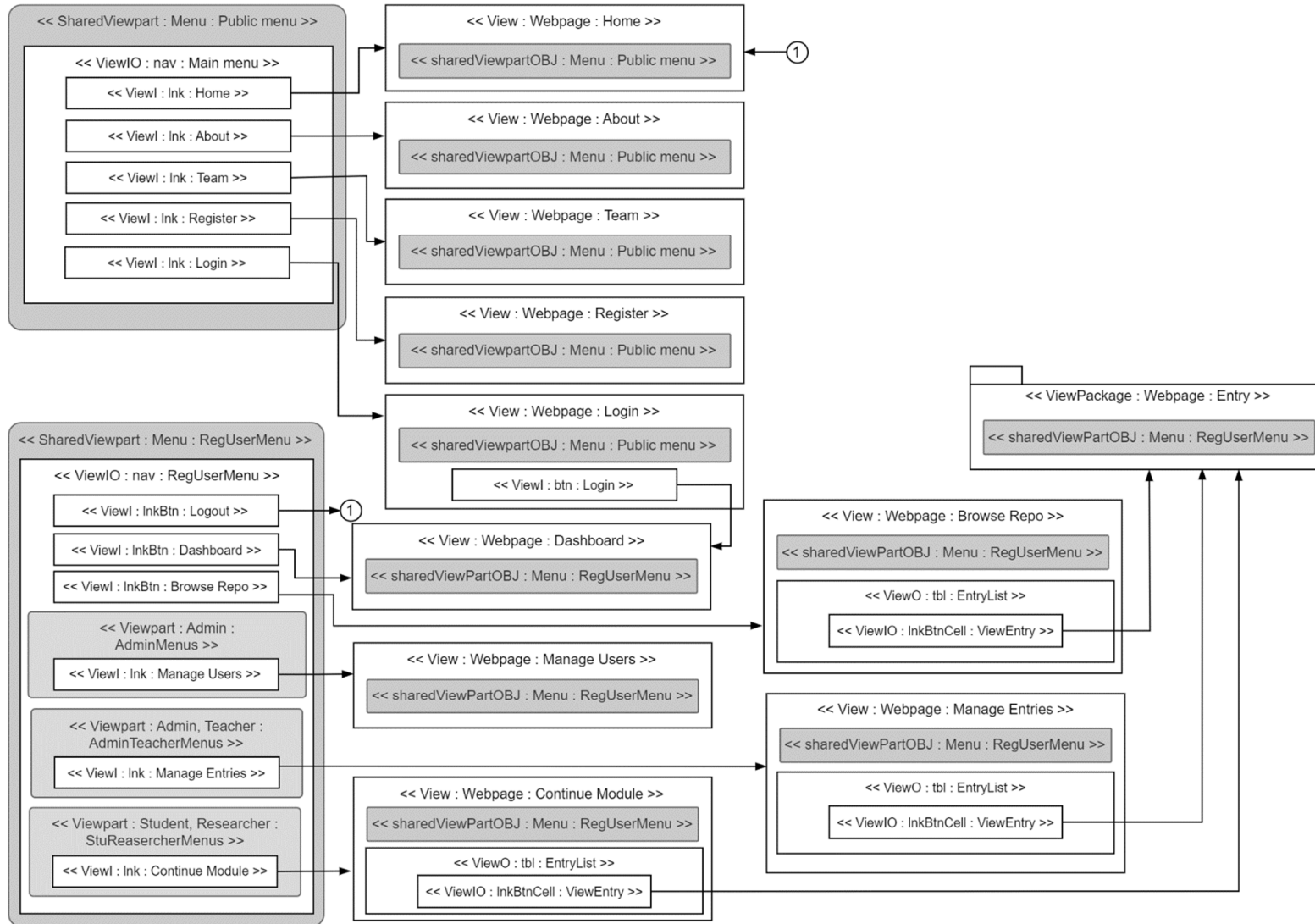


**NOTE:** The registered users have to log in to the system to use any feature.

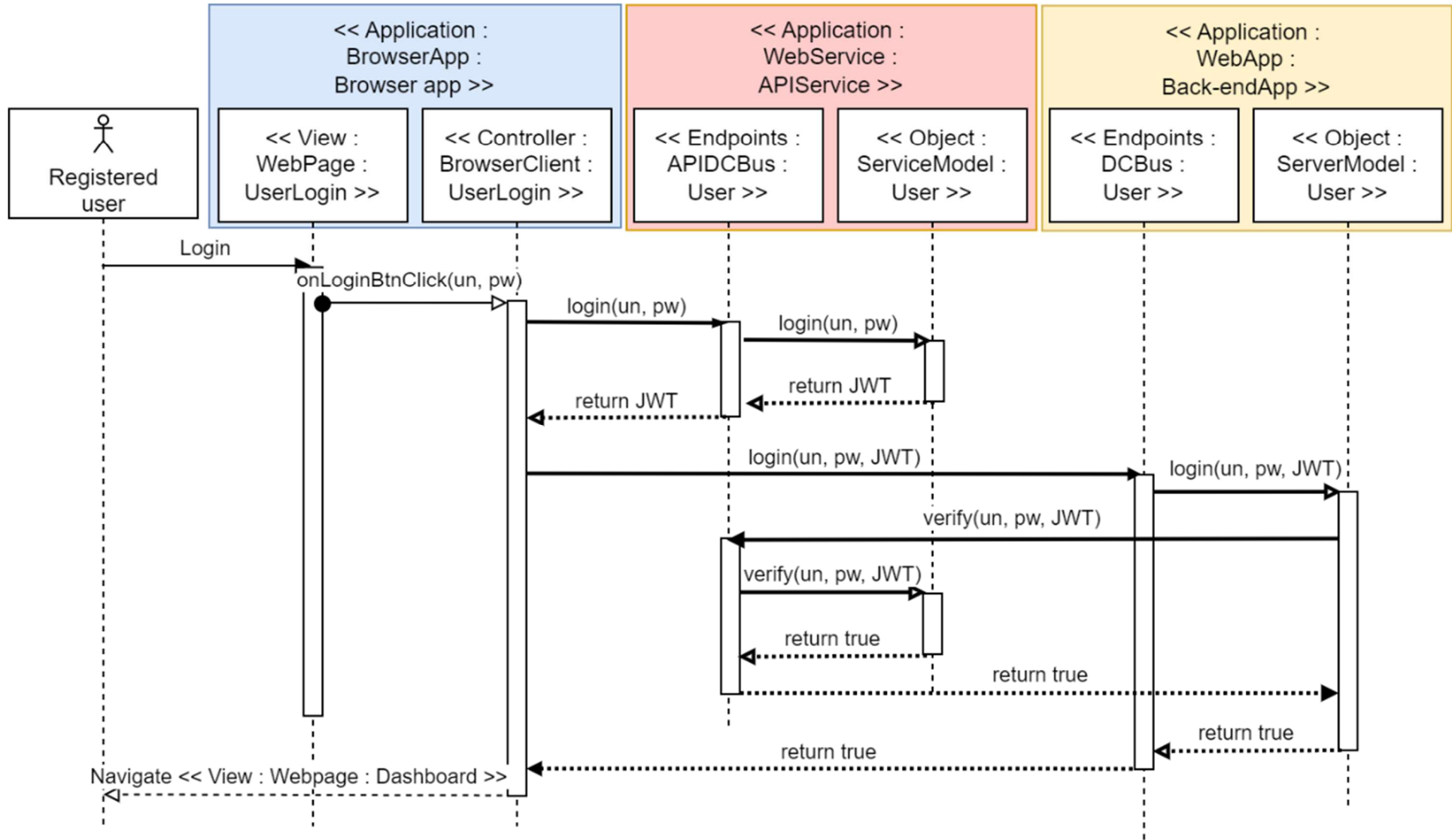
### Appendix 2 – LMS Architecture Diagram



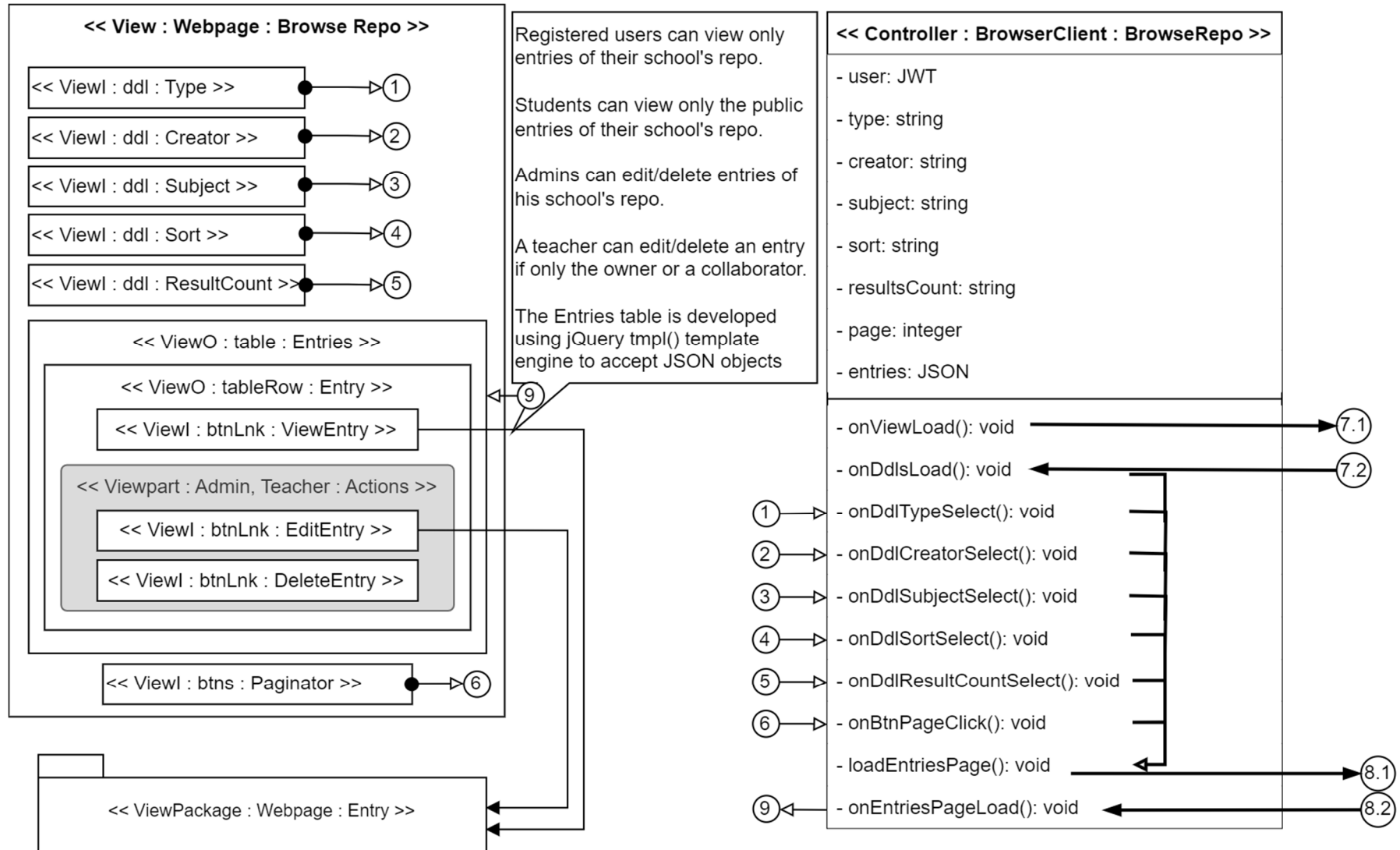
### Appendix 3 – View-Navigation Diagram



**Appendix 4 – View-Process Sequence Diagram**

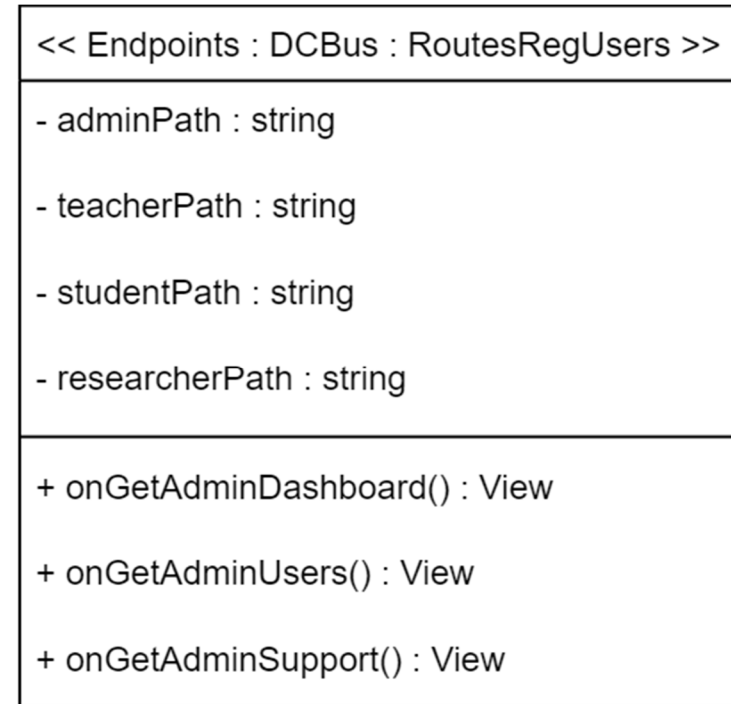
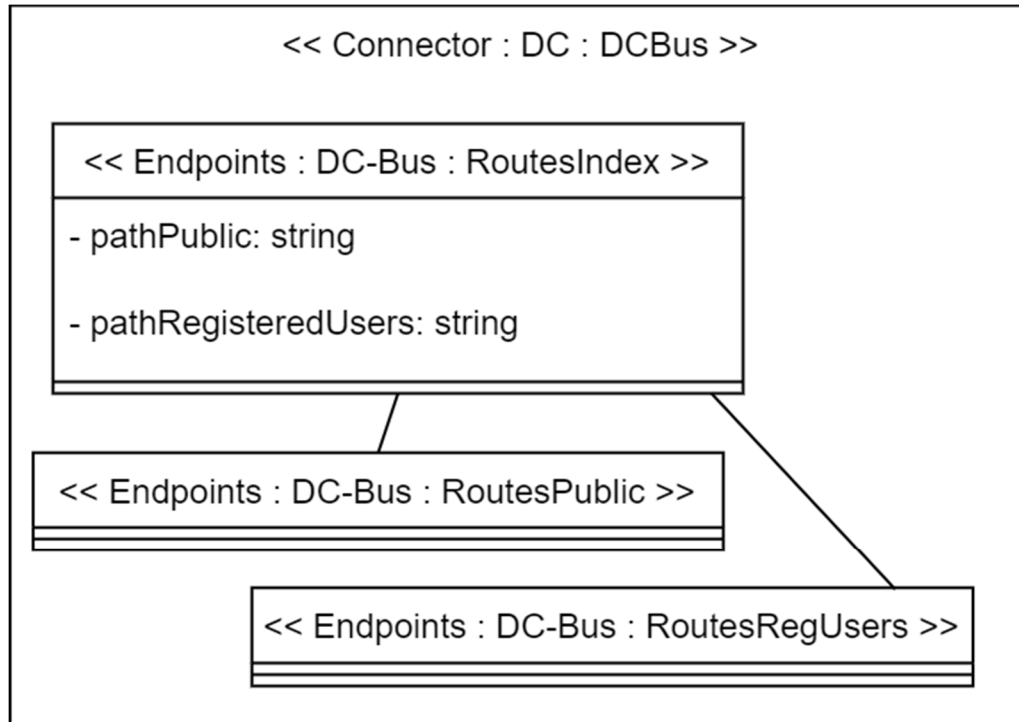


**Appendix 5 – View-Controller Diagram of Browse Repo**

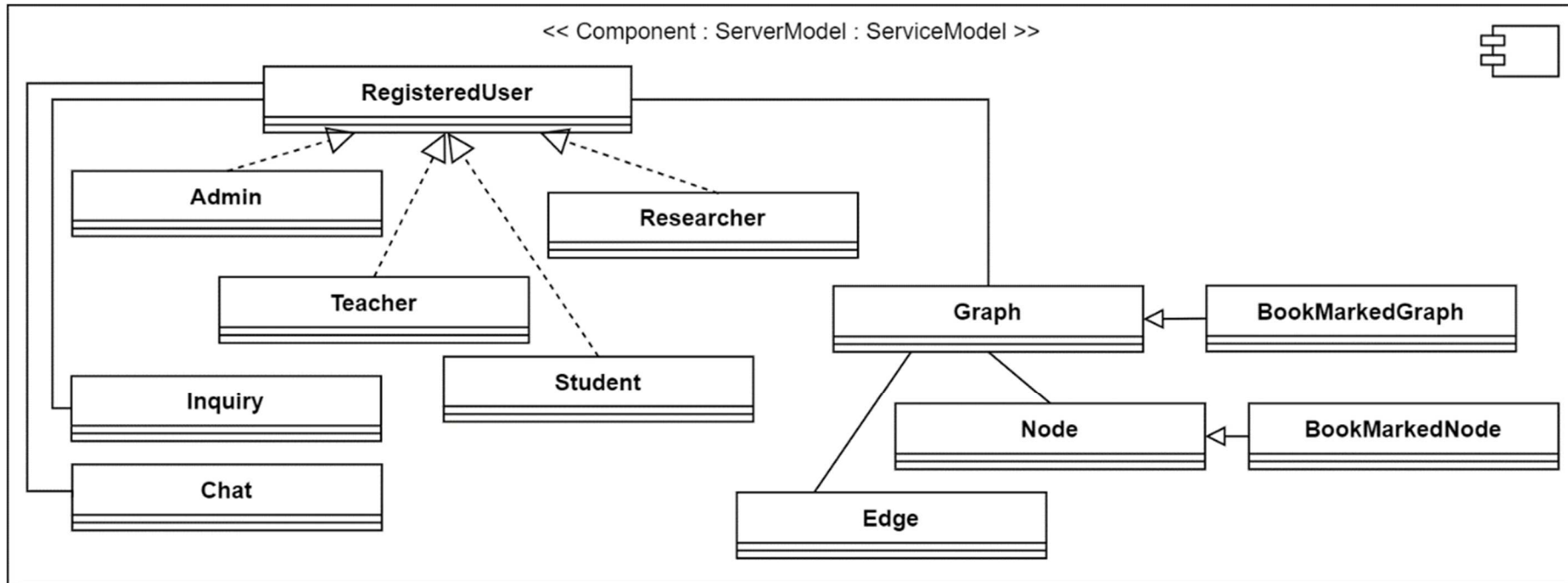


**Appendix 6 – DC-bus Diagram and EndpointsCollection Diagram**

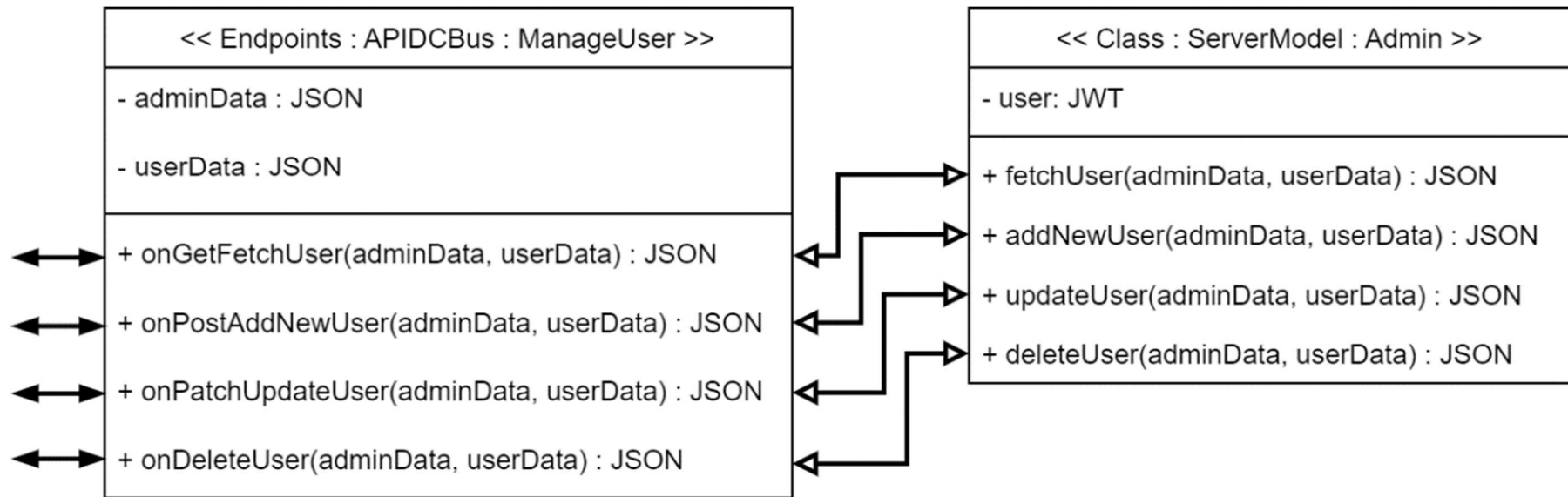
- The DC-bus diagram provides the internal design of the *Back-endApp* element of the architecture.
- The EndpointsCollection diagram contains the **partial design** of the *RoutesRegUsers* class of the DC-bus diagram.



Appendix 7 – AppModel Diagram and ModelClass Diagram



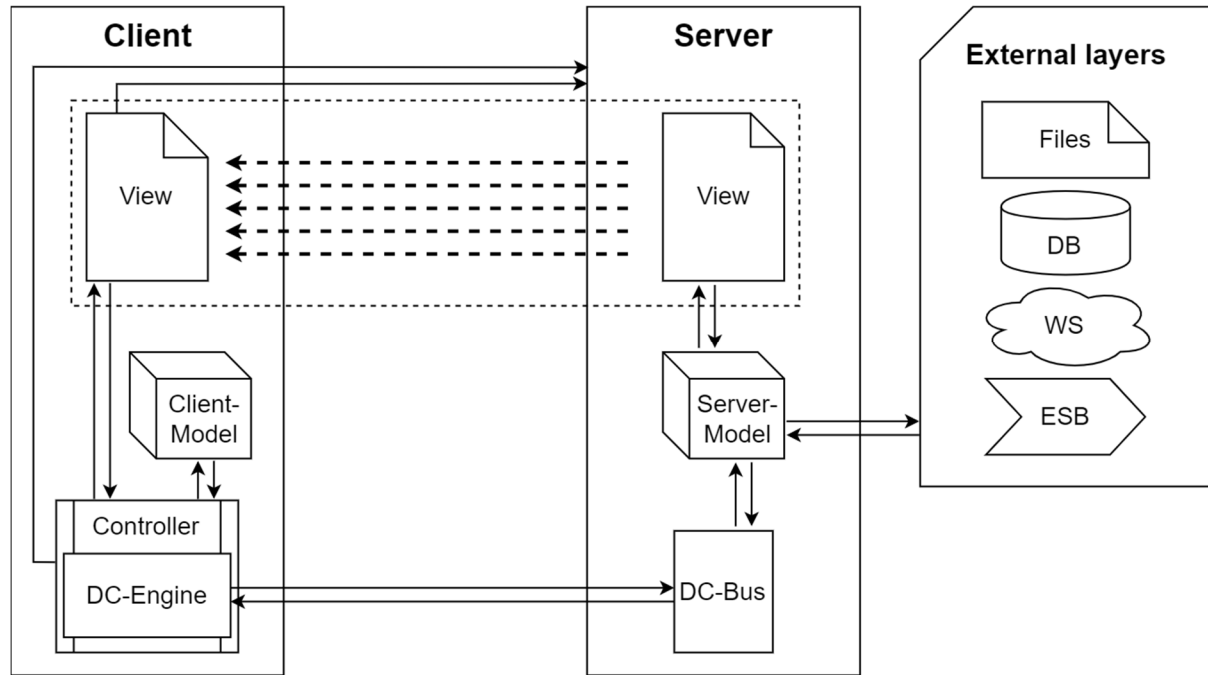




**Appendix 8 – The RiWAArch Style**

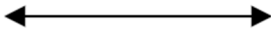
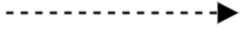




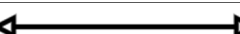
**Read** **the** **paper:**

[https://www.researchgate.net/publication/346514514\\_RiWAArch\\_Style\\_An\\_Architectural\\_Style\\_for\\_Rich\\_Web-Based\\_Applications](https://www.researchgate.net/publication/346514514_RiWAArch_Style_An_Architectural_Style_for_Rich_Web-Based_Applications)



**Appendix 9 – RiWAsML Reference**

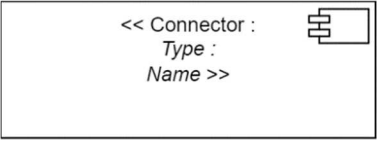
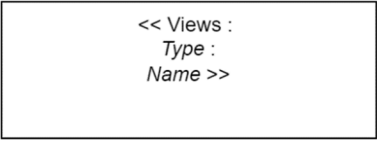
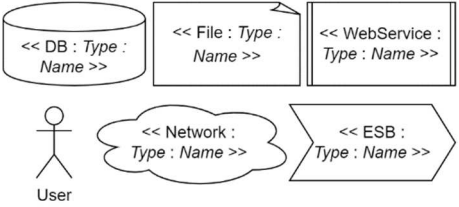
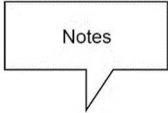
**General Notations**

Element	Type	Notation
Label	-NA-	<< Element : Type : Name >>
Communication channels	Standard communication (HTTP and other standard protocols)	 Regular
	Delta-Communication	 Return (sequence diagram)
	Delta-Communication – push (reserved)	 Regular
	View-Controller	 Return (sequence diagram)
	Method call and return	 Regular
		 Return (sequence diagram)
		 Return (sequence diagram)

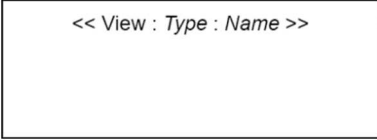
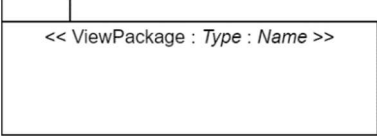
	With numbered connectors	
--	--------------------------	--

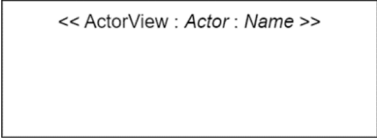
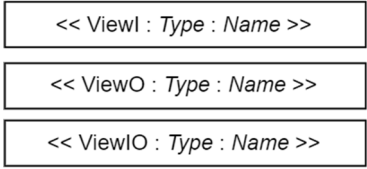
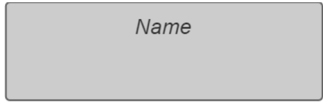
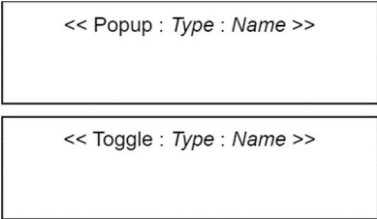

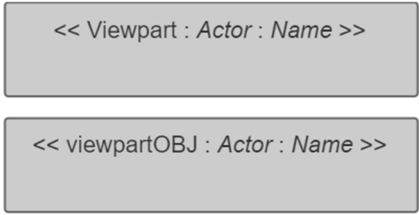
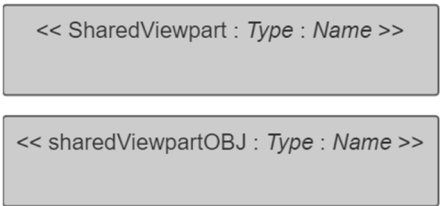
### High-level Diagrams and their Notations

Element	Type	Notation
<b>Level 1 Applications Model</b>		
<b>Tier</b> – horizontal grouping of elements based on their roles	Presentation Application Storage	
<b>Platform</b> – provides the environment for an Application element to run.	<b>Device:</b> Hardware + OS Application-level environment type	
<b>Application</b> – An element executable within its platform, communicating with other Application elements in the system to perform functions.	WebApp MobileApp DektopApp	
<b>Level 2 View Process Model</b>		
<b>Component</b> – a logic processing element within an Application element.	Controllors ClientControllors ServerControllors ClientModel ServerModel	

<p><b>Connector</b> – a communication processing element within an Application element, enabling the Application element to communicate.</p>	<p>DCEngine DCBus</p>	
<p><b>Views</b> – element containing a collection of GUIs of a client-side Application element.</p>	<p>WebPages Activities Windows</p>	
<p><b>Other high-level elements</b> – DB, File, WebService, User/Actor Network, ESB</p>	<p>-NA-</p>	
<p><b>Notes</b> – element to add related text-based details.</p>	<p>-NA-</p>	

**Low-level Diagrams and their Notations**

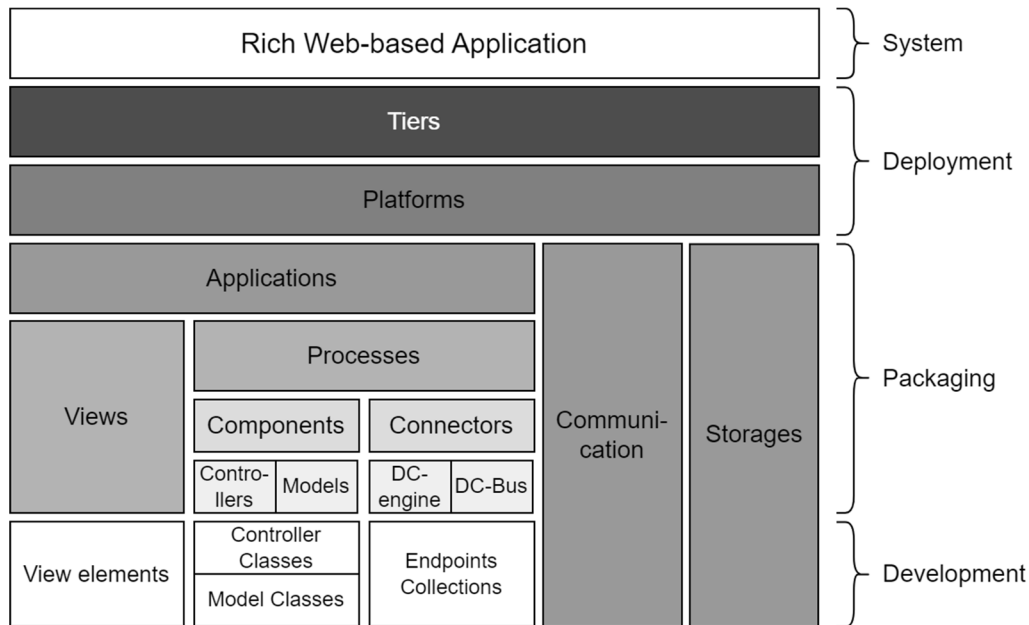
Element	Type	Notation
<b>View Model, View-Navigation Model</b>		
<p><b>View</b> – a single GUI implementation element</p>	<p>WebPage Activity Window</p>	
<p><b>ViewPackage</b> - a single GUI implementation element that wraps supporting elements.</p>	<p>WebPage Activity Window</p>	

<p><b>ActorView</b> – A view implementing element containing elements of a particular actor.</p>	<p>Actor: The actor of the Viewpart</p>	
<p><b>View I/O elements</b> – Input/output elements of views.</p>	<p>Refer to Table 6.1</p>	
<p><b>Container/Section</b> – a virtual section of a view.</p>	<p>-NA-</p>	
<p><b>Popup/Toggle</b> – a widget in a view which can be shown/hidden or enabled/disabled.</p>	<p>Blocking NonBlocking Show Hidden Enabled Disabled</p>	
<p><b>Trigger communication channel</b> – connect the GUI element, which triggers an event to the event handler of the controller.</p>	<p>-NA-</p>	
<p><b>Viewpart</b> – a GUI section for a particular actor.  <b>ViewpartOBJ</b> – an object of a Viewpart.</p>	<p>Actor: The actor of the Viewpart</p>	
<p><b>SharedViewpart</b> – a Viewpart for a particular actor, shareable with multiple views.</p>	<p>Refer to table 6.1</p>	

<p><b>SharedViewpartOBJ</b> – a SharedViewpart object</p>					
<p><b>AppControllers Model, Controller Model, View-Controller Model</b></p>					
<p><b>Controller</b> – A class which implements event handling for a particular view.</p>	<p>Client</p> <p><b>NOTE:</b> The name should be the same as the related view's name</p>	<table border="1" style="width: 100%;"> <tr> <td style="padding: 2px;">&lt;&lt; Controller : Type : ViewName &gt;&gt;</td> </tr> <tr> <td style="padding: 2px;">- propName: type</td> </tr> <tr> <td style="padding: 2px;">+ methodName(params): type</td> </tr> </table>	<< Controller : Type : ViewName >>	- propName: type	+ methodName(params): type
<< Controller : Type : ViewName >>					
- propName: type					
+ methodName(params): type					
<p><b>Invoke</b> – indicate a controller's method invoking a popup/toggle.</p>	<p>Show/hide/enable/disable</p>	<p style="text-align: center;">&lt;&lt; Type &gt;&gt; →</p>			
<p><b>AppModel Model</b></p>					
<p><b>ModelClass</b> – A class which implements domain logic.</p>	<p>ClientModel</p> <p>ServerModel</p>	<table border="1" style="width: 100%;"> <tr> <td style="padding: 2px;">&lt;&lt; Class : Type : ClassName &gt;&gt;</td> </tr> <tr> <td style="padding: 2px;">- propName : type</td> </tr> <tr> <td style="padding: 2px;">+ methodName(params): type</td> </tr> </table>	<< Class : Type : ClassName >>	- propName : type	+ methodName(params): type
<< Class : Type : ClassName >>					
- propName : type					
+ methodName(params): type					
<p><b>DC-bus Model and EndpointsCollection Model</b></p>					
<p><b>EndpointsCollection</b> – A class which implements communication APIs of a server application.</p>	<p>DCBus</p> <p>HTTPBus</p>	<table border="1" style="width: 100%;"> <tr> <td style="padding: 2px;">&lt;&lt; Endpoints : Type : Name &gt;&gt;</td> </tr> <tr> <td style="padding: 2px;">- propName : type</td> </tr> <tr> <td style="padding: 2px;">+ endpointName(params) : type</td> </tr> </table>	<< Endpoints : Type : Name >>	- propName : type	+ endpointName(params) : type
<< Endpoints : Type : Name >>					
- propName : type					
+ endpointName(params) : type					

## Appendix 10 – RiWAsDM process

### The RiWAsML Architecture



### RiWAs Design Approach with RiWAsDM

RiWAsDM appreciates using the top-down design approach since the RiWAs share some high-level common characteristics and essential features, which can be realised with the RiWAArch style. Besides, it's not inappropriate to use the bottom-up approach when the low-level details of the features of the RiWA are precisely identified, and the engineering team members have some experience in RiWA engineering.

### RiWAs Engineering Approach with RiWAsDM

RiWAsDM recommends using the *Agile Model-Driven Development (AMDD)* approach for RiWAs engineering to benefit from both model-driven engineering and agile engineering.

### Rules and Guidelines for Designing RiWAs with RiWAsML

This section first states the rules and guidelines to follow when using the introduced RiWAsML. Most of the rules and guidelines are already discussed in related sections while introducing the RiWAsML models and their model-elements. Those rules and guidelines are listed in this section, with further elaborations as required for clarity and reference. The rules are given as “Rule:” to be followed when using the RiWAsML, and the guidelines are mentioned as “Guideline:” to consider



as best practice. This section aligns with step 3.2 of the RiWAsDM implementing process (refer to Figure 1.2 in section 1.5.3).

### **Rules and Guidelines for General Elements of RiWAsML**

This section sets the rules and guidelines for the RiWAsML label and communication channels. Further, some overall guidelines are given in the direction of improving the simplicity and the readability of the design.

#### Label

**Rule:** The RiWAsML model-elements should use the RiWAsML label format.

**Rule:** The RiWAsML model-elements should use the specified values for the element segment of the label.

**Rule:** The RiWAsML model-elements should use the specified values for the type segment of the label when the element type is exact.

**Guideline:** When the element is of a type the RiWAsML does not specify, a suitable custom type could be assigned.

**Guideline:** RiWAsML suggests using the following cases for the values of the element, type, and name segments of the RiWAsML model-elements' to align with OODD practices.

- Generally, use the Pascal case for naming.
- For naming objects such as *ViewpartOBJ* and *SharedViewpartOBJ*, method names, and attributes, use the Camel case.

#### Communication Channels

**Guideline:** RiWAsML realises the communication between the architectural elements based on the request-response model. Thus, the bidirectional arrows used to depict the communication channels have a specific meaning, and the requesting and requested elements are known as follows.

- **Between view and controller:** The view requests the controller by triggering events and passing the data to the controller. The controller responds by doing the needful and updating the view with information.
- **Between controller and client-model:** A controller calls a client-model's method, and the method processes the request and returns the result.
- **Between controller and DC-bus:** A controller sends a DC request to the DC-bus via the DC-engine. The DC-bus does the needful, and the results are sent using the DC response to the controller via the DC-engine.
- **Between DC-bus and server-model:** The DC-bus calls a server-model's method, and the method processes the request and returns the result.

**Guideline:** It is possible to use unidirectional arrows when required, for example, when views and controllers of a browser-based client send HTTP requests to the server.

**Guideline:** A bi-directional arrow can still be used when required to denote communication between the methods of the same class or multiple classes. In controllers, event handlers always call the other methods; hence, the calling and the called methods are straightforward. For communication between methods other than the event handlers, use a black dot on the calling method to depict the caller, similar to the event handling view elements.

#### General Guidelines

**Guideline:** RiWAsML suggest using colours for the model-elements to improve simplicity and readability,

#### Rules and Guidelines for High-level Designing with RiWAsML

This section specifies the rules and guidelines for the *level 1 Applications diagram*, *level 2 view-process diagram*, *level 1+2 architecture diagram*, and their elements.

**Guideline:** The high-level aspects can be hierarchically designed using the separated *level 1 Applications diagram* and *level 2 view-process diagram* or can be included in a single design using the *level 1+2 architecture diagram* depending on the size and the complexity of the RiWA. Priority should be given to the readability of the design.

**Guideline:** In RiWAs with a browser-based client, the case of the server sending views to the client as the response to the HTTP requests is not depicted on the diagrams, considering it is a known fact.

**Guideline:** The tiers are not required to have the same height and width (refer to Figure 8.9).

**Guideline:** The Platform elements can expand across the tiers as required (refer to Figure 8.3).

**Guideline:** On high-level diagrams, the views and AppControllers elements always use plural form for the type and name segments of the label. In the case of views, the element segment is also in plural.

#### Rules and Guidelines for Low-level Designing with RiWAsML

This section specifies the rules and guidelines for the low-level design diagrams and their elements. Additionally, some development-supportive rules or guidelines are given as “Rule [development]:” Or “Guideline [development]:”.

#### View Diagram

**Guideline:** A view diagram may not contain all the GUI elements and may show only the GUI elements required to implement functionalities.

**Guideline:** A view diagram does not provide the actual layout.

**Guideline:** A view can be designed with its dedicated controller as a pair on the same diagram.

**Guideline [development]:** Use a GUI element's type and name label segment values to derive its development name. For example, consider the GUI element with the label `<< ViewI : btn : Delete >>`; in that case, use `btnDelete` as the development name of it.

**Guideline:** Use *Viewparts* for the readability of the design; they may or may not be used in actual development.

**Guideline [development]:** For a *ViewPackage* with *ActorViews*, the view's actual development name is the name of the *ViewPackage*. Use the names of the *ActorViews* as the display names of the view for the actors of each *ActorView*.

**Rule [development]:** For *Popups*, use the type segment to specify the behaviour of the *Popup* as "Blocking" or "NonBlocking". For *Toggles*, use the type segment to specify the initial state as Show/Hidden/Enabled/Disabled.

**Guideline [development]:** Do not mix the JavaScript code with the view's HTML code for browser-based client apps. If the controller has less code and decides to write it on the HTML file, always write the JS code on the head's script section without mixing it with HTML code anywhere else on the document.

**Guideline:** Each view should be available on the *view-navigation diagram*. There cannot be any views not included in the *view-navigation diagram*.

#### View-Navigation Diagram

**Guideline [development]:** The primary purpose of the *view-navigation diagram* is to identify the views which can implement multiple sets of related features and denote the different routes to navigate to them for different actors (refer to section 4.4.4.2). It is crucial to identify the multiple features developed in each view, the actors of these features, and the various navigation paths to these views, and then design them using the *view-navigation diagram* to assist in realising the arrangement of all the views in a RiWA towards reducing the complexity of the development.

**Guideline [development]:** The *view-navigation diagram* captures all the views to be developed in the RiWAs and the navigation between them (refer to section 4.4.4.2). There cannot be any diagram of view not included in the *view-navigation diagram*. However, it is sufficient to design only the required views using the *view diagrams*, and it is not necessary to design all the views available on the *view-navigation diagram*.

**Guideline [development]:** Navigation links starting from hyperlink GUI elements are always link-based navigation and developed as standard hyperlinks on the view. Navigation links starting from other types of GUI elements are process-based and should be implemented in the controller of the view, which includes the navigation starting GUI element.

### AppControllers Diagram and Controller Diagram

**Guideline:** It is not mandatory for all the views to have a dedicated controller; for example, views with only readable content, like a help page, do not necessarily need a controller. Therefore, the number of controllers in the *AppController diagram* can be less than the number of views in the RiWA. It is effective to identify which views require a controller and denote this detail in the design.

**Guideline:** It is beneficial to design the controller with its related view on the same diagram.

**Guideline:** Decide the rich features to be developed on the view, identify all the event handlers required on the controller to implement them, and include them on the design with notes explaining their functionalities.

**Rule:** Even though the parent *AppControllers element* of the *AppController diagram* is a type of component, the parent *AppControllers element* or the *controller* elements in it do not use UML's standard interfaces; instead, they should always use the RiWAsML communication channels.

**Guideline [development]:** Do not mix event handling code with the view's code; write code to register event handlers on the controller.

### AppModel Diagram

**Guideline:** Each AppModel comprises a complete class diagram.

**Rule:** An *AppModel diagram* should use a parent *model component* element.

**Rule:** Even though the *model* is a type of *component*, it does not use UML's standard interfaces; instead, the model component and its classes should always use the RiWAsML communication channels to indicate the communication with the outside.

**Guideline [development]:** Even if the client-model is developed using a language like JavaScript without OOP practices, develop the ModelClasses in separate files as depicted on the design.

**Guideline [development]:** Some web server-side frameworks provide a concept named “model” for database Object Relational Mapping/Model (ORM) or Object Data Model (ODM) implementations [170] [171]. These ORM/ODM models are different from the RiWAArch style's concept of model, which is based on the MVC. The ORM/ODM models can be considered a part of the RiWAArch style's model, which assists in implementing the database-related functionalities.

### DC-bus Diagram and EndpointsCollection Diagram

**Guideline:** Even though the RiWAsML considers the connector a type of component, using the component symbol on the connector elements is not mandatory to maintain the visual separation between the connectors and components.

**Guideline [development]:** A web service API development concept like SOAP [144] or REST [145] can be used to develop the DC-bus. Consider the relevant rules and guidelines of the selected API development technology when designing and developing DC-bus and EndpointsCollections.

**Guideline [development]:** Web server-side development frameworks typically use the term “controller” [139] for the DC-bus-related aspects and “route” [172] [173] [174] for the endpoint, which are derived from MVC and REST. Be aware of these terms and do not substitute them with the RiWAArch style’s controller.

**Guideline [development]:** Multiple EndpointsCollections can be developed on the same class/script where necessary.

**Guideline [development]:** Select a consistent data wrapping technique/technology like XML or JSON for the endpoints and denote that on the *DC-bus diagram* and/or *EndpointsCollection diagram* using notes.

#### View-Process Sequence Diagram

**Guideline:** Wrap the swimlanes of the same Application element using an Application element.

**Guideline:** The Application elements and their view, controller, connectors, and models should be aligned with the other design diagrams.

#### Other Related Design Concerns

**Guideline:** Use a suitable designing method/tool like Entity-Relationship diagrams (ER diagrams) to design the databases.

**Guideline:** Other standard UML diagrams, like *network architecture diagram* and *activity diagram*, shall be used as required. However, it is advisable to use the RiWAsML label on the UML elements in a suitable way to align with the rest of the RiWAsML diagrams.

## Appendix D.2. Expert 1 Feedback

### Evaluator Details

1. **Name:** Agata Kulczynska
2. **Email:** agata.kulczy@gmail.com
3. **Bio:**

Agata Kulczynska graduated in 2021 with a B.Sc. in Computer Science from the University of Westminster, London, with an award for the Best Final Project. She also received an award for being One of the best-performing female graduates of the Computer Science and Engineering department at the University of Westminster. Agata obtained an M.Sc. in Artificial Intelligence in 2022 from the Queen Mary University of London.

From 2020 to 2023, Kulczynska worked part-time as an Information Technology Support officer at Elmech Ltd, Research Associate Software Developer at the University of Westminster, and full stack developer at Dromedaware Ltd. Since Jan 2023, she has been working as a software engineer at Clear Channel UK.

Agata is experienced in designing and implementing rich web-based software solutions using a wide range of technologies. She has primarily worked on developing a knowledge repository that explored novel ways of storing information in the form of graphs and presenting it to students named Smartest, as well as a web-based application that introduced digitalisation and automation within a medium-sized construction company. Currently, she is working on an application that encompasses a variety of functionalities for one of the leading out-of-house media companies in the United Kingdom.

4. **LinkedIn (or similar professional) profile:** <https://www.linkedin.com/in/agata-kulczynska/>
5. **Do you give consent to publish the details provided in this section? (yes/no)** (if there is any detail you don't want to publish, please mention): yes

**Element Names and communication channels**

<b>Diagram and its elements</b>	<b>Usefulness</b>	<b>Justification</b>
<u>Element names (label)</u>	5	The proposed structure of the elements' names is clear and should be understood by a reader who is generally familiar with the UML specification. Separating the label into Element, Type, and Name segments makes it easier for the reader to gain an understanding of the individual elements while analysing a complex architectural diagram.
<u>Communication channels</u>		
-Standard communication	5	A standard arrow element should be familiar and understandable for most of the readers, making the direction of the communication clear and ready to read.
-Delta communication	5	As above, a standard arrow element should be familiar and understandable for most of the readers, making the direction of the communication clear and ready to read.
-View-controller communication	5	The view-controllers communication is clear and distinguishable from both the standard and the delta communication notations.
-Method call and return	5	As above, the method call and return is clear and makes it easy to understand the flow of the execution. The boldness between the view-controller and the method call and return seems to be more distinguishable than in the case of the standard and delta communication, making it easier to differentiate between the different notations for the reader.

**High-level Model and Elements Evaluation**

<b>Diagram and its elements</b>	<b>Usefulness</b>	<b>Justification</b>
<u>Architecture diagram</u>	5	A very high effect score on the usefulness scale was assigned since the presented diagram outlines a high-level architecture of the system while providing a sufficient and relevant amount of details regarding the constituting application's components and their types, giving an

		overall broad view of the discussed software solution and suggesting a high comprehensiveness.
Tier element	5	Tier elements allow the reader to gain an initial understanding of the application's architecture while providing the fundamental details regarding each segment. Categorising tier elements into <i>Presentation</i> , <i>Application</i> , and <i>Storage</i> seems to be an adequate technique to define the main building blocks of the application, while the <i>External</i> label allows the outline of any other crucial elements which do not fall into one of the aforementioned categories, such as <i>Authentication Service</i> .
Platform element	4	Platform elements seem to provide additional details regarding the environment in which the application is running, as well as the primary programming language utilised. This information is valuable and allows to gain further understanding of the system's design and architecture, which has a positive overall impact on the development support.  However, to fully appreciate the usefulness and complexity of the discussed element, it would be beneficial to present a diagram in which a few executable applications run within one platform and potentially outline interactions between them.
Application element	4	The application element seems to be a relevant building block within which further split into views, controllers, models, and delta communication elements can be captured effectively utilising the proposed methodology. Outlining the type of the application enables to understand whether it is a web-based, mobile, or desktop solution.  However, as mentioned above, it could be valuable to present a design in which a few executable applications run and interact within one platform. For instance, it would be interesting to see a diagram expanded for a solution including several microservices, which could potentially increase the overall learnability and comprehensiveness.
Views element	5	Views elements allow to identify which depicted applications contain views, as well as to determine the type of the represented views, which seems to be an appropriate level of detail to include in a high-level model of a system.
Components element	5	Component elements appear to be valuable to outline further specifics regarding the relevant application and include details of the implemented models and controllers, as well as their types. Through further examination of the different types, it can be understood that



		components encompass the logical processing segments and are mostly focused on the models and controllers.
Connectors element	4	<p>Connector elements allow the reader to understand how the applications communicate with each other via utilisation of the DC Engines and DC Buses. They also seem to be useful to present the utilised protocols and any potential external APIs.</p> <p>With modern solutions such as message queuing services, it would be interesting to see how some alternative communication means could be represented on the diagram, too.</p>
Other elements (database, external services)	5	Other elements allow to capture any additional crucial modules which could not be represented with the basic elements. Thus, I believe they have an overall high effect on the design of the diagram.

**Low-level Models and Elements Evaluation**

<b>Diagram and its elements</b>	<b>Usefulness</b>	<b>Justification</b>
<u>View-navigation diagram</u>	4	<p>The view-navigation diagram provides a broad and detailed insight into the views included in the application, their functionalities, relevant actors, as well as the navigation paths between them.</p> <p>In the future, in order to increase comprehensiveness and learnability, it could be valuable to provide a guideline regarding designing and selecting the most relevant views for the View-navigation diagram, as with the more advanced systems, it might become highly complex for the engineers.</p> <p>Nonetheless, the diagram is clear and effective in portraying the views of the system and the navigation between them. The variety of the elements allows to represent crucial segments of modern web-based applications, as well as to group them for increased readability and simplicity. Thus, a high effect score from the usefulness scale was assigned.</p>

		<p>However, as elaborated on in the second paragraph, the usefulness could potentially be increased by providing more guidelines regarding the most effective selection of the views and their corresponding elements. Such solution would potentially make it easier for the engineers to capture the complexity of the designed system, and increase the usefulness from 4 to 5.</p>
View element	5	View element is clear and provides the reader with an insight into the type and the functionality of the view.
View package element	5	View package element serves as an efficient way to define which views are related, having a significant impact on the simplicity aspect by introducing additional modularisation.
Viewpart element	5	Viewpart element can be assigned a very high effect score, as it allows to define the GUI elements that are the most crucial and outline which segments can be utilised for the navigation purposes between the different views.
SharedViewpart	5	I believe that the SharedViewpart elements increase the general simplicity and readability of the solution, as they allow to omit unnecessary repetition of the individual GUI elements.
SharedViewpartOBJ	2	The main characteristics of the SharedViewpartOBJ elements were not always that easy to distinguish from the SharedViewpart elements for me, as they can share the same types and names, impacting the overall readability.
ViewI/ViewO/ViewIO	4	<p>Defining inputs and outputs between the views can prove as an efficient way to allow the reader to gain a further understanding of the navigation within the system, provided that the types and naming of all the I/O elements is clear and well-defined.</p> <p>I believe that the ViewI/ViewO/ViewIO elements are valuable, however not as crucial for the engineers to understand the overall design as the other elements which were assigned score 5. Thus, relatively, a high effect from the usefulness scale was given.</p>

**View Diagram and Controller Diagram**

<b>Diagram and its elements</b>	<b>Usefulness</b>	<b>Justification</b>
<u>View diagram</u>	4	<p>The View diagram provides a comprehensive representation of the relevant view element, giving the reader a chance to gain further understanding of the view’s functionalities as well as the crucial GUI segments, including the input and output elements.</p> <p>For an increased learnability, it would be valuable to provide a set of guidelines regarding the selection of the most significant interface elements which should be covered within the view diagram.</p> <p>Nonetheless, the current design offers a broad selection of elements which allow to capture the significant internal elements of the view in an efficient and clear way. Therefore, a high score of usefulness was assigned.</p> <p>As pointed out in the second paragraph, learnability could be increased by providing an additional set of guidelines regarding the elements’ selection. In my view, elevating learnability would then increase the overall usefulness of the diagram to 5.</p>
-View element	5	View element can be utilised to provide the fundamental details regarding the view, such as its type and name, therefore it was assigned the highest score from the usefulness scale. It impacts the general readability and comprehensiveness of the system.
-View package element	5	ViewPackage element increases the readability and simplicity, as it allows to categorise the views, and then define which group the given view belongs to. Such solution makes it more efficient to design a system that is modular and scalable, as well as to understand the relationships between the functionalities of the individual views.
-Viewpart element	5	Viewpart element provides a seemingly efficient way to outline the most significant parts of the GUI, their role within the application, as well as the associated actors,

		which is important while representing the systems in which various users can perform different actions.
- ViewI/ViewO/ViewIO	5	The discussed elements are valuable in representing the different input and output actions associated with the represented view. I believe they play a significant role in increasing the aspect of the development support, as they provide additional information regarding the I/O flow.  A very high effect on the usefulness scale was assigned since the discussed elements allow to capture event handling within the view.
<u>Controller diagram</u>	5	The controller diagram has a very high impact on the usefulness scale, as it allows the reader to understand what event handlers are required and triggered by the view. Not only does it increase the comprehensiveness and readability, but also the development support.
-Controller class element	5	I believe the controller class element to be highly useful and valuable to gain a deeper understanding of all the actions that can be triggered from the view. It influences the overall comprehensiveness, readability, as well as the development support, making it easier to pinpoint what actions should be activated by individual views' elements.
<u>View-Controller diagram</u> (as a single diagram)	5	Taking everything into consideration, the View-Controller diagram seems to have a significant impact on the overall quality attributes of the designed system, especially the comprehensiveness, readability, and development support. It provides a relevant level of details regarding implementation of the view, as well as its most crucial GUI elements and the associated triggers. Additional notes provide a supplementary, valuable insight into the view's functionalities.

**DC-Bus Diagram and Endpoints Collection Diagram**

Diagram and its elements	Usefulness	Justification
<u>DC-Bus diagram and Endpoints collection diagram</u>	4	DC-Bus diagram appears to be an efficient method to better structure the API, therefore increasing the modularity and simplicity. The Endpoints Collection Diagram provides details regarding a class of endpoints in an efficient way. However, the Endpoints Collection Diagram could potentially benefit from the inclusion of some additional details, as elaborated on in the relevant table section. Thus, the mark 4 was assigned.
-Connector element	5	The modularity and simplicity is greatly increased by structuring the API into more granular classes and representing them on a comprehensive and easy-to-understand diagram.
-Endpoints class element	4	The endpoint class element appears to prove to be a good representation of a class of endpoints.  However, comprehensiveness and development support could potentially be increased by introducing a higher level of detail for each endpoint, with some additional information regarding the request methods (as the method name might not always state it) or media types.

**Application-Model Diagram and Model Class Diagram**

Diagram and its elements	Usefulness	Justification
<u>Application-model diagram and Model class diagram</u>	5	Both application-model and model class diagrams appear to be valuable in outlining the system’s architecture. I believe it is of great significance to provide the reader with a detailed understanding of the classes building the application and their relevant functions; thus, the highest score on the usefulness scale was assigned.
-Component element	5	In my view, outlining the main classes within the model has a positive impact on an increased modularity of the designed system. Furthermore, I believe it influences the growth in readability by providing the user with a broad view of the included classes, along with the connections between them.

-Class element	5	Class element influences readability and understandability by providing more details into an individual class from the application-model diagram. Outlining individual methods, their outputs and parameters appear to be beneficial for development support, potentially making it easier to implement the designed classes.
----------------	---	---

**View-Process Sequence Diagram**

<b>Diagram and its elements</b>	<b>Usefulness</b>	<b>Justification</b>
<u>View-process sequence diagram</u>	5	In my view, sequence diagrams are generally highly valuable in supporting the process of representing and explaining the communication means between the different segments that form the system, as well as providing further details regarding the exchanged messages. I believe that it is of great importance to depict all the core elements that are included within the communication process so as to aid the readability and the development support. Therefore, all the listed elements were assigned a score of 5 on the usefulness scale.
-Application element	5	Application appears to be one of the core building blocks outlined in the diagram. Grouping individual elements into application segments advantageously impacts both simplicity and comprehensiveness.
-View element	5	The view element seems to aid in representing the possible actors' interactions with the application and efficiently depict the following flow of the communication process.
-Controller element	5	Controllers allow the reader to gain a deeper understanding of what actions are triggered by individual views and how they are communicated further to the services.
-Endpoints element	5	EndpointsCollection elements support the development and further understanding of the system by providing relevant details regarding the utilised API.
-Model class object element	5	The model class object element appears to play an important role in understanding the flow of communication and depicting what kind of information is expected to be returned.

		The model class object element is readable and encompasses all the attributes required to gain a fundamental understanding of the represented class, such as its type, name, properties, and their types, as well as the methods with their signatures.
--	--	---

**Evaluation of the RiWAs Design Methodology (RiWAsDM)**

<b>Quality attribute</b>	<b>Opinion</b>	<b>Justification</b>
Simplicity	5	In my view, the methodology greatly supports the separation of concerns by decomposing the system while adopting the top-down design approach. I believe that the introduction of elements such as the DC-Bus and DC-Engine further aid the modularisation and decoupling, ensuring increased efficiency of the further management.
Comprehensiveness	4	I believe that a high effect in comprehensiveness was achieved through the introduction of a wide variety of models and model-elements, allowing to efficiently design a rich web-based application. The introduction of the DC-Bus and the DC-Engine elements further complements comprehensiveness by allowing to design the aspects realised by the RIWAArch style. The core aspects, rules, and guidelines supplied in Appendix 10 are clear and concise.  To further aid the discussed quality attribute and support potential users to prepare the design and then map it into the development process, it could be useful to see more examples of designs of rich web-based applications, for instance, the ones that are built out of multiple micro-services.
Learnability	4	As stated in Section 3, the presented design methodology is based on UML and the RiWAArch style, and therefore, it should be clear to understand for a reader who is familiar with the general UML specification. The rules and guidelines outlined in Appendix 10 are concise and easy to follow.  Some additional guidelines could be provided when it comes to building more complex diagrams, which

## Appendices

		require the inclusion of a higher level of details and selection of the most crucial components, such as the diagrams focused around the Views and the Navigation.
Readability/understandability	5	In general, the models and the model-elements contain the right amount of detail, suitable for the type of diagram they belong to. The notation is concise and outlined in a clear way. Owing to this, it should be easy for the reader to gain an understanding of the design and the functionalities of the individual elements.
Development support	5	<p>In my view, the presented models and model-elements include enough details to allow for an efficient mapping into actual development.</p> <p>For the potential further work focused on the more advanced systems, it could be helpful to include more design elements which could guide the developers in making decisions concerning other crucial aspects of the rich web-based applications, focused around the cloud solutions or security components.</p> <p><b>NOTE:</b> Within the context, all the diagrams were clear and included a variety of elements and a great level of details, which supports the development very highly.</p>
Integrability	4	The methodology introduces modularisation and decoupling of components, which should aid an iterative approach to implementation and support agile model-driven development.

### Overall Feedback

<b>Overall rating on the RiWAsML/RiWAsDM</b>	4
<b>Overall feedback and justification.</b>	
<p>The methodology demonstrates robust strengths in simplicity, comprehensiveness, and readability. Its systematic approach, employing top-down design principles alongside elements such as the DC-Bus and DC-Engine, effectively facilitates system decomposition and</p>	



modularisation. The inclusion of a variety of models for designing rich web-based applications, coupled with concise rules and guidelines, further increases the general quality attributes.

Regarding learnability, its reliance on UML and the RiWAArch style ensures clarity, especially for readers familiar with the UML specifications. However, there's an opportunity for improvement in providing additional guidance and further examples of more varied, complex high-level diagrams. This augmentation would enhance understanding and practical application.

Furthermore, the methodology's models strike a balance between detail and clarity in notation, supporting readability and making it easier for readers to understand the functionalities of individual elements.

The room for enhancement could potentially be found in the comprehensiveness and the development support. By incorporating more design elements catering to critical aspects such as cloud solutions and security components, the methodology could offer more substantial guidance, aiding developers with clearer insights during the implementation phase. Moreover, it would be interesting to see the introduced elements of the DC-Bus, and the DC-Engine expanded for different types of communication, for instance, messaging queues.

### Appendix D.3. Expert 2 Feedback

#### Evaluator Details

1. **Name:** David Chan Fee
2. **Email:** chanyod1@westminster.ac.uk
3. **Bio:** Research Associate

David Chan Fee obtained a B.Sc. in Computer Science with First Class Honours from the University of Westminster in 2018. During the undergraduate program, David worked as a Support Staff member at the Sherpa Event Support & Logistics from 2016 – 2017. Then, he worked as a student helper, research assistant, and KTP Associate at Lumina Learning at the University of Westminster. Since 2020, David has been working as a Research Associate at the University of Westminster.

During David’s employment, he worked as a software engineer, developing web-based systems using various technologies and tools. Recently, he contributed to engineering an online graph-based learning tool called “SMARTTEST” as the team lead and a full stack developer.

4. **LinkedIn (or similar professional) profile:** <https://www.linkedin.com/in/david-c-2533b3148/>
5. **Do you give consent to publish the details provided in this section? (yes/no)** (if there is any detail you don’t want to publish, please mention): Yes

#### Element Names and communication channels

Diagram and its elements	Usefulness	Justification
<u>Element names (label)</u>	4.9	All elements seem to be appropriately named. Regarding ViewPartOBJ and SharedViewPartOBJ, to me, the “OBJ” wording seemed a bit too abstract and I thought another more descriptive word could be used instead, like “Region” or “section”; I may be wrong as I may not have as clear a picture behind the use of OBJ. I also think the use of capitalization and abbreviation of “Object” to “OBJ”, doesn’t quite fit in with the rest of the naming style used for the other elements.
<u>Communication channels</u>		

## Appendices

-Standard communication	5	Simple, straightforward.
-Delta communication	5	Simple, straightforward. Different enough to be distinguishable from the Standard communication notation. Based on my own experience, I'm unsure of the meaning behind the "delta" wording and how it relates to the nature of the communication (which covers AJAX communication, but also socket communication, I assume). This may be something to consider when introducing this aspect to new developers of a similar expertise/experience level of my own (e.g. adding in parentheses the nature of the communication beside the "delta communication" label; I also haven't deducted points as this is probably more of a difference in our personal experiences than a general issue with the label).
-View-controller communication	5	Simple, straightforward.
-Method call and return	5	Makes sense, nice and simple.

## High-level Model and Elements Evaluation

Diagram and its elements	Usefulness	Justification
<u>Architecture diagram</u>	5	It's easy to understand what the diagram is describing. I like that the components of each layer are visually organised into tiers, describing intuitively that components in the same tier share similar roles. I also like that the architecture is able to be modelled to this level using just RiWAsML notation, where it would be less clear on how to approach this with base UML. The role, technology used and medium for each component is succinctly described in a predictable and easy to understand way.
Tier element	5	Visually groups platforms together, clarifying which platforms are related and which are not. The annotations help understand a platform's role and where that platform is located within the overall architecture.

Platform element	5	Gives context on the software environment/framework and hardware platform an architecture element will be running on.
Application element	5	Helps with understanding the type of package an application will be delivered through.
Views element	5	Quite clear on the type of view being used, and what they are for. It's nice to know that there is a bi-directional communication channel between the views and the client controller, and also, when communication doesn't have to go through the client controller (in the case of the ClientWebPages view, which I assume may represent form submission).
Components element	5	Gives details on the type of component. It's helpful to know what qualifies as a "component", differentiating between views and connectors. Using this element also helps to clearly and concretely capture communication of one component to another element, which helps provide lower-level communication details between two platforms. It's also helpful that components may also contain other elements (like the connector in the controller component).
Connectors element	5	Makes it easy to understand which part of an element (e.g. controller, application, platform) will be mainly responsible for communicating and interfacing with other elements within the architecture.
Other elements (database, external services)	5	These elements are not visually grouped with the main body of the architecture diagram, making it easy to understand these are external components.

### Low-level Models and Elements Evaluation

#### View-Navigation Diagram

Diagram and its elements	Usefulness	Justification
<u>View-navigation diagram</u>	5	The diagram provides an easy to understand view of how to navigate between views. Going into more detail, it's helpful to

		know which pages use the public menu and which use the registered user menu, and which pages will have which menu present on their page. The different shapes of the elements also help to visually distinguish them apart from each other. The overall visual organisation in the diagram which helps to group together the pages for different menus.
View element	5	Makes clear the type of view and the view's name.
View package element	5	Makes it clear that there are multiple entrypoints to a view, and helps to distinguish itself from a "View" element.
Viewpart element	5	Makes it clear which view parts are for specific user groups (e.g. which view parts are for admins, which view parts are for teachers) and as a result, also which view parts are for all users.
SharedViewpart	5	Makes clear which view parts are shared between different groups of users and the name and role of the view part.
SharedViewpartOBJ	5	There to just make clear which SharedViewParts are present within which webpages, and providing just enough information without detailing all the smaller subparts of that SharedViewPart.
ViewI/ViewO/ViewIO	5	This group of elements make clear what is intended for input, output and both input and output.

### View Diagram and Controller Diagram

Diagram and its elements	Usefulness	Justification
<u>View diagram</u>	5	Annotations make it easy to understand which ViewI elements map to which controller methods. These annotations also don't seem unobtrusive if you don't know what they mean beforehand. The text box is nice for adding overall context to the diagram.
-View element	5	Nice as it groups all view-related parts together and describes the type of view and its name.
-View package element	5	Found it helpful to know that the ViewI elements are also a part of another view, helping to indicate that they will most likely be handled by the same controller or in a similar manner.

Appendices

-Viewpart element	5	Makes it clearer which view parts are for specific user groups and indirectly, which view parts are for all user groups.
- ViewI/ViewO/ViewIO	5	Helps to clarify which view parts are interactive and which are not (and which views contain interactive and non-interactive elements).
<u>Controller diagram</u>	5	Familiar format to base UML; easier to understand and parse as a result. Annotations make it easy to understand which ViewI maps to which controller method and if certain methods happen in sequence.
-Controller class element	5	Like base UML, so easier to understand. The “x : y : z” annotation helps clarify the role, type and name of the controller.
<u>View-Controller diagram</u> (as a single diagram)	5	It’s easy to see which parts are for the view and which parts are for the controller.

**DC-Bus Diagram and Endpoints Collection Diagram**

<b>Diagram and its elements</b>	<b>Usefulness</b>	<b>Justification</b>
<u>DC-Bus diagram and Endpoints collection diagram</u>	5	Nice to see an overview of how the routes map together and the routes for each user group.
-Connector element	5	Gives a nice hierarchical overview of how the routes map together. Simple, but serves a purpose.
-Endpoints class element	5	Like base UML, so generally easy to understand.

**Application-Model Diagram and Model Class Diagram**

<b>Diagram and its elements</b>	<b>Usefulness</b>	<b>Justification</b>
<u>Application-model diagram and Model class diagram</u>	5	Have a generally clear idea of which endpoints and ServerModels will map to which components.
-Component element	5	Nice to see how different components map together in the overall system. Follows base UML conventions, so could understand those parts.
-Class element	5	Easy to understand which endpoints map to which ServerModel methods.

**View-Process Sequence Diagram**

<b>Diagram and its elements</b>	<b>Usefulness</b>	<b>Justification</b>
<u>View-process sequence diagram</u>	5	The diagram gives a nice comprehensive overview of the login process. The trigger communication channel from the login button is a nice detail. The length of the lifelines also make sense.
-Application element	5	Nicely segments different parts into groups, making it clear which parts belong to which application. The colour coding is appreciated. The annotation at the top also makes it clear the type of application being described and the name of the application.
-View element	5	Easy to understand the type of view and the view's name.
-Controller element	5	Easy to understand the type of controller and the controller's name.
-Endpoints element	5	Easy to understand which user group the endpoint collection is for and the endpoint collection's name.
-Model class object element	5	Easy to understand the type of object and what it will be used for.

**Evaluation of the RiWAs Design Methodology (RiWAsDM)**

Quality attribute	Opinion	Justification
Simplicity	5	I felt that the RiWAsDM was able to capture details and aspects of software architecture that is not normally captured with base UML (e.g. Tiers and Platforms). It also did so in a generally simple and non-obtrusive way (e.g. the connectors help avoid visual complexity as they avoid drawing physical connections across a diagram from one remote point to another).
Comprehensiveness	5	Based on my experience, I can't say what the RiWAsDM may have missed, and so, I assume that it covers a good range of aspects for helping build and model RiWAs. I liked that the rules and guidelines covered a basic set of scenarios, but also went into more specific aspects like a view diagram only having to contain the GUI elements necessary to describe a functionality, or that a view doesn't need to have a corresponding controller, where appropriate. The numbered connectors also help cover and treat things like function execution order neatly, which goes to show it is well-designed and fits in with the overall philosophy of this design methodology easily. Just for comprehensiveness, in Appendix 9, perhaps the "Label" element in the table could have some descriptive text in the "Type" column.
Learnability	4.9	New notation was relatively easy to grasp (e.g. numbered connectors, the dotted styling for return arrows in sequence diagrams). I have given a rating of less than 5 as I think the arrow notations should be more visually distinct to help make them easier to remember, and therefore, learn. Based on my experience, I'm not sure of clear way to achieve this, however.
Readability/understandability	4.9	The notation became easier to understand the more diagrams I looked at. Despite not understanding what each bit of notation meant (e.g. the Trigger communication channel notation), this did not



	<p>interfere with the overall understanding of diagrams (i.e. unknown notation was not visually distracting and the nature of an action could be gleaned from other details – e.g. a button usually triggers a call via an event, despite the notation being able to provide these details if understood). When more familiar with the notation, the small details are definitely appreciated – the trigger communication channel is a nice distinction, as it helps highlight when communication starts, as opposed to just showing communication flow, and the numbered connectors provide a clean way of relating two remote regions together, as it avoids awkwardly drawing across the diagram, which would interfere with other elements. The numbered connectors also provide a succinct way to describe function execution order, which looks better than using notes, for example. Regarding the score being less than 5, I think the rules and guidelines section of Appendix 10 would benefit from having visual examples positioned beside them to help communicate and clarify the meaning of certain points. Also in Appendix 2, when first experiencing the variety of arrow notation (without the aid of the reference) I found it wasn't clear and a bit confusing why some arrows were bold and why some also used a combination of bold and a different arrow head type; because of this, I feel that if visual distinctions are being made between arrows, that a non-obtrusive visual aid may be helpful (e.g. a key), or, the visual distinction should be clear enough so that anyone looking at the diagram with fresh eyes could guess it easily (although in a project setting, perhaps team members will have enough knowledge and context to know what certain parts in the diagram will do and how they will work, but this also means context is an important part to provide alongside diagrams, perhaps). In Appendix 3, I thought the element groups could be spaced further apart from each other, to help with easier initial parsing, and to also make notations</p>
--	--

		like the numbered connector and the arrow notation from login button to dashboard, easier to notice.
Development support	5	Based on experience with the RiWAsDM in applying changes to SMARTTEST, these can generally be mapped one-to-one – although I acknowledge that this experience is limited and that the RiWAsDM should be tested against more and different types of RiWAs to get a more accurate gauge of its scope and applicability.
Integrability	5	The RiWAsDM was able to be integrated with SMARTTEST on some level despite not being used for original development. I can see the RiWAsDM being integrated smoothly within RiWA software development, using diagrams appropriate for each stage of the software development life cycle (e.g. using higher-level diagrams like an Application-Model Diagram at early stages of development and a View-Controller Diagram for later stages of development).

**Overall feedback**

<b>Overall rating on the RiWAsML/RiWAsDM</b>	5
<p><b>Overall feedback and justification.</b></p> <p>The RiWAsDM provides a more detailed and succinct way to model RiWAs than basic UML – e.g. aspects such as mapping view GUI elements to controller functions are handled elegantly with numbered connectors. The DC-Bus/DC engine helps to capture the AJAX aspect of modern RiWAs and distinguish it from regular HTTP communication; this also helps to endorse good software design, especially at the stage of mapping design to development, by clearly abstracting AJAX-based functionalities as key aspects of the overall architecture. Although the meaning of some notation is not immediately clear (e.g. the various arrow notations), this does not affect the general understanding/communication of a diagram and can be supplemented by having a reference at hand when viewing a diagram.</p>	

## Appendix D.4. Expert 3 Feedback

### Evaluator Details

1. **Name:** Uthpala Samarakoon
2. **Email:** Uthpala.s@sliit.lk
3. **Bio:**

Uthpala Samarakoon obtained a B.Sc. (Hons) in Information Technology in 2007 from the Sri Lanka Institute of Information Technology. She obtained a M.Sc. in Information Management in 2010 from the Sri Lanka Institute of Information Technology. Uthpala submitted a thesis on the title *Incorporating Usability Factors in the Digital Didactical Design of Tablet-based Learning in Early Primary Education* in 2024 to complete an M.Phil. at the University of Colombo, School of Computing, Sri Lanka.

Uthpala has been working as a lecturer and a senior lecturer in the field of information technology and software engineering since 2007 at the IT department of the Sri Lanka Institute of Information Technology. She teaches UML-based software design and has supervised undergraduate research projects in software engineering for over ten years.

Uthpala Samarakoon’s research interests are on E-Learning, HCI and their applications, IT Education, and Knowledge Management.

4. **LinkedIn (or similar professional) profile:** <https://www.linkedin.com/in/uthpala-samarakoon-76a51439/>
5. **Do you give consent to publish the details provided in this section? (yes/no)** (if there is any detail you don’t want to publish, please mention): Yes

### Element Names and communication channels

Diagram and its elements	Usefulness	Justification
<u>Element names (label)</u>	5	The clear, UML-compliant naming structure (Element, Type, Name) simplifies understanding individual elements within complex diagrams.
<u>Communication channels</u>		
-Standard communication	5	Simple, familiar arrow symbols ensure communication direction which is immediately grasped by readers.

-Delta communication	5	Arrow symbols directly convey communication direction for most readers. So, clearly defined and easily understandable
-View-controller communication	5	The interaction design for view controllers offers a fresh and innovative approach, different from the conventional communication styles.
-Method call and return	5	Both the method calls and returns are demonstrably clear, facilitating an effortless understanding of the execution flow.

### High-level Model and Elements Evaluation

<b>Diagram and its elements</b>	<b>Usefulness</b>	<b>Justification</b>
<u>Architecture diagram</u>	5	Architecture diagrams are important to visualize the structure, components, and interactions within a web application. It included various aspects of development, maintenance, and communication. Introducing an architectural diagram for UML is important. Hence, web developers can get a concise overview of the system's design.
Tier element	5	By dividing the system into separate layers (tiers), the diagram visually represents the major functional areas and their interactions. This makes it easier to understand the overall architecture and how different parts work together.
Platform element	5	Platform elements in an architectural diagram play a crucial role in showcasing the foundation upon which the entire system operates. This element displays the underlying technologies and software components that support the application. Hence, it is very useful.
Application element	5	The application element is the foundation of an architectural diagram, representing the core business functionality. The application element visually shows the system's intended functionalities, clearly communicating its purpose and value to stakeholders. Hence, it is useful to understand how the system interacts with users, processes data, and delivers its intended outcomes.
Views element	5	The "views" element in an architectural diagram carries significant importance as it provides different perspectives on the system. It provides for the needs of varied stakeholders and is useful to get a

		comprehensive understanding of its structure and functionality. Each view focuses on specific aspects and simplifies understanding for different audiences.
Components element	5	This serves as the building blocks of the system, playing a key role in visualizing its structure, functionality, and interactions. Components break down the system into smaller, well-defined units, each compressing specific functionalities and responsibilities. This helps in understanding complex systems, promotes better maintainability, and simplifies development and modification processes.
Connectors element	5	This conveys how different components interact and communicate within the system. Connectors clearly illustrate the relationships and dependencies between components. This is crucial for understanding how data flows throughout the system, identifying potential bottlenecks, and ensuring smooth communication pathways. So I find it this is useful.
Other elements (database, external services)	5	These elements carry significant importance for understanding the system's data storage, retrieval, and interaction with external entities. Databases represent the system's persistent data storage mechanism. Knowing the chosen database technology and external dependencies is useful for decision-making regarding performance optimization, cost management, and potential scalability limitations.

## Low-level Models and Elements Evaluation

### View-Navigation Diagram

Diagram and its elements	Usefulness	Justification
<u>View-navigation diagram</u>	5	View navigation in a web application plays a significant role in user experience (UX) and the overall success of the app. It helps users by guiding them to the information and functionalities they need efficiently and without frustration. Currently, UML does not support view navigation diagrams. Hence this will support developers to get a better idea about different views of the system and relationships.

View element	5	This newly introduced view navigation UML diagram shows a good level of visual hierarchy and grouping of elements. The diagram covers the main view elements and their relationships too. Different users (developers, business analysts, managers) have different needs and levels of technical expertise. Views ensure specific information is tailored to each user group, enhancing comprehension and usability.
View package element	5	The view package element serves as a container for grouping related views that share a common purpose. This helps to organize the diagram and improve its clarity, especially for complex systems with many views. Hence, this is useful to improve the clarity of the diagram.
Viewpart element	5	The viewpart element is useful to identify a view for a particular actor. So in development, it is easy to get an idea about the different views of various actors.
SharedViewpart	5	This combines multiple sharable views for a particular actor. This is useful for identifying the views that belong to major user categories. It improves the clarity of the diagram too.
SharedViewpartOBJ	2	The purpose of this is not very clear.
ViewI/ViewO/ViewIO	5	These elements are useful to identify the input and output of different views and the navigation between them.

### View Diagram and Controller Diagram

Diagram and its elements	Usefulness	Justification
<u>View diagram</u>	5	View diagrams in rich web app development are like blueprints for its UI. They visually communicate about layout, functionality, and data flow, ensuring clear understanding. This increases collaboration, and guides for a user-friendly, efficient application. Hence, introducing a view diagram to UML is a good attempt.
-View element	5	Provide a direct idea about an element's name and type. This helps to understand the entire system.

-View package element	5	It enables the grouping of views based on shared characteristics, ultimately outlining which category each view belongs to. This approach fosters the development of modular and scalable systems, where individual views play distinct roles within a clearer functional structure. Understanding these relationships becomes effortless, leading to a more intuitive and maintainable system.
-Viewpart element	5	The Viewpart element offers a clear and concise way to map out the core GUI components, their purpose within the app, and the user roles interacting with them. This insight proves invaluable for representing systems with diverse user groups and varying actions.
- ViewI/ViewO/ViewIO	5	These elements offer a value in various input and output actions linked to the view they represent. These elements significantly enhance development support by providing a clear picture of the information flow, making development more efficient and comprehensive.
<u>Controller diagram</u>	5	Controller diagram is important to understand the system. They show which events trigger specific responses, making development and understanding much easier.
-Controller class element	5	Controller classes offer a deep understanding about view actions, enhancing comprehension, readability, and development support by clarifying which actions each view element triggers.
<u>View-Controller diagram</u> (as a single diagram)	5	A View-Controller Diagram is important in web application development due to its ability to visually represent the relationships and interactions between the UI components (views), the application logic (controller), and the underlying data (model).

**DC-Bus Diagram and Endpoints Collection Diagram**

Diagram and its elements	Usefulness	Justification
<u>DC-Bus diagram and Endpoints collection diagram</u>	5	Introducing a DC-Bus Diagram for rich web apps could bridge the gap between data flow and UI by visually mapping how backend events trigger specific UI updates, improving communication, optimizing performance, and simplifying maintenance. This will lead to smoother, more dynamic user experiences. Hence introducing this will be an added advantage for rich web application development.
-Connector element	5	Breaking the API down into smaller, well-defined classes and visualizing them clearly can make it much easier to use and understand.
-Endpoints class element	5	Analysis of the "endpoint class element" indicates its potential as a suitable model for capturing the common characteristics of a certain class of endpoints.

**Application-Model Diagram and Model Class Diagram**

Diagram and its elements	Usefulness	Justification
<u>Application-model diagram and Model class diagram</u>	5	Introducing an Application-Model Diagram which wrapped the class diagram by its high-level component element for rich web apps could improve communication between user interactions, UI updates, and underlying data and collaboration. This streamlines development, and leads to more effective and maintainable applications.
- Component element	5	This acts as a map, providing users with a comprehensive overview of the different classes and their connections, and enhancing their navigational abilities within the system.
-Class element	5	This element could simplify the implementation of the designed classes by providing developers with a clearer understanding of each class's functionalities and expected interactions.



**View-Process Sequence Diagram**

<b>Diagram and its elements</b>	<b>Usefulness</b>	<b>Justification</b>
<u>View-process sequence diagram</u>	5	I strongly believe sequence diagrams are powerful tools for visualizing and understanding system communication. They excel at both clarifying interactions between different components and digging deeper into the details of exchanged messages. This makes them invaluable for both representing and explaining complex system flows.
- Application element	5	The diagram identifies the "Application" as a fundamental component, and further subdividing it into segments seems beneficial for both simplifying and enriching our understanding.
-View element	5	By using the "View element," we can clearly visualize how different actors interact with the application, including the sequence of steps and exchanged messages involved.
-Controller element	5	Examining controllers find out the specific events driven by each view and how they interact with relevant services.
-Endpoints element	5	Endpoints act as communication gateways, offering critical information about the APIs used within the system. This detailed view ensures smoother implementation and a deeper understanding of system interactions.
-Model class object element	5	By examining model class objects, we gain insights into the communication sequence and the structure of the information expected in return.

**Evaluation of the RiWAs Design Methodology (RiWAsDM)**

<b>Quality attribute</b>	<b>Opinion</b>	<b>Justification</b>
Simplicity	5	Be able to explain
Comprehensiveness	5	Be able to explain
Learnability	5	Be able to explain
Readability/understandability	5	Be able to explain

## Appendices

Development support	4	It is difficult to evaluate this attribute only by looking at the given examples.
Integrability	4	It is difficult to evaluate this attribute only by looking at the given examples.

### Overall Feedback

<b>Overall rating on the RiWAsML/RiWAsDM</b>	4.5
<p>This methodology excels in clarity, organization, and usability thanks to its systematic approach, diverse models, and intuitive notation. However, it could benefit from more complex examples, broader development support (cloud, security), and expanded communication mechanisms for elements like the DC-Bus. While UML familiarity eases learning, additional guidance would be helpful.</p> <p>Overall, this new language is clear and can be understood easily. Newly introduced diagrams enhance the usability and applicability of UML-based diagrams to rich web app development.</p>	

## Glossary

---

<b>A</b>	[1st occurrence] .....	2
Adoptability	[definition] .....	40
[1st occurrence] .....	[detailed] .....	39
[context] .....	Design approach	
[general] .....	[1st occurrence] .....	29
<i>Agile Model-Driven Development (AMDD)</i>	[detailed] .....	32
[1st occurrence] .....	Distributed system	
[detailed] .....	[1st occurrence - detailed] .....	38
AppController element	[definition] .....	38
[definition] .....	DSML	
Application element	[1st occurrence] .....	2
[1st occurrence] .....	[detailed] .....	36
[definition] .....	<b>E</b>	
AppModel element	EndpointsCollection element	
[definition] .....	[definition] .....	92
Architectural Description Language (ADL)	Engineering approach	
[detailed] .....	[1st occurrence] .....	8
[review] .....	[detailed] .....	27
Architectural elements	<b>G</b>	
[detailed] .....	GUI element	
Architectural style/software architectural style	[definition] .....	87
[1st occurrence] .....	<b>H</b>	
[detailed] .....	High-level component	
[review] .....	[definition] .....	82
Architecture/software architecture	High-level Connector element	
[1st occurrence] .....	[definition] .....	83
[detailed] .....	High-level design	
<b>C</b>	[1st occurrence] .....	2
Container/section elements	[detailed] .....	30
[definition] .....	High-level Views element	
<b>D</b>	[definition] .....	84
Delta-Communication (DC)		



Glossary

<b>W</b>	[definition].....	38
Web-based application	[detailed].....	38
[1st occurrence] .....		2