

WESTERN SYDNEY UNIVERSITY



School of Computing, Engineering and Mathematics

Auto-generation of Rich Internet Applications from Visual Mock-ups

Christopher Vinod D'Souza

A dissertation submitted in fulfilment of the requirements for the degree of
Doctor of Philosophy.

February 2018

©Christopher Vinod D'Souza

Dedication

I dedicate this thesis to:

My wonderful mother;

Late Lily D'Souza

for your endless sacrifices,

My amazing and beloved wife;

Pakamart Lertlumpleepun for your loving heart and caring soul,

My precious daughter;

Florene for being the greatest bundle of my joy,

My brothers and sisters;

Eric, Steve, late Clare, Letitia, Patrick, Wilma and Prajual for moulding my life

Acknowledgements

I am humbled to be completing this long, arduous but incredibly rewarding journey. I would like to thank God for giving me the strength to persevere with it. This long journey would not have been possible without the patience, commitment and endurance of my wife Bee, supervisors, professors at WSU, supervisory panel and friends. I am greatly indebted to you all.

I am grateful to Prof. Athula Ginige for accepting me as your student and to my colleague Tamara for recommending me to you. Prof. Athula you taught me how to think, question and challenge existing ways of life during this journey. You also taught me how to do research in a friendly and collaborative spirit, without which I would not have been able to complete this thesis.

I wish to thank *Prof Vincenzo Deufemia* for the endless support during your sabbatical from University of Salerno, Italy. You taught me the virtues of visual modelling. You also taught me how to write research papers collaboratively. Thank you.

I am indebted to *Danny Liang*, for your regular encouragement and the talks about technologies available for implementation of the tool used in this thesis. In addition, your own thesis on Smart Business Objects was also a source of inspiration for my research.

I am especially thankful to WSU administrative and technical staff at Parramatta Campus. *Veena, Nabil, Susan, Cheryl, Ruby, and Guang* you have been very helpful. I was fortunate to have your support during the period of my candidature.

I am also grateful to my current and former colleagues, *Pam, Alanah, Chi-Pui, Jan, Rabia, Kim, Peng, Girija* and *Tamara* at Australian Catholic University for your constant support and encouragement during my study. Alanah, thank you for mentoring me during my early years at ACU. I am also thankful to *Kate* and *Mary* for your help when I was researching with you at the Australian Technology Park.

I would like to extend my appreciation towards my fellow researchers at the AIEMS lab. I am especially thankful to *Ashini, Marcus* and *Manish* for testing the application developed for

this thesis. In addition, *Marie* and *Shoba* thank you for sharing and listening to many of my research seminars. Your support and encouragement helped me overcome many tribulations.

I am also thankful to the other testers of my application. *Peng, Sajan, Govind, Sidney, Nasir* and *Saiful*. I am extremely grateful for the long hours you put for testing.

To my family friends in Sydney: *Sudhir, Grace, Bulbul, Vishwa, Gavin, Serena, Leonard, Ying, Sajan* and *Jeeva*, thank you all for treating me like one of your own. The hospitality you showered on me made it easier to ride this journey.

Finally, I'm eternally grateful for my family for their love, support, understanding, and many sacrifices. Without you this journey wouldn't have even begun.

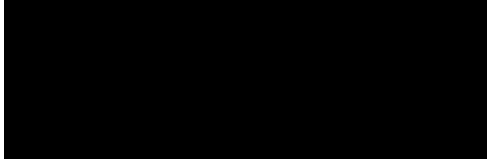
I'm forever grateful to my Mum, *Lily*, for inspiring me to do this PhD. By doing your B.A and B Ed after raising eight children, you taught me age is not a barrier in achieving success.

To my beloved wife, *Bee*, you never complained even once about the time I took to complete this PhD. Without your support, I would have given up. You also presented me with a beautiful daughter, *Florene*, half way during this journey. *Florene* made it a lot easier to balance work, study and family life together.

To my brothers and sisters, *Eric*, late *Clare, Steve, Letitia, Patrick, Wilma* and *Prajual*, I love you all. Being one of the youngest, thank you for nurturing me during my childhood days

Statement of Authentication

The work presented in this thesis is, to the best of my knowledge and belief, original except as acknowledged in the text. I hereby declare that I have not submitted this material, either in full or in part, for a degree at this or any other institution.

A large black rectangular redaction box covering the signature area.

(Christopher Vinod D'Souza)

Table of Contents

Contents

	LIST OF TABLES	vii
	LIST OF FIGURES AND ILLUSTRATIONS	ix
	ABBREVIATIONS	xi
	ABSTRACT	xiii
	LIST OF RELATED PUBLICATIONS	xv
1	INTRODUCTION	1
1.1	Overview on the relevance of this research	1
1.2	Impact of enterprise size on features of software projects	4
1.3	Business Analysts as developers of SME applications	7
2	MODELLING TRANSACTIONAL WEB APPLICATIONS	11
2.1	Traditional web application architecture	12
2.2	Rich Internet Application (RIA) and its features	14
2.2.1	AJAX architecture and communication structure	15
2.2.2	Effect of client-server architecture on the complexity of RIA development	17
2.3	Modelling of traditional web applications	19
2.4	RIA modelling methods	21
2.4.1	Traditional RIA modelling methods	22
2.4.2	RIA development using technological tools.....	22
2.4.3	Meta-modelling approach to RIA development	24
2.4.3.1	<i>Static view of the RIA UI meta-model</i>	25
2.4.3.2	<i>Dynamic view of RIA UI meta-model</i>	26
2.4.4	Model Driven Engineering (MDE) of RIAs	27
2.4.5	Visual mock-up approaches to software development	29
2.4.6	Summary on modelling of web applications and its implications for BAs	34
3	RESEARCH QUESTIONS AND RESEARCH DESIGN	36
3.1	Research question(s)	36
3.2	Research Design	40
3.2.1	Design Science Research in IS	41
3.2.2	Guidelines for conducting DSR in IS.....	44
3.2.3	Checklist for DSR in IS	45
3.3	Summary	46
4	MOCK-UP LANGUAGE SPECIFICATIONS	47

4.1	Essential features of SME web applications.....	47
4.2	Mock-up language features to express the requirements of SME applications	49
4.2.1	Mock-up segment for creating new entities	52
4.2.2	Mock-up segment for search management	56
4.2.3	Mock-up segment for insert business transactions.....	61
4.2.4	Mock-up segment for managing report generation	64
4.2.5	Mock-up segment for update operation	66
4.3	Meta-model of the mock-up language	71
4.4	Summary of the mock-up language specifications	79
4.5	Features of the tool for easy integration with the language.....	81
5	DESIGN OF AN AUTO-GENERATOR OF WEB APPS FROM VISUAL MOCK-UPS	83
5.1	Algorithms for database schema generation from mock-up	84
5.1.1	Identifying DFYWs	84
5.1.2	Identifying DFY Containers	85
5.1.3	Identifying Entity-Relationships (E-Rs) among database tables	86
5.1.3.1	<i>Identifying E-Rs from nested DFY Containers.....</i>	<i>87</i>
5.1.3.2	<i>Identifying E-Rs from Search Containers.....</i>	<i>88</i>
5.1.3.3	<i>Identifying E-Rs from “temporarily store for insert” annotated navigation widgets</i>	<i>89</i>
5.1.4	The generated E-R model of the example application	91
5.2	MVC-MC Generator for search operations	92
5.2.1	Component generation for a search operation	97
5.2.1.1	<i>Auto-generation of Client-Side Views for Search.....</i>	<i>99</i>
5.2.1.2	<i>Auto-generation of Server-Side Models for Search.....</i>	<i>104</i>
5.2.1.3	<i>Auto-generation of Client-Side Models for Search result.....</i>	<i>106</i>
5.2.1.4	<i>Auto-generation of Client-Side Controller for Search related operations</i>	<i>107</i>
5.2.1.5	<i>Auto-generation of Server-Side Controller for Search related operations</i>	<i>109</i>
5.3	MVC-MC Generator for Insert operations.....	110
5.4	MVC-MC Generator for Report generation.....	118
5.5	MVC-MC Generator for Update operations.....	121
5.6	MVC-MC Generator for Delete operations	125
5.7	Management of the auto generated application	125
5.8	MVC-MC components of the case study application	127
5.9	Trial evaluation of the design	130
6	VALIDATION.....	132
6.1	Usability validation concepts and the C-INCAMI framework	134
6.1.1	C-INCAMI framework for usability testing.....	135

6.1.1.1	<i>Defining the testing requirements</i>	136
6.1.1.2	<i>Designing the measurement metrics</i>	137
6.1.1.3	<i>Designing the evaluation indicators</i>	138
6.1.1.4	<i>Implementing the measurement</i>	138
6.1.1.5	<i>Analyse and report the evaluation</i>	139
6.2	Usability validation plan	139
6.2.1	Business Analysts as usability testers	140
6.2.2	Training usability testers	140
6.2.2.1	<i>Training to use the Balsamiq tool</i>	140
6.2.2.2	<i>Training to use the visual mock-up language with Balsamiq</i>	141
6.2.3	SME Application Case Studies for Usability Testing	142
6.3	Usability testing of the mock-up language.....	142
6.3.1	Defining the testing requirements of the mock-up language.....	143
6.3.2	Usability requirements tree of the mock-up language.....	144
6.3.2.1	<i>Effectiveness in use</i>	145
6.3.2.1.1	Sub-task correctness effectiveness	145
6.3.2.1.2	Sub-task completeness effectiveness	145
6.3.2.1.3	Task successfulness effectiveness	146
6.3.2.2	<i>Efficiency in use</i>	146
6.3.2.2.1	Sub-task correctness efficiency	146
6.3.2.2.2	Sub-task completeness efficiency	146
6.3.2.2.3	Task successfulness efficiency	146
6.3.2.3	<i>Satisfaction in use</i>	147
6.3.3	Usability testing tasks for validating the mock-up language	147
6.3.4	Designing the measurement metrics for the usability of the language.....	155
6.3.4.1	<i>Designing the measurement of sub-task correctness effectiveness</i>	156
6.3.4.2	<i>Designing the measurement of sub-task completeness effectiveness</i>	156
6.3.4.3	<i>Designing the measurement of task successfulness effectiveness</i>	157
6.3.4.4	<i>Designing the measurement of sub-task correctness efficiency</i>	158
6.3.4.5	<i>Designing the measurement of sub-task completeness efficiency</i>	159
6.3.4.6	<i>Designing the measurement of task successfulness efficiency</i>	159
6.3.4.7	<i>Designing the measurement metrics for satisfaction in use of the mock-up language</i>	160
6.3.5	Specifying acceptable threshold levels for evaluation indicators.....	162
6.3.6	Implementing the measurement - Mock-up Language Usability values	163
6.3.6.1	<i>Data collection for direct metrics</i>	164
6.3.6.2	<i>Computing indirect metrics</i>	166
6.3.7	Analysis and reporting of the evaluation of mock-up language	171

6.4	Usability testing of the auto-generated applications	174
6.4.1	Defining the testing requirements of the generated applications	175
6.4.2	Designing the measurement for the usability of the generated applications	176
6.4.3	Designing the usability evaluation indicators of the generated applications	177
6.4.4	Implementing usability testing measurement of the generated applications.....	177
6.4.5	Analysis and reporting of the evaluation of the generated applications.....	177
6.5	Testing functional correctness of the generated applications.....	178
6.6	Validation of the auto-generating tool as an integrated system.....	183
7	GENERAL DISCUSSIONS, LIMITATIONS, FUTURE DIRECTIONS AND CONCLUSIONS.....	184
7.1	General discussions based on DSR in IS checklist questions	185
7.2	Limitations and future directions of the research	195
7.2.1	Mock-up limitations and opportunities for future research.....	197
7.2.2	Database limitations and opportunities for future research	199
7.2.3	Technological limitations and opportunities for future research	200
7.2.4	Testing limitations and opportunities for future research	201
7.2.5	Adaptability limitations and opportunities for future research.....	201
7.3	Conclusions.....	203
8	REFERENCES.....	205
	GLOSSARY	215
APPENDIX 1	ALGORITHMS FOR GENERATING DATA MODEL FROM MOCK-UP	222
Appendix 1.1	Algorithm to identify a DFYW	222
Appendix 1.2	Algorithm to identify a DFY Container	222
Appendix 1.2.1	Algorithm to identify whether a container is a “unique” container	223
Appendix 1.2.2	Algorithm to store references to all inner DFY containers in each container	223
Appendix 1.2.3	Algorithm to store references to all DFY Containers	224
Appendix 1.3	Algorithm to find E-Rs from nested DFY Containers	224
Appendix 1.4	Algorithm to find E-Rs from Search Container.....	225
Appendix 1.4.1	Algorithm to find Search Containers.....	226
Appendix 1.4.2	Algorithm to find a DFYW’s Container name	226
Appendix 1.4.3	Algorithm to find E-Rs from Search Containers	226
Appendix 1.5	Algorithm to find E-Rs from “temporarily store for insert” annotations	227
Appendix 1.5.1	Algorithm to find all “Temporarily Store for Insert” annotated navigation widgets.....	228
Appendix 1.5.2	Algorithm to find all “Select for Insert” annotated navigation widgets.....	229
Appendix 1.5.3	Algorithm to find containers that are not targets of “select for insert” or “temporarily store for insert” annotated navigation widgets.....	230
Appendix 1.5.4	Algorithm to find target of “Select for Insert” annotated navigation widget starting from a Search Container	231

Appendix 1.5.5 Algorithm to get all DFY Container names referenced in a Data View Container	231
Appendix 1.5.6 Algorithm to find whether a Widget is a Data View Widget	232
Appendix 1.5.7 Algorithm to check whether a Container is a Data View Container	232
APPENDIX 2 ALGORITHMS FOR GENERATING COMPONENTS FOR SEARCH OPERATIONS	233
Appendix 2.1 Deriving Server-Side Model algorithms for search operation	233
Appendix 2.1.1 Helper function for deriving Server-Side Model algorithms for search operation	233
Appendix 2.2 Deriving Client-Side Model attributes for search result operations	234
Appendix 2.3 Deriving Client-Side Controller for search and search result	234
Appendix 2.3.1 Client-Side Controller helper function for search operation	235
Appendix 2.3.2 Client-Side Controller helper functions for search result traversals	236
Appendix 2.3.3 Client-Side Controller helper functions for on-load event operation	237
APPENDIX 3 ALGORITHMS FOR GENERATING COMPONENTS FOR INSERT OPERATIONS	238
Appendix 3.1 Algorithm to manage ‘select for insert’ action	238
Appendix 3.1.1 Algorithm for CSC to manage ‘select for insert’ action	238
Appendix 3.1.2 Defining CSM for storing data on a ‘select for insert’ action	239
Appendix 3.1.3 Initializing CSM with user selected data on a ‘select for insert’ action	239
Appendix 3.2 Algorithm to manage ‘temporarily store for insert’ and “commit insert” actions ...	240
Appendix 3.2.1 Defining Client-Side Controller for an insert operation	240
APPENDIX 4 ALGORITHM FOR GENERATING COMPONENTS FOR REPORT MANAGEMENT	242
Appendix 4.1 Algorithm for defining Client-Side Model for report generation	242
Appendix 4.2 Algorithm for defining a Client-Side View for report generation	242
Appendix 4.3 Algorithm for defining a Client-Side Controller for report generation	243
APPENDIX 5 ALGORITHM FOR GENERATING COMPONENTS FOR UPDATE OPERATIONS	245
Appendix 5.1 Algorithm for defining Client-Side Controller for update	245
Appendix 5.2 Algorithm for defining Client-Side View for update	245
Appendix 5.3 Algorithm for defining Client-Side Model for update	246
Appendix 5.4 Algorithm for Initializing Client-Side Model for update	246
APPENDIX 6 USABILITY TESTING DETAILS	248
Appendix 6.1 Biographical details of the usability testers	248
Appendix 6.2 Usability Testing Case Studies	249
Appendix 6.2.1 Testing Case study a- Question Answer System	249
Appendix 6.2.2 Testing Case study b- Student-Teacher Consultation System	249
Appendix 6.2.3 Testing Case study c- Patient-Dietician Consultation System	250
Appendix 6.2.4 Visual Modelling Tasks in the Question and Answer System	251
Appendix 6.2.5 Visual Modelling Tasks in the Teacher Consultation System	256
Appendix 6.2.6 Visual Modelling Tasks in the Patient-Dietician Consultation System	260

Appendix 6.2.7 End user tasks in the auto-generated Teacher Consultation System	264
Appendix 6.2.8 End user tasks in the auto-generated Question & Answer System	265
Appendix 6.2.9 End User Tasks in the Patient-Dietician Consultation System	266
INDEX	268

List of Tables

Table 1: Skills required for some mock-up tools.....	34
Table 2: Integrated view of the research questions	39
Table 3: Design Science Research guidelines adopted from Hevner, March and Ram(2004).....	45
Table 4: Checklist for DSR in IS	46
Table 5: Summary of definitions of container types.....	80
Table 6: Summary of annotations for behavioural specifications.....	80
Table 7: MVC-MC Components for the example case study	127
Table 8: DSR evaluation methods (Hevner, March & Ram 2004, p.86)	132
Table 9: Sub-tasks in Database Field Yielding Container mock-up specification task	149
Table 10: Sub-tasks in Search Container specification task	150
Table 11: Sub-tasks in Search Result Container specification task	151
Table 12: Sub-tasks in Data View Container specification task.....	152
Table 13: Sub-tasks in Navigation Only Container specification task	152
Table 14: Sub-tasks in Update Container specification task	153
Table 15: Sub-tasks in behavioural tasks' specifications.....	154
Table 16: Sub-tasks in behavioural tasks' specifications (continuation).....	155
Table 17: Threshold values for Quality in Use indicator levels	163
Table 18: Effectiveness in Use and Efficiency in Use values for DFYC mock-up task	168
Table 19: Effectiveness in Use and Efficiency in Use values for Search Container mock-up task	168
Table 20: Effectiveness in Use and Efficiency in Use values for Search Result Container mock-up task	169
Table 21: Effectiveness in Use and Efficiency in Use values for Data View Container mock-up task	169
Table 22: Effectiveness in Use and Efficiency in Use values for Update Container mock-up task.....	169
Table 23: Effectiveness in Use and Efficiency in Use values for Navigation Only Container mock-up task	169
Table 24: Effectiveness in Use values for behavioural tasks in the mock-up.....	170
Table 25: Efficiency in Use values for behavioural tasks in the mock-up	170
Table 26: Cross checking research question 1 with findings	194
Table 27: Cross checking research question 2 with findings	195
Table 28: Cross checking research question 3 with findings	195
Table A-1: Biographical Details of Testers	248
Table A-2: Actions in the "Creation of a Question entity" task.....	251
Table A-3: Actions in the "Creation of an Expert entity" task.....	251
Table A-4: Actions in the "Expert Login" task	253
Table A-5: Sub-tasks for the "searching questions and displaying them" task.....	253
Table A-6: Sub-task actions for searching a Question	253
Table A-7: Sub-task actions for managing results following the search of a Question	253
Table A-8: Sub-task actions for the display Question(s) within a Search Result Container	253
Table A-9: Tasks for answering or deleting a Question or for its assignment to an Expert	254
Table A-10: Actions for managing the task of "assignment of a Question to an Expert" or for "updating" or "deleting a Question"	254
Table A-11: Actions in the "update of a Question" task	255
Table A-12: Actions in the task of managing assignment of a displayed Question to an Expert	255
Table A-13: Actions in the task of selecting an Expert from Search Result Container for potential linkage with a previously selected Question	255
Table A-14: Actions for displaying a selected Expert for linkage with a previously selected Question	255
Table A-15: Actions for creating a navigation only container as a header	256
Table A-16: Actions for "Creation of a Student entity" task	256
Table A-17: Actions for the "Creation of a Question entity" task	256
Table A-18: Sub-tasks for the "searching, displaying and or deletion of Teacher Consultation" task	258

Table A-19: Actions for displaying and or deleting one or more Teacher Consultation entities	258
Table A-20: Tasks for linking a Teacher Consultation (TC) with a Student and the update of TC status	258
Table A-21: Actions in task to search for Student and or Teacher Consultation	258
Table A-22: Actions in task to select a data-set in in Search Result Container	259
Table A-23: Actions in task to display and link selected Student with Teacher Consultation entity	259
Table A-24: Actions in task to update status of Teacher Consultation entity	259
Table A-25: Actions in “Creation of a Patient entity” task	260
Table A-26: Actions in “Creation of a Dietician entity” task	260
Table A-27: Sub-tasks for storing further details of existing Patient during a consultation	261
Table A-28: Action for displaying a Patient entity in a Data View Container within Search Result Container	261
Table A-29: Actions in task to display a Patient and potentially add further details	261
Table A-30: Actions in task to insert further details such as Height or Weight of a patient during a consultation	261
Table A-31: Sub-tasks for assigning a Patient to a Dietician	262
Table A-32: Action in task to search Patient and Dietician	262
Table A-33: Actions in task to display and link selected Patient with selected Dietician	262
Table A-34: Actions in the “Creation of a Student entity” task	264
Table A-35: Actions in the “Creation of a Teacher Consultation entity” task	264
Table A-36: Actions in the task for “searching, displaying and or deletion of a Teacher Consultation entity” ...	264
Table A-37: Actions in task for linking a Teacher Consultation (TC) with a Student and the update of TC status	264
Table A-38: Actions in task for using a main navigation header in the Teacher Consultation System	265
Table A-39: Actions in task for “searching and displaying Question entities”	265
Table A-40: Actions in task for answering or deleting a Question or for its assignment to an Expert	265
Table A-41: Actions in task for using a main navigation header in the Question Answer System	266
Table A-42: Actions in task for storing further details of existing Patient during a consultation	266
Table A-43: Actions in task for assigning a Patient to a Dietician	266
Table A-44: Actions in task for using a main navigation header in the Patient Dietician System	267

List of Figures and Illustrations

Figure 1: Relevance of this research to SMEs	9
Figure 2: Web application components	11
Figure 3: MVC architecture in a traditional web application	13
Figure 4: AJAX Architecture (Scott, 2007)	16
Figure 5: RIA communication structure	16
Figure 6: MVC-MC architecture for RIA	18
Figure 7: Static View of the RIA UI meta-model	26
Figure 8: Dynamic view of the RIA UI meta-model	27
Figure 9: Auto-generating components desired to answer research question 2	39
Figure 10: Design Science Research cycles	41
Figure 11: Research plan based on DSR in IS	42
Figure 12: Mock-up for a Travel Deals web app	51
Figure 13: Use Cases in the Travel Deals web app	52
Figure 14: Mock-up segment for creation of Travel entity and Administrator entity in Travel Deal web app	53
Figure 15: Mock-up segment for managing search and search results in Travel Deal web app	58
Figure 16: Mock-up segment illustrating searching multiple entity types in a search container	60
Figure 17: Mock-up segment for insert transaction processing in Travel Deal web app	62
Figure 18: Expanded view of the mock-up segment for booking confirmation page in Travel Deal web app	66
Figure 19: Mock-up segment for managing update operation in Travel Deal web app	68
Figure 20: Mock-up to highlight update of multiple entity types in an update container	70
Figure 21: Meta-model of the mock-up language	72
Figure 22: Perceiving a UI as a group of containers	87
Figure 23: E-R Model due to nested DFY Containers	88
Figure 24: Customer-Travel relationship from criteria in Search Container	88
Figure 25: Payment-Customer-Travel entity relationships from "temporarily store for insert" annotations	89
Figure 26: Search Container with "Temporarily store for insert" annotations among containers	90
Figure 27: E-R Model of the Travel Deal case study	92
Figure 28: Sequence diagram showing interaction among web components for a search operation in Travel Deal web app	95
Figure 29: Snapshot of the auto-generated code for Client-Side View of a Search Container in Travel Deal web app	101
Figure 30: Snapshot of the auto generated code for Client-Side View of a Search Result Container in Travel Deal web app	104
Figure 31: An abstract Client-Side Model for search result	107
Figure 32: Snapshot of the auto-generated CSM code segment for storing user selected search results data-set	112
Figure 33: Sequence diagram highlighting component interaction in an insert business transaction	113
Figure 34: Snapshot of auto-generated CSV of Selected Order Details container in Travel Deal web app	114
Figure 35: Various forms of linked list of containers possible in an insert transaction	117
Figure 36: SequencedDiagram highlight component interaction for generation of a report	119
Figure 37: Snapshot of auto-generated code segment of the CSV for Displaying Selected Travel Deal in Travel Deals web app	120
Figure 38: Snapshot of auto-generated code segment of the CSV for Displaying Customer Details in Travel Deals web app	120
Figure 39: Snapshot of auto-generated code segment of the CSV for Displaying Payment Details in Travel Deals web app	120
Figure 40: Sequence diagram highlighting component interactions in an update operation	122
Figure 41: Segment of auto-generated code for CSCSearch illustrating selection of entities for update from CSMSearchResult in Travel Deal web app	123
Figure 42: Segment of auto-generated Client-Side View code for update in Travel Deal web app	123
Figure 43: Mock-up segment for update of multiple entity types in a single page	124
Figure 44: An instance of the auto-generated Travel Deals page before the search operation	130

Figure 45: An instance of the auto-generated Travel Deals page after the search operation	130
Figure 46: Usability problems found versus number of testers.....	135
Figure 47: System Usability Scale. Copyright Digital Equipment Corporation, 1986	161
Figure 48: Illustrating tabulation of data collection.....	165
Figure 49: Effectiveness in Use and Efficiency in Use for mock-up specifications of Containers.....	171
Figure 50: Effectiveness in Use and Efficiency in Use for mock-up specifications of behavioural operations.....	172
Figure 51:Plot of testers’ responses to SUS questions on mock-up language usage.....	174
Figure 52: Plot of testers’ response to SUS questions on usage of auto-generated applications.....	178
Figure 53: Screenshots of the auto-generated Question -Answer system	180
Figure 54: Screenshots of the auto-generated Patient-Dietician system	181
Figure 55: Screenshots of the auto-generated Teacher-Student Consultation system	182
Figure 56: Knowledge base of this research	189
Figure 57: Using potential “function” keyword in future versions	203
Figure A-1: Data structure to store first level nested DFY Containers in DFY Containers	225
Figure A-2: Use Cases in Question Answer System.....	249
Figure A-3: Use Cases in Teacher Consultation System	250
Figure A-4: Uses Cases in the Patient Dietician Consultation System.....	250
Figure A-5: A mock-up of the Question and Answer System	252
Figure A-6: A mock-up of the Teacher Student Consultation System.....	257
Figure A-7: A mock-up of the Patient Dietician System	263

Abbreviations

AJAX	Asynchronous JavaScript and XML
BA	Business Analyst
CIM	Computation Independent Model
C-INCAMI	Contextual-Information Need, Concept model, Attribute, Metric and Indicator
CRUD	Create Read Update Delete
CSC	Client-Side Controller
CSM	Client-Side Model
CSV	Client-Side View
CTT	Concur Task Tree
DFYC	Database Field Yielding Container
DFYW	Database Field Yielding Widget
DSR	Design Science Research
DSR in IS	Design Science Research in Information System
E-R	Entity - Relationship
ERP	Enterprise Resource Planning
GUI	Graphical User Interface
HTML	Hyper Text Mark-up Language
IS	Information System
ISO	International Standards Organization
IT	Information Technology
JSON	JavaScript Object Notation
M&E	Measurement and Evaluation
MDA	Model Driven Architecture
MDE	Model Driven Engineering
MVC	Model View Controller
MVC-MC	Model View Controller - Model Controller
MVC-MVC	Model View Controller - Model View Controller
OMG	Object Management Group
OODD	Object Oriented Design and Development
OOHDM	Object Oriented Hypertext Design Method
OOWS	Object Oriented Web Solution
OOWS for RIA	Object Oriented Web Solution for Rich Internet Application
ORM	Object Relational Mapping
PIM	Platform Independent Model
PSM	Platform Specific Model
RDBMS	Relational Database Management System
RIA	Rich Internet Application
RIA MVC	Rich Internet Application Model View Controller
SBOML	Smart Business Object Modelling Language
SME	Small to Medium Enterprise
SRS	Software Requirement Specification
SSC	Server-Side Controller

SSM	Server-Side Model
SUS	System Usability Scale
UI	User Interface
UML	Unified Modelling Language
UWE	UML-based Web Engineering
UWE-R	UML-based Web Engineering for RIA
WebML	Web Modelling Language
WYSIWYG	What You See Is What You Get
XHTML	Extensible Hyper Text Mark-up Language
XML	Extensible Mark-up Language

Abstract

Capturing and communicating software requirements accurately and quickly is a challenging activity. This needs expertise of people with unique skills. Traditionally this challenge has been compounded by assigning specialist roles for requirements gathering and analysis, design, and implementations. These multiple roles have resulted in information loss mainly due to miscommunication between requirement specialists, designers and implementers. Large enterprises have managed the information loss by using document centric approaches, leading to delays and cost escalations. But documentation centric and multiple role approaches are not suitable for Small to Medium Enterprises (SMEs) because they are vulnerable to market competitions. Moreover, SMEs require effective online applications to provide their service. Hence the motivation for carrying out this research is to explore the possibilities of empowering requirement specialists such as Business Analysts' (BAs) to take on additional responsibilities of designers and implementers to generate web applications. In addition, SME owners and BAs can communicate better if they perceive the application requirements using a What You See Is What You Get (WYSIWYG) approach. Hence, this research explores the design and development of mock-up-based auto-generating tool to develop SME applications.

A tool that auto-generates an application from a mock-up should have the capacity to extract the essential implementation details from the mock-up. Hence a visual mock-up language was created by extending existing research on meta-models of UIs for a class of popular modern web-based business applications called Rich Internet Applications (RIAs). The popularity of RIAs is due to their distinctive client-side processing power with desktop application like responsiveness and look and feel. The mock-ups drawn with the mock-up language should have sufficient level of details to auto-generate RIAs. To support this, the mock-up language includes constructs for specifying a RIA's mock-up in terms of layouts and the widgets within them. In addition, the language uses annotations on the mock-up to specify the behaviour of the system. In such an approach the only additional effort required of a Business

Analyst is to specify the requirements in terms of a mock-up of the expected interfaces of the SME application. Apart from the mock-up language, a tool was designed and developed to auto-generate the desired application from the mock-up. The tool is powered by algorithms to derive the database structure and the client-side and server-side components required for the auto-generated application. The validation of the mock-up language and auto-generating tool was performed by BAs to demonstrate its usability. The measurement and evaluation results indicate that the mock-up language and the auto-generator can be used successfully to help BAs in the development of SME application and thereby reduce delays, errors and cost overruns. The important contributions of this research are: (i) the design of a mock-up language that makes it easy to capture the structure and behaviour of SME web applications. (ii) algorithms for automatic derivation of the expected database schema from a visual mock-up. (iii) algorithms for automatic derivation of the client and server-side application logic. (iv) application of an existing measurement and evaluation process for the usability testing of the mock-up language and the auto-generated application.

This research followed the Design Science Research (DSR) method for Information System to guide the IS design and to capture the knowledge created during the design process. DSR is a research method useful in solving wicked problems requiring innovative solutions for incomplete, contradictory or changing requirements that are often difficult to recognize.

This research opens new ways of thinking about web application development for future research. Specifically, mock-ups with few easy to understand annotations can be used as powerful active artifacts to capture the structure and behaviour of applications not just of small but also large enterprises. Auto-generating tools can then create fully functional and usable applications holistically from such mock-ups, thereby reducing delays and cost overruns during software engineering.

List of related publications

(in descending chronological order)

- D'Souza, C., Deufemia, V., Ginige, A., & Polese, G. (2018). Enabling the Generation of Web Applications from Mockups. *SOFTWARE—PRACTICE AND EXPERIENCE*. <https://doi.org/10.1002/spe.2559>.
- Caruccio, L., Deufemia, V., D'Souza, C., Ginige, A., & Polese, G. (2015). A Tool Supporting End-User Development of Access Control in Web Applications. *International Journal of Software Engineering and Knowledge Engineering*, 25(02), 307–331. <https://doi.org/10.1142/S0218194015400112>
- Deufemia, V., D'Souza, C., & Ginige, A. (2013). Visually modelling data intensive web applications to assist end-user development (p. 17). ACM Press. <https://doi.org/10.1145/2493102.2493105>
- D'Souza, C., Ginige, A., & Liang, X. (2012). End-user friendly UI modelling language for creation and supporting evolution of RIA. In *Proceedings of the 7th International Conference on Software Paradigm Trend* (pp. 190–198). Rome, Italy: SciTePress - Science and Technology Publications.
- D'Souza, C., & Ginige, A. (2010). MVC-MC: A rich internet application architecture for optimal separation of concerns. In *Proceeding of the Int. Conf. Computer and Software Modeling, 2010* (pp. 78–82). Manilla.

1 INTRODUCTION

Software development approaches for SME application need to be different from large enterprise applications. This is because SMEs have a distinct set of challenges and opportunities than large enterprises. Modern approaches such as the agile processes too are constrained by many teams working in parallel on many small increments, thereby losing focus on the overall business requirements. So, a novel approach is explored where BAs are tasked with producing a holistic active requirement specification artifact in the form of visual mock-ups of the user interface which is used to auto-generate fully functional applications. An UI mock-up is a model of a software that looks like the real UI but is designed to gather information to build the real system. The aim is to explore the possibility of applying this approach for the development of SME application. Section 1.1 provides an overview on the relevance of this research. Section 1.2 discusses the general difference between large enterprises and SMEs and their impact on software projects to support them. Section 1.3 discusses the growing need for BAs to be able to manage both “business analysis” as well as “developmental” activities in SME projects.

1.1 Overview on the relevance of this research

This section provides an overview of the relevance of this research on exploration of new possibilities in managing software chaos in SME projects. It argues that while traditional software development approaches are suitable for large enterprises, modern approaches such as Agile are still not meeting the expectations of SMEs. Here “Agile” is an umbrella term for any iterative and incremental software development methodology that uses continuous planning, continuous testing, continuous integration and continuous feedback through the collaborative effort of self-organizing cross-functional teams. It highlights the need for a lighter software

development approach for SME applications using the services of a lean team of BAs to manage the requirements and the implementation of the system. Such an approach to software development may suit the innovative culture of SMEs. It concludes with a call to fill the gap in the support level currently available for BAs to fulfil dual roles of analysts and implementers.

Software is ubiquitously used in almost all aspects of our daily life. However consistent software engineering research from Standish Group, a reputed software research organization indicates that even after nearly 70 years of software development practise, about 50% of the projects are delivered are over-budget, behind schedule and with fewer than expected features (The Standish Group 2009; Lech 2013; Fernández & Penzenstadler 2015). Such projects are generally called challenged projects. Terms such as “software chaos” or “software crisis” are commonly used to refer to this malaise to highlight the need to change existing approaches to software development. There are many reasons for challenged projects of which poor communication among stakeholders during the software engineering process is a main concern(Walia & Carver 2009; Kraut & Streeter 1995; Curtis, Krasner & Iscoe 1988). The main stakeholders in a software project are the clients, analysts, designers, implementers, testers and users. The key role of an analyst is to understand and define the requirements, that of a designer is to specify platform independent solutions while the implementers role is to provide platform specific solutions. Poor communications among clients and analysts leads to errors in Software Requirement Specification (SRS) document. An SRS is a description of the functional and non-functional requirements of a software system to be developed, where functional requirements specify the behaviour and non-functional requirements specify certain criteria to judge the behaviour. Undetected requirement errors in the SRS can have a catastrophic effect on software project. Distinguished software engineering expert, Sommerville (2007) warns that each requirement error can be 100 times more costly to fix than an implementation error. Additionally, miscommunications can also occur between analysts, designers and implementers during the design and implementation phase of the development process due to misinterpretations of SRS and designs.

The traditional way of reducing miscommunication is to create documents following each activity in the development process. The documents are then used by the clients, analysts, designers and implementers to troubleshoot problems. For example, in the Waterfall model of software development, the documents are assumed to provide unequivocal descriptions of requirements and designs. However excessive emphasis on documentation delays the project. That is, a Waterfall model of software development is a document centric process of software development with distinct activity phases such as conception, initiation, analysis, design, construction, testing, deployment and maintenance. Hence the Waterfall model is suitable for those systems where the extra time required for the documentation is negligible compared to the flexibility offered by the application for long term use. Consequently, long lasting applications are generally the domain of large enterprises.

On the other hand, SMEs prefer modern agile methodologies that focus on quickly producing functional applications with minimal documentation. This is because SMEs have limited resources and constantly need innovative applications to remain competitive. However current agile approaches take several months rather than weeks to develop functional applications. Hence some researchers of SME projects claim that agile approaches are still not light enough to suit SME culture. One of the reasons for the slowness is the presence of several teams working in parallel to expedite the development and as Curtis et al (1988) observe, communication bottle necks and breakdowns are common wherever several software teams are involved in a project.

A common way to reduce miss-communication among teams is to reduce the number of teams participating in the development process of a SME application. Ideally a single team comprising of one or more BAs rather than other IS developers is preferable because BAs are considered to have the best requirement management skills among IS professionals and since requirement errors have the greatest impact on the cost and delivery times of the project (Sommerville 2007). Though BAs could be engaged in the development activity currently, hardly any help is available to them to take on the additional role of developers. A BA is defined as a person involved in the practise of enabling change in an organization by defining the needs and

requirements and recommending value based solutions to stakeholders (International Institute of Business Analysis 2017). The role of the analysts in this new situation is to create the requirement specifications as well as develop a functional application. However, BAs are generally considered to have weak development skills. Hence Model Driven Engineering (Kent 2002) principles could be used to auto-generate the application from conceptual requirement specification models. However very little support is currently available to BAs to carry on dual roles of analyses and development activities. The above discussion highlights the importance of empowering BAs to develop software without requiring the help of designers and implementers.

1.2 Impact of enterprise size on features of software projects

The size of an enterprise has an impact on the features expected of the supporting software. Longevity of business applications is the main requirement for large enterprises whereas flexibility to quickly develop applications with minimal cost is a top priority for SME applications. Enterprises are commonly classified as either Small to Medium (SME) or Large. SMEs generally have few employees though it could also be high as 250 employees (Turner, Ledwith & Kelly 2012). In this section, modern enterprise systems are compared with traditional enterprise system from a software project management perspective. It argues that large enterprises are not adversely affected by new business requirements whereas SMEs are vulnerable.

The popular adoption of expensive Enterprise Resource Planning (ERP) systems such as SAP, Oracle and PeopleSoft by large organization in modern times is a testament to the need for long lasting business applications to reduce the overall cost. ERP is a software system that integrates applications that allows an organization to manage the business and automate many back-office functions related to IS. In ERP systems, longevity is achieved by designing flexible systems to manage all the known current and assumed future requirements. Integration and configuration of several generic application modules is a common design feature of modern enterprise systems. Each module is a standard solution based on a series of assumptions to support most of the business activities across functions, across business units and across the world.

The generic nature of the application modules makes it flexible to manage currently assumed requirements. However if new requirements emerge, expensive add-on modules will be required to extend the life of the enterprise system (Davenport 1998).

On the other hand the traditional approach to building flexible systems for large enterprises is by using document centric process models such as Waterfall or Spiral software development methodologies for made to order systems(Royce 1970; Boehm 1988). Royce insists on written artifacts to force the developer to provide unambiguous descriptions of the requirements and the design before the implementation as a part of the Waterfall approach to software development for large enterprise systems. Thus, in the Waterfall model, at the end of the requirement gathering and analysis phase a requirement specification document is created and a design document is created following the design phase. The Spiral model is even more document centric than the Waterfall approach because it also uses a risk analysis model for each major requirement. Despite this, document driven approach is suitable for large enterprise applications since the requirements are not considered to change frequently and the development time is normally in terms of years. Any new requirements are managed by creating newer versions of analysis and design documentations before changing the implementation. From the above discussion, it is evident that no matter whether traditional or modern ERP systems are used, making changes for business new requirements is an expensive proposition for large enterprises. However, their impact on large enterprise is not adverse.

Managing new requirements is not a financial problem for large organizations because they enjoy some advantages over SMEs that enables them to overcome competition in market. Some of them include: the ability to exploit the market due to decades of monopoly, the ability to use mature knowledge management systems to explore future market avenues and the availability of vast resources both financially and materially to overcome ad-hoc loss(Antonelli & Scellato 2015; Bernroider & Koch 2001; Birley & Norburn 1985). That is large enterprises have various buffers to absorb temporary losses despite the lack of ability of enterprise systems to quickly adapt to newer requirements. In fact Sumner (1999) notes, one of

the critical success factors of enterprise wide information management system projects is to limit the add-ons for new requirements until a complete overhaul is required. The situation is however different for SME applications.

As mentioned at the beginning of this section, quickly building new applications with minimal cost is a top priority for SMEs. SMEs need to be flexible to new business requirements to survive the competition in the absence of reserve resources. The ability of businesses to quickly adapt to opportunities is considered to be an important critical success factor for business (*Exploiting the Software Advantage - Lessons from Digital Disrupters* 2015). Luckily SMEs have flat organizational structure and fewer departmental interfaces than large enterprises (Ghobadian & Gallear 1997). SMEs are generally located at a single site and normally respond rapidly to environmental changes with elevated level of innovativeness but with people focussed rather than documentation focussed project management practises. In addition most of the innovative ventures are largely self-financed (Perrini, Russo & Tencati 2007). Despite the lack of resources, SMEs spend over 40% of their resources on new projects during their life span (Turner, Ledwith & Kelly 2012). These factors indicate that software development approaches to SME applications should be treated differently from large enterprise applications. In fact the Project Management Institute has acknowledged the need to tailor their popular Project Management Body of Knowledge (PMBOK®) guide to SME projects (Turner, Ledwith & Kelly 2012). Understanding and helping SMEs build applications to support their objective is an important issue because in many countries SMEs account for over 99.9% of enterprises and contribute over 60 of the turnover while providing over 65% of the employment(Ghobadian & Gallear 1997; Turner, Ledwith & Kelly 2012).

In view of this, modern approaches using Agile Development Methodology are increasingly becoming popular among SMEs because of the lack of emphasis on documentation centric development (Abrahamsson et al. 2002). Agile software engineering methodology is an empirical process that recommends short iterations, continuous testing, self-organizing teams, and constant collaborations with frequent planning (Highsmith 2002). Agile processes are geared towards projects that have volatile requirements. Hence it is suitable for SMEs that need to constantly adapt to

new requirements, to remain competitive. However specialist SME project management researchers such as Turner et al. (2012, 2010) recommend further simplification to the existing Agile methods to suit SME application development. They suggest the core element of the development approach should consist of generalists (or non-technical specialists) working in small and fewer project teams to focus on requirement management to support the constantly evolving needs of their customers. Such a method should be simple to use and clearly show value to win the support of the entrepreneur. In other words, there is a growing call to employ requirement management specialists (that is BAs) to be involved in all phases of the development process.

1.3 Business Analysts as developers of SME applications

The discussion in the previous section reveals that current agile approaches are inadequate for SME projects because they do not focus enough on requirement management to quickly develop new applications. Since BAs have good requirement management skills among IS professionals, there is growing voice to utilize their services for design and development activities of applications. For example, in a seminal study of critical success factors of enterprise wide information management systems, Sumner (1999) observes that many project managers lamented on the lack of skills of BAs with both “business” and “technical” knowledge. Sumner expresses displeasure of the fact that though such a BA could replace 10 specialist programmers on an average, it is very hard to find them. However, to be fair to BAs, “programming” or “implementation” is traditionally not considered as an area of responsibility of BAs.

Paul, Cadle, and Yeates (2014) note that the main role of a Business Analyst (BA) is to investigate a business system where improvements are required. For successful accomplishment of this role, a BA should have holistic knowledge of the requirements of: IT systems, stakeholders, business processes and the organization. Fundamentally they should be able to analyse whether the IT systems are supporting the stakeholders, business processes and the organization. This indicates that one of the common responsibilities of a Business Analyst is System Analysis. System Analysis is

a subtask of a Business Analyst and focuses on analysing and specifying the IS system requirements for further software development. That is system analysis includes analysis of: system requirements, database storage requirements, system process requirements and user interface requirements. The focus in this thesis is to engage BAs in not only system analysis activity but also in development of applications, to reduce software chaos.

From the earlier discussions on Sumner's (1999) recommendation to engage BAs in the implementation activity and based on the call by Turner et al. (2010, 2012) to employ fewer professionals with generic skills to simplify the management of SME projects, it is evident that BAs are expected to extend the responsibilities beyond the traditional analysis activity. This is especially valid for SME projects since the complexity of the projects are low compared to large ones and so the project could be developed by a lean team of BAs. Hence in SME projects, software chaos due to miscommunication can be reduced by not involving separate professionals for analysis, design and implementation activities (Grigera et al. 2012). Moreover, requirements and design tasks are rarely treated as separate activities when performed by nonprogrammers (Ko et al. 2011).

However currently the resources available to BAs to take on additional roles of designers and implementers are not adequate. This may be attributed to general assumptions that BAs are not required to have good developmental skills. An Australian study of the education needs of BAs indicates that BAs exhibit large knowledge gaps in technical (developmental) skills though they possess good soft skills and business skills (Richards et al. 2011). This reiterates Sumner's(1999) earlier concern about the shortage of BAs with integrated "business" and "development" skills. This shortage may be due to lack of support for BAs in the development effort.

Hence this research aims to fill this gap of empowering BAs to develop SME applications. In addition, since SMEs commonly use transactional web applications to manage their business this research aims to help BAs in the development of a class of modern web applications called Rich Internet Applications (RIA) which are used for managing business transactions on the web. Here the term "business transaction"

refers to an interaction between a business and its client and may involve one or more tasks. A transactional web application is one which manages each business transaction in a series of one or more physical (computational) transactions, in which if one physical transaction fails, the entire process is considered to have failed. RIAs are popularly used by modern businesses because their user interface and response times are analogous to desktop applications, which make them user friendly. In this thesis unless otherwise stated, “web applications” and “RIAs” are used synonymously.

Figure 1 summarizes the relevance of this research to SMEs (organizations) application development, in terms of features of the organization, developer skills and expectations of the technology. It highlights that SMEs can use the services of BAs to quickly build transactional RIAs to meet the strategic objective of quickly responding to new business requirements to be competitive.

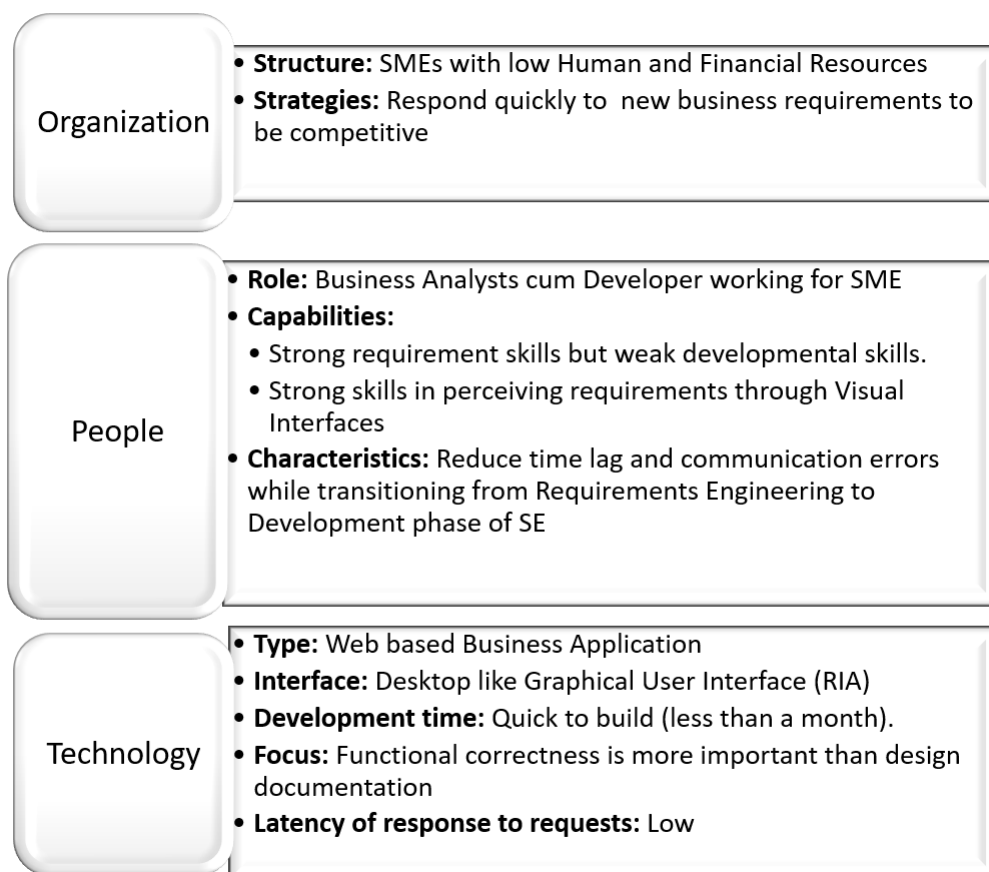


Figure 1: Relevance of this research to SMEs

The next chapter provides background information about the terminology, architectural concepts and modelling approaches used during analysis, design and development of transactional web applications. The emphasis is on finding the impact of architecture and the modelling methods on the ease of development of web applications. The ease of development is an important consideration since BAs are expected to participate in the development activity of SME applications and “ease of use” is an important consideration in adoption of this recommendation.

The overview of the thesis structure is as follows: Chapter 2 provides a background on modelling of transactional web applications to know the various developmental approaches used, both traditionally and currently. This yields the gap in the current research, leading to the research questions, which is discussed in Chapter 3 along with the research design method used to guide the research. Chapters 4 and 5 are the core chapters of this thesis since they deal with the design of a mock-up language and the design of an auto-generating tool respectively, to help BAs in the development of SME applications. Chapter 6 discusses how the usability of the mock-up language and the auto-generating tool is validated. Chapter 7 provides general discussions about the research along with its limitations, future directions and conclusions. Chapter 8 deals with references and the rest of the sections are appendices. There are six appendices. Appendix 1 to Appendix 5 provide finer details of the algorithms for the design of the auto-generator discussed in Chapter 5. Specifically, Appendix 1 provides contains algorithms for the derivation of the database model from the mock-up. Appendix 2 discusses the algorithmic details for deriving the logic for a search operation. Similarly, Appendix 3 discusses the algorithms from an insert operation perspective while Appendix 4 and 5 discuss the algorithms for report generation and update respectively. Finally, Appendix 6 contains the final details of the validations performed in Chapter 7.

2 MODELLING TRANSACTIONAL WEB APPLICATIONS

Transactional web applications are one among several types of web applications. Some examples of this type of web applications include online booking for travel, online banking and online shopping. Most SMEs fall under this category for managing their business processes, though they may also use collaborative and social web applications such as Facebook to provide extra customer support. Transactional web applications offer online interactivity and database support for transactional processing of business processes. “Transaction” in transactional web applications implies each business transaction is performed as a series of physical transactions. For example, a *manage sale order* business process may include physical transactions such as verify order, process payment and commit order to database. The typical components of traditional transactional web applications are shown in Figure 2.



Figure 2: Web application components

The web browser is a HTML based client which requests the services to a web server using internet protocols such as HTTP. The web server receives the request and forwards it to the web application server. Client-server communication is said to occur when a client-side component sends a request to a server-side component (or vice versa) to service a business request. The web application server contains the business logic, front end logic as well as the database logic of the application. It uses the services of a database server to create, search, update or delete operations on data in a database (Chen & Heath 2005). Most of the database servers for business applications are managed by Relational Database Management Systems (RDBMS). RDBMS is a database management system that is based on set theory. When the application server is ready with the solution to the request, it sends the information to the browser in a HTML page. That is, in a traditional web application all the

processing is done on the server side and the client (web browser) is used only to send or receive information from and to the user.

Since BAs are encouraged to develop SME web applications, some of the factors that affect the development of web applications are discussed in this chapter. Two key factors that have an impact on ease of development are the complexity of the architectural design and the modelling methods used. Obviously, it is easier to support BAs in the development process if the web applications follow a simple architectural design and the modelling method does not include cognitively challenging activities. Hence it is imperative to know what architecture and modelling methods are currently used. In the following sub-sections, Section 2.1 discusses a traditional web application architecture that is popularly used to reduce development time and Section 2.2 discusses the same with respect to RIAs. Sections 2.3 and 2.4 discuss the various modelling methods used for traditional web application and RIA development respectively.

2.1 Traditional web application architecture

Web applications should be simple to design yet be flexible to accommodate new business requirements and user-friendliness. Simple designs necessitate simple software application architectures in managing the escalating cost of developing systems for new business requirements. Here the term “architecture” is used to refer to the conceptual structure and logical organization of a computer-based system. Often new business requirements arise due to the changing nature of the business itself as well as due to the need to support customers using diverse types of interactive devices. User friendliness refers to the look and feel of the UIs as well as with respect to the responsiveness of the system.

To provide flexible designs Reenskaug (1979) introduced a new architecture that decouples the business logic components from other components such as UI components, while working on Graphical User Interface systems at Xerox Corporation. Reenskaug proposed a solution in the form of Model View Controller (MVC) architectural pattern where a pattern is a software engineering term used for

a general solution to a recurring problem. This pattern is popularly followed by most transactional web applications. A recurring architectural problem in many web applications is the challenge of decoupling the sub-system for user interface components (also known as View) from the sub-system for controlling each business process (also known as Controller) and the sub-system for the business logic component (as known as Model). The decoupling is desirable because it enables independent evolution of either the Model or the View or both with minimal coupling with one and another.

The Model in MVC represents the component that deals with the business entities. The Model component does not contain any logic regarding user interface or other components. Similarly, the View represents the logic that deal only with the generation of the front-end logic (i.e. browser side code). And the Controller acts like a manager for a user request. The Controller contains the sequence of steps required to service a business process but seeks the help of the Model to perform each request in the business process and when the desired information is received from the Model it directs the View to generate a dynamic HTML web page with the desired data embedded in the page. The generated page is then sent to the browser. This is shown in Figure 3.

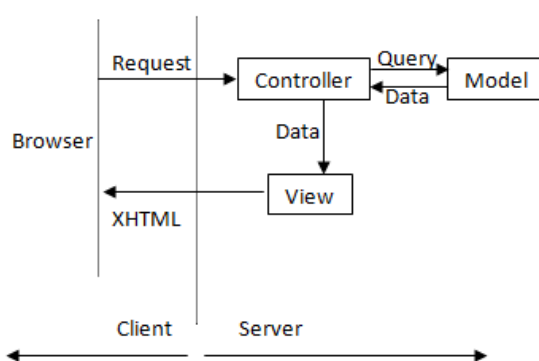


Figure 3: MVC architecture in a traditional web application

From the figure, it is evident that in traditional web applications the Model, View and Controller all lie on the server side. That is, the client side does no processing activity and any user request requires a round-trip to the server over the internet. This time lag is typically more than tens of seconds. Hence though the MVC architecture is a

simple design of flexible sub-systems, it causes poor response times in web applications and poor response times often drives customers to competitor sites (Chaffey 2011). In addition, since the View is created on the server side in the form of HTML code, the look and feel of the generated UI is poor compared to desktop applications.

The above discussion indicates that traditional web applications using the MVC architecture is not suitable for SMEs customers. However this changed in 2003-2004 following the introduction of web 2.0 where web applications started having more client side processing resulting in better responsiveness and UI features than traditional web applications (Cormode & Krishnamurthy 2008). Traditional web applications are now referred as web 1.0 applications and web 2.0 applications are called Rich Internet Applications (RIA). In a web application context, client-side processing refers to processing on the browser using technologies such as JavaScript. The following section studies the features of several types of RIA architectures and how their architectures impact the ease of RIA development.

2.2 Rich Internet Application (RIA) and its features

RIAs are popularly used by modern business applications for their user-friendly features such as desktop like UIs and good response times (Busch & Koch 2009). Like all web 2.0 applications, RIAs have client-side components running on a web browser and server-side components. In Web 1.0 applications the client-side is mostly powered by HTML code whose principal function is to present the UI for user interaction. Server-side components consist mostly of the business logic for servicing the requests from the client-side. In addition, the server-side also manages data in a database. The time taken to service a request is called latency of response. The average latency of response of RIAs is lower than the traditional web applications. This is because in RIAs the client-side is powered by a client-side processing engine in addition to the HTML code. The presence of the client-side processing engine enables RIAs to create a rich set of UIs with low average latency of responses. Asynchronous communication is often used to reduce the latency of response. In RIA context, asynchronous communication means, client-server communication can take place at

the same time as the user interacts with a web page. In addition, RIAs, can asynchronously refresh or update parts of the client page rather than whole page refresh thereby further reducing the latency of response. Popular examples of RIAs are Facebook, Google and E-bay. This section discusses the various RIA architectures found in literature, with an emphasis on the complexity of designs for development. Asynchronous JavaScript and XML (AJAX) is one of the earliest forms of RIA architecture. The AJAX architecture is discussed in the sub-section 2.2.1. Sub-section 2.2.2 discusses the effect of client-server architecture on design complexity.

2.2.1 AJAX architecture and communication structure

AJAX is a RIA that uses JavaScript as a client-side processing language in the browser and has capabilities to make asynchronous calls to the server to refresh parts of a web page without whole page refresh. XML is the format in which data is formatted while communicating between the client and server. XML is a popular textual language for structuring data for communications on the web. Figure 4 illustrates the working of AJAX architecture. The client-side JavaScript (known as AJAX engine) acts as a combined Client-side Controller and View. A Client-side Controller is a client-side control logic unit to manage Client-side View layer and Client-side Model layer. A Client-side View is the logical unit that manages presentation of data and UI on the client-side and a Client-side Model is a logical unit for managing business entities on the client-side. The AJAX engine results in better UIs and client-server responses than traditional web applications. Better response times are obtained because web pages can quickly be created by client-side Views there by reducing the number of client-server requests

However, the AJAX engine is not available to the client during the first request to the server. When the first request is sent, a Server-side Controller loads the AJAX engine as well as the main page of the View, where the Server-side Controller is a part of the application logic to control the operations of the program on the server-side. Figure 5 illustrates the RIA communication structure. In Figure 5 the first request to the server is numbered 1. Following requests may either be handled independently on the client-side or jointly by client and server sides. Request 2 indicates a request

which can be handled by the client alone. However, if the request cannot be handled locally on the client-side, due to the need for server-side data, the AJAX engine will send the request to the Server-side Controller. This is illustrated as request 3.

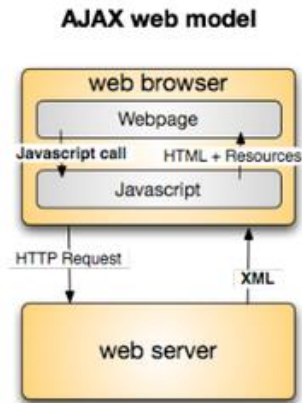


Figure 4: AJAX Architecture (Scott 2007)

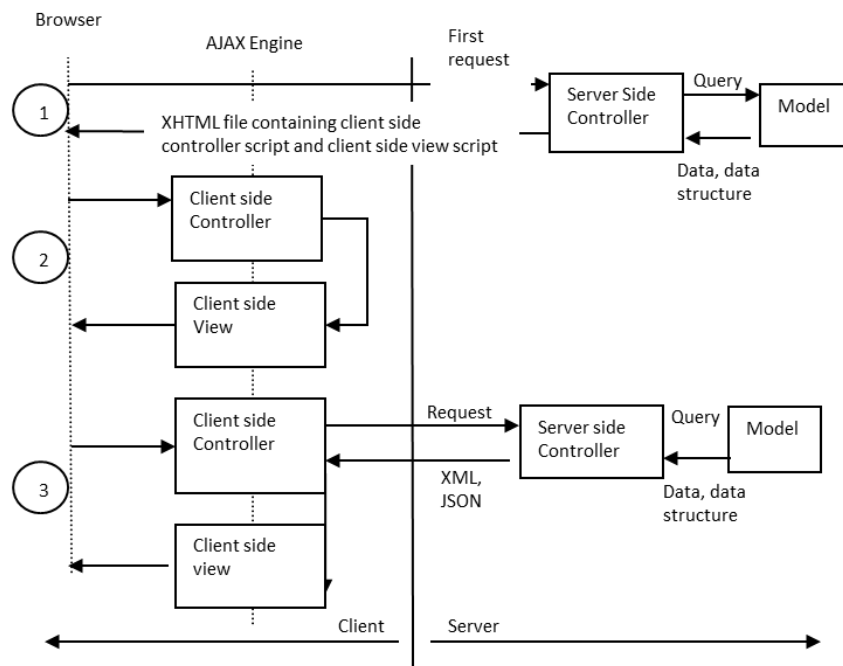


Figure 5: RIA communication structure

Though Figure 4 does not contain the details of the server-side architecture, most RIAs are powered by variants of MVC architectures. The variations are mostly made to reduce the complexity of designs and yet to provide desktop application like responsiveness. In other words, an optimal architecture can be considered as one which does not have a complex design and yet have good speed of response and good

UI look and feel. The next section discusses some of the prevailing RIA architectures found in the literature to find their pros and cons with respect to the complexity of the design as well as responsiveness to user interactions.

2.2.2 Effect of client-server architecture on the complexity of RIA development

Most of the RIA architectures have a full or a partial version of MVC on the client side and on the server side. In this section, some RIA architectures are discussed with a view on the latency of response and the complexity of the architecture. The architectures considered are: AJAX (Noda & Helwig 2005), MVC-MVC (Bozzon et al. 2006a, 2006b; Fraternali et al. 2010), RIA MVC (Morales-Chaparro et al. 2007) and MVC-MC (D'Souza & Ginige 2010).

The technical details of the AJAX architecture have already been discussed in Section 2.2.1. The AJAX architecture results in lower latency of response both for the initial request and subsequent requests from the server. The drawback of AJAX is that it does not decouple the Model, View and Controller in the AJAX engine. Though this makes it simple to design, the tight coupling makes it difficult to manage changes to the Model or View when the business requirements change. This problem has been overcome in the MVC-MVC architecture (Bozzon et al. 2006a, 2006b; Fraternali et al. 2010).

The MVC-MVC architecture uses a MVC framework in the browser and a MVC framework in the server. The duality of the MVC structures can reduce the latency of response. This is because the client-side MVC can first attempt to process the request locally instead of sending it to the server. Further a page generated by a View can be structured as an aggregation of sub-pages, rather than as a monolithic page. Even if a request cannot be serviced on the client-side, the size of the requested data will be low due to the small size of the sub-page to be generated, resulting in faster response times. However, the distributed nature of MVCs structures makes MVC-MVC design complex since a web page can potentially be created partly on the client-side and partly on the server-side. The complexity increases because changes are required to

be made to the structural and behavioural models of both client and server-side MVCs. To reduce the complexity of MVC-MVC architecture, the RIA MVC architecture was proposed (Morales-Chaparro et al. 2007).

The RIA MVC architecture reduces the design complexity of MVC-MVC by moving the MVC framework completely to the client-side. Consequently, the application will be easy to develop. However, this architecture will result in high initial load times and potential security breaches due to the absence of server-side Controllers. Server-side controllers are important to ensure only authentic users with authorized access are served. Hence though the architecture is simpler than MVC-MVC RIA, this architecture may not be secure to use. The limitations of MVC-MVC and RIA MVC have been overcome in the MVC-MC architecture (D'Souza & Ginige 2010).

The MVC-MC architecture has client-side MVC component and a server-side MC component and the Views are only generated on the client side. Figure 6 illustrates the MVC-MC architecture for RIA. Here, the initial request is handled by the server-side Controller and all subsequent requests are handled by appropriate client-side Controllers. Moreover, the View logic is only on the client-side.

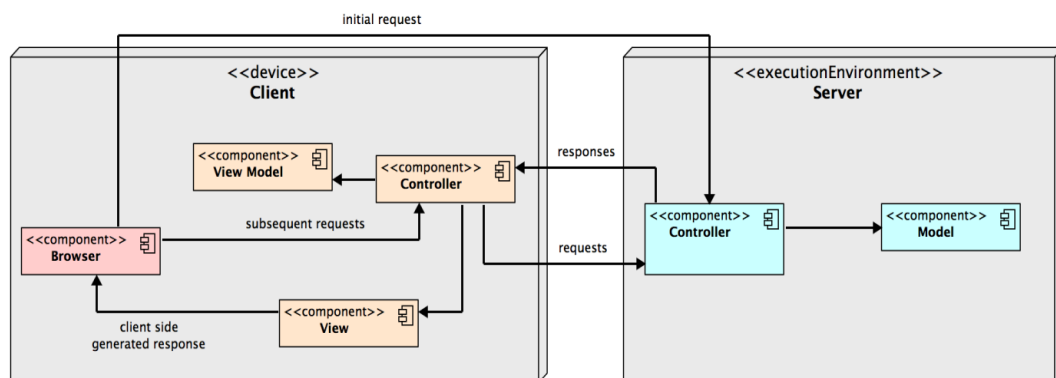


Figure 6: MVC-MC architecture for RIA

This reduces the complexity of the design in the architecture compared to the MVC-MVC architecture. Moreover, it retains the Server-Side Controller for security purposes. This results in an optimized architecture that is nearly as simple as AJAX, yet the system is easy to evolve due to the presence of decoupled components in the client side. In addition, the average latency of response is lower than the other

architectures since Views are only created on the client side with only data elements being transferred from the server to the client.

The above discussion indicates that the various client-server architectures have certain advantages and disadvantages. A RIA with low design complexity and response times can be considered to have an optimal architecture for SME applications. Such an optimal architecture is expected to have following features: not too high initial load time; fast response rates (low latency) for subsequent requests; decouple the Model, View and Controller but avoid unnecessary distribution of the MVC layers between the client and the server to reduce the design complexity.

The next section discusses the various modelling methods used in traditional and RIAs and the corresponding skills required for the modelling. The section highlights that most existing web application modelling methods require strong object-oriented development skills and hence are not suitable for BAs who to take on additional developmental responsibilities.

2.3 Modelling of traditional web applications

Several approaches are suggested in the literature for the modelling of traditional web applications during the development process. Almost all of them are adaptations of the Object-Oriented Design and Development (OODD) method to manage web application development. The adaptation is mainly to make up for the lack of emphasis on modelling of the presentation and navigational concepts in OODD. In the OODD method a system is designed as components in the form of interacting business objects where an object is an instance of a class and a class is a blueprint that encapsulates the behaviour and properties of a business object. The encapsulation restricts the scope of the behaviour to the properties of the class. Such an approach makes it less time consuming to manage during the development and evolution of the software (Jacobson, Booch & Rumbaugh 1999). The principal objective of OODD approach is to help designers and implementers think in terms of real world business objects. OODD is a popular software engineering approach because it helps designers and implementers in reducing errors, consequently

reducing software chaos during the implementation and design phase of the development.

Some of the popular traditional web modelling approaches are: Object Oriented Hypertext Design Method (OOHDM) (Schwabe, Rossi & Barbosa 1996), Web Modelling Language (WebML)(Ceri, Fraternali & Bongio 2000), UML-based Web Engineering Approach (UWE)(Koch & Wirsing 2001; Koch et al. 2008) and Object Oriented Web Solution (OOWS)(Pastor, Fons & Pelechano 2003). UML is a general purpose graphical modelling language predominantly used in the field of software engineering for visualizing the requirements and design of a system. The above-mentioned web modelling methods principally differ from each other in the way web navigation and presentation models are represented but at their core follow the OODD method (except for WebML). However, many developers find it hard to master the concepts of Object Oriented principles, giving rise to poorly designed object-oriented systems. Some reputed object technologists from IBM lament: “[when] object technology (OT) projects fail or face serious problems, a common thread is not having people who are really skilled in thinking in objects, object design, design patterns, and object-oriented programming” (Larman, Kruchten & Bittner 2001, p.8). The challenge of the OODD method is the steep learning curve required to master many abstract concepts. This leads to cognitive overload and consequently poor design solutions (Sweller 1988). Hence web modelling approaches that use OODD method are not suitable for BAs whose area of expertise is requirements gathering, analysis and management rather than providing design solutions.

WebML on the other hand employs a conceptual level language for high-level design of web applications. It provides a graphical notation to model the web site in terms of user's site views. Each user's site view is modelled of a set of pages and each page is composed of abstract content and navigational elements. Further it uses WebRatio¹ as a tool to auto-generate some aspects of the application from the conceptual model. A conceptual model is a representation of a system in terms of easy to understand concepts and is frequently used to gather and confirm

¹ <https://www.webratio.com/site/content/en/home>

requirements. The use of graphical concepts in the conceptual model makes it amenable to use by BAs. In addition, since it can also auto-generate the application from the WebML conceptual model this modelling approach is potentially suitable for BAs for development activities. However the main disadvantage of this approach is the model complexity increases greatly if a fully functional application is to be created even for fairly simple systems (Shakuntla, Sharma & Sarangdevot 2013). In addition, it also assumes that the user has existing knowledge of objects and their properties, though it does not directly use the ODD method. Having provided an overview of some of the approaches for traditional web modelling, the next section considers the same from a RIA perspective.

2.4 RIA modelling methods

RIAs are modelled using various methods. The methods vary in the level of effort required to design the system. For example, the traditional RIA modelling method discussed in Section 2.4.1 is aimed at designers and implementers and hence requires high-level development skills. On the other hand, methods that employ technological tools are aimed for people with lower development skills. Some technological tools for RIA development are discussed in Section 2.4.2. The design of technological tools for RIA development themselves require more abstract modelling approaches in the form of meta-models. The meta-modelling approaches are discussed in Section 2.4.3 and Section 2.4.4 discusses the principles of Model Driven Engineering which is useful for auto-generation of application. Finally, Section 2.4.5 discusses visual mock-up approaches to web application development. Visual mock-ups are increasingly becoming popular because they are easily perceived by clients' due to their implicit What You See Is What You Get (WYSIWYG) approach. In WYSIWYG approach stakeholders get to see what the end-result will look like while the interface or document is modelled. In addition, they are quick to build and greatly help in reducing communication errors with the clients.

2.4.1 Traditional RIA modelling methods

Several RIA modelling methods are prescribed in the literature with most of them being extensions of the traditional web application modelling methods discussed in Section 2.3. The extensions are provided to support client-side modelling of data, processes, navigation and communication alongside the traditional server-side components. They also support decoupling the content from the navigation and the presentation structures in both the client side as well as the server-side components. Some of the popular RIA modelling methods are OOWS for RIA (Valverde & Pastor 2009; Valverde et al. 2009), OOH4RIA (Garrigós, Meliá & Casteleyn 2009; Meliá et al. 2010), UWE-R (Busch & Koch 2009), WebML Extension for RIA (Bozzon et al. 2006a, 2006b) and OOHDM Extension for RIA (Urbietta et al. 2007).

The main difference among the RIA methods is with respect to the level of details in design, level of abstraction of web functionality and level of abstraction for navigation modelling. Since these methods follow the same basic principles of their corresponding traditional web application methods, further details are not discussed here. One common observation about the RIA methods is that they are intended for designers or programmers so offer very little support for BAs and nonprogrammers. However modern methods are increasingly using technological tools which are proving to be beneficial to BAs. Some of these tools are discussed in the next section.

2.4.2 RIA development using technological tools

Various technological tools requiring lower cognitive skills than OODD method are also used for the design and development of RIAs. Two popular ones are Microsoft's Silverlight (*Microsoft Silverlight* n.d.) and JavaFX based on Java technology (*JavaFX Developer Home* n.d.). They contain tool kits from which skeletal code of RIAs can be auto-generated. Designers can use the tool kits to define the presentation, navigation and the data models. This increases the productivity of designers. However, both these tools do not auto-generate a fully functional RIA. That is

developers are expected to have multiple IS skills to design the various models and manually integrate them together to generate a fully functional RIA.

GeneXus is another tool that comes close to helping BAs without entailing programming skills to auto-generate RIAs (Gonda & Jodal 2007; *GeneXus Overview* n.d.). GeneXus users first specify the desired *domain* (business) *entities* and then select a desired front-end *view* from a repository to present the domain entity. The *domain entities* are also used to auto generate the database schema of the application. A database schema is the organization of data as a blueprint of how the database is constructed. Several types of *views* are used. Examples of *views* include, form views to update a database, report views to display data retrieved from a database and procedure views for batch processing on a database. Database tables are then auto-generated from the user defined data specifications. GeneXus also permits the user or a BA to modify the default view and correspondingly updates the database tables. However, GeneXus's major disadvantage is that BAs need to have excellent idea of the data requirements and relationship among data, because the initial view is created from these requirements. This may be a handicap since BAs may not have the expertise to specify the data structures required for the domain entities. Furthermore, the front ends are auto-generated from system templates and hence may not satisfy any unique requirements of the client. Moreover, it is observed that GUIs takes about 45% of the design time and 50% of the implementation time(Myers & Rosson 1992). Hence businesses may prefer not to use such a tool since they do not have control over the UI design.

Similarly another tool that is used in several Western Sydney University research projects employs Component Based E Application Deployment Shell (CBEADS) to auto-generate web applications (Ginige 2003). CBEADS employs application components to auto-generate the database and the user's front-end views of the system. The application components contain the business entities of the system that can be specified by a BA at a conceptual level as inter-related smart business objects using a Smart Business Object Modelling Language (SBOML) (Liang, Marmaridis & Ginige 2007). SBOML uses succinct, non-programming, pseudo-English sentences to model business objects and the relations among them. For example, a SBOML

statement such as *“in organization, employee has first name, last name might have many office (has room number, building id)”* is easily understood by end-users or BAs. In the above example relationship between two business objects, namely employee and office are specified via SBOML. An advanced version of this tool currently produces RIA. The advantage of this approach is that a BA can quickly develop a functional web application with no programming skills required. However, the limitation is that the BA should perceive the system in terms of interacting business objects using SBOML, which is a cognitively challenging task. Moreover, BAs may not fully comprehend the technical consequences of a relating a business object wrongfully with other business objects. Another limitation is that the BA has no control over the look and feel of the auto-generated front-end views. This may particularly be a challenge because often the requirements related to interface and interaction issues are paramount in Web Applications (Rivero, Rossi, et al. 2011; Rivero et al. 2010).

The above discussion indicates that most of the existing tools either require the developers to have high-level cognitive skills to specify the business entities from which the database structure and the application logic can be identified, or they need to explicitly specify the database structure before the tool auto generates other components of the system. However, this may be a challenging activity for BAs. Hence tools utilising meta-models is sometimes recommended for faster development of web applications and RIAs while reducing the cognitive load on the developers. A meta-model is a model derived from a category of models by deriving common patterns in them. Some of the meta-modelling approaches are discussed in the next section.

2.4.3 Meta-modelling approach to RIA development

Many web application development tools use meta-models to build web application frameworks. Ginige (2010) observed that once a framework is created for the meta-model it provides a way to rapidly develop and deploy enterprise applications. Liang, Marmaridis and Ginige (2007) have successfully shown that tools using meta-models can be effectively used for supporting web application development by BAs.

Meta-models may be used either at component level or at a systems level. For example, a meta-model of UIs of RIAs may be used to deliver simple abstractions of the UI components and behaviours. Recently Valverde and Pastor (2009) proposed a meta-model for the UIs of RIA. They abstract UIs as a combination of static views and dynamic views. The static view abstracts fundamental UI element types among the multitude of UI elements in a web application while the dynamic view abstracts the fundamental behavioural changes to the UI due to user interaction. Here the meta-models reduce the complexity of modelling the UIs due to the reduction in the types of widgets considered in the design. Additional details of this approach are covered in the following two sub-sections.

2.4.3.1 Static view of the RIA UI meta-model

The meta-model of RIA UI is defined as a composition of widgets. A widget is a visual component of the UI. Its main responsibility is to provide data and interaction with the user. A widget is abstracted as an entity with a set of properties. For example, every widget has properties 'visible', to show/hide the widget and 'enable', to enable/disable user interaction. Figure 7 below represents the structural view of the meta-model of the RIA UI in the form of a class diagram. A class diagram is a Unified Modelling Language (UML) construct identifying the structural relationships among important concepts (also known as classes) under consideration in a system. More details on classes and class diagrams are available in Unified Modelling Language (UML) Resource Page (1997). Five types of widgets are identified at a meta-level:

Dataview Widgets (WDataView): to display structured data.

Input Widgets (WInput): allow input of data from the user.

Navigation Widgets (WNavigation): capture the target from which the UI is to be perceived.

Service Widgets (WService): allow the user to initiate the execution of a service from the domain logic.

Container Widgets (WLayout): organize and contain other widgets. A primary difference between a container widget and the other widgets is that a container widget does not generate events

Figure 7 illustrates that at a meta-level, a RIA UI may be modelled as a set of five basic widgets defined above and most of the widgets may be associated with some user or system generated event. The meta-model reduces the number of UI components to five, thereby reducing the complexity of modelling the vast array of UI elements in RIAs.

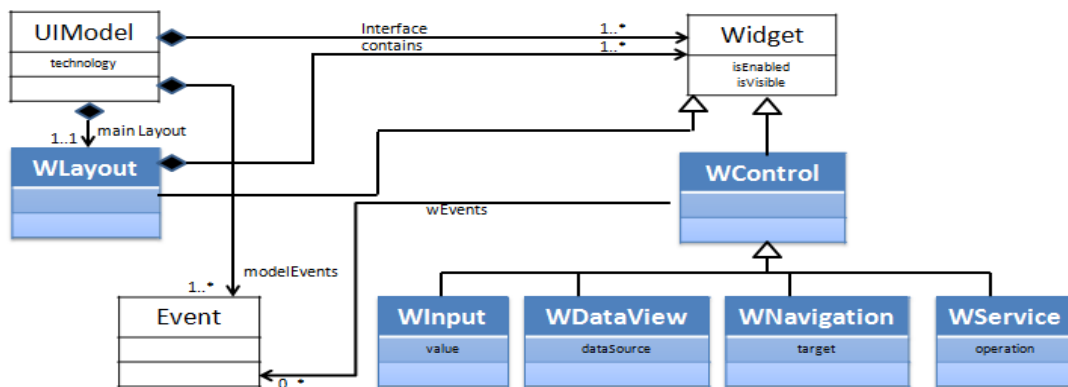


Figure 7: Static View of the RIA UI meta-model

2.4.3.2 Dynamic view of RIA UI meta-model

When a user interacts with the widgets, events are triggered which causes reactions on either the same widget or on other widgets. Figure 8 represents the dynamic view of the meta-model of the RIA UI. Five types of reactions are identified:

Changes to the UI (PropertyChange): This reaction results in a change of the UI properties of a target widget.

Request for Data on Demand (DataRequest): This reaction results in a request for information from the server, if it is not already available on the client-side.

Functionality Execution (Invocation): This reaction results in a requests-response communication with the domain logic.

Input Validation (Validation): This reaction results in a validation message when data is input in an input widget.

Navigation (UITransition): The navigation reaction results in changing the point from which the application UI is perceived by the user.

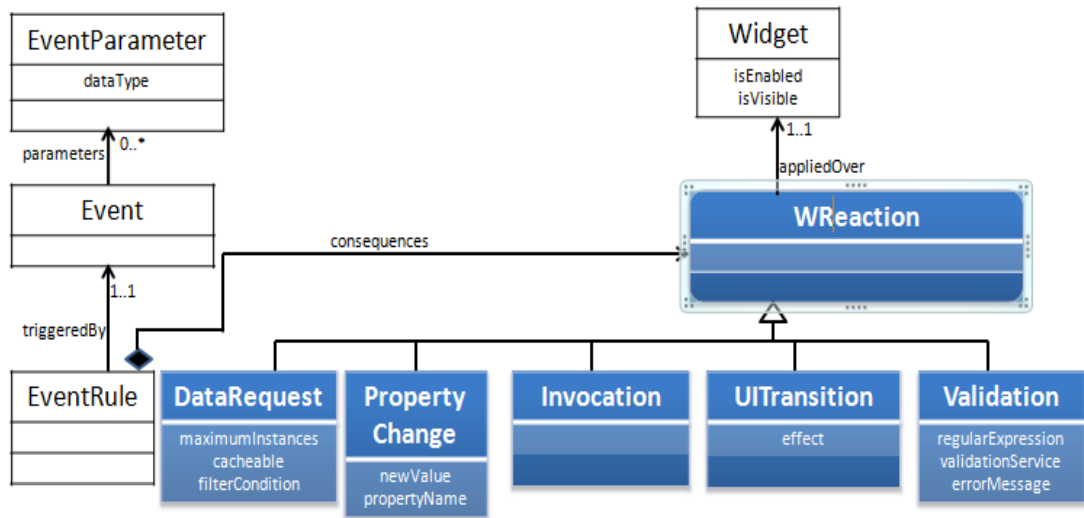


Figure 8: Dynamic view of the RIA UI meta-model

Figure 8 illustrates that each widget is associated with certain behaviour which are identified as reactions to user events. The reactions may trigger other activity either on the client side or on the server side. Like the static view, the consideration of only five types of reactions at a meta-level reduces the complexity of modelling the behaviour of individual RIAs.

2.4.4 Model Driven Engineering (MDE) of RIAs

In Section 2.4.1 some RIA development methods such as OOWS for RIA, OOH4RIA, UWE-R and OOHDH Extension for RIA were mentioned. These models attempt to provide a systematic approach to the various activities in RIA development. However, the employment of these systematic processes increases the manual workload though the developed software may be more efficient and easy to maintain. That is little, or no time is saved during the construction process. Hence in 2001, MDE was proposed to automatically generate software from models. MDE is an engineering approach for the automatic production of software from simplified models rather

than detail rich models. The primary difference between the MDE approach and the traditional approach to Software Engineering is - in MDE a model is an active artifact that is automatically transformed to an implementation whereas in the traditional approach the models are passive artifact used by IS professionals while manually producing software(Kent 2002). The automatic transformation assures that models and the final implementation are always in sync without manual effort.

The strength of MDE is that it can potentially allow people with no programming skills to develop applications. For example Valderas, Pelechano and Pastor (2006) use a MDE approach to develop web applications from end user conceptual models. The conceptual model is in the form of end-user interaction task models wherein an interaction task model captures UI interactions. They use ConcurTaskTree (CTT) to model the end-user interaction tasks. CTTs are graphical tree structure models that capture end-user interaction tasks as well as the chronological relationships in which the tasks are performed. They also use Activity Diagrams to capture the associated domain entities required for each user interaction task. Here activity diagram refers to an UML construct to model both computational and organizational processes.

A popular architectural framework for MDE is the Model Driven Architecture (MDA) by the Object Management Group (OMG), a not-for-profit computer industry consortium (Architecture Board ORMSC 2001). MDA is a software design approach with a set of guidelines for structuring specifications in the form of models and facilitating transformations between different model types using automated tools and services. Essentially in MDA, designers create Conceptual Models that are generally computation independent. That is Computation Independent Models (CIM) depict the business needs of the application using business user vocabulary and hides the computational details. Conceptual models are later transformed to Platform Independent Models (PIMs) and further on to Platform Specific Models (PSMs). PIMs make it easier to validate the correctness of the model without getting cluttered by platform specific details. Here “platform” is a technological solution to provide consistent functionality through interfaces. Examples of platforms include operating systems, programming languages, relational database management systems and client-side interfaces of systems. A PIM refers to a model that contains no platform

specific details to enable its mapping to any desired platform by transformations. “The PIMs provide formal specifications of the structure and function of the system that abstracts away technical details”(Architecture Board ORMSC 2001, p.6). A Platform Specific Model (PSM) is a model of a software or business system that is linked to a specific technological platform. A PSM combines the specifications in the PIM with the details required to stipulate how a system uses a platform.

2.4.5 Visual mock-up approaches to software development

A leading group of end user software engineering researchers observe that there is not much work done on modelling of interactive web-based applications by non-programmers (Ko et al. 2011). Although non-programmers (such as BAs) want to develop web applications and are capable of envisioning interactive applications currently available tools only realize a fraction of their potential (Rosson, Ballin & Rode 2005). This can change by utilising visual or User Interface mock-up approaches to software development.

UI mock-ups are increasingly used for requirements gathering and validation (Panach et al. 2008; Zhang & Chung 2003) and for prototyping UIs of software systems (Hartson & Smith 1991). This is mainly because mock-ups are useful for programmers and comprehensible by analysts and nonprogrammers at the same time (Mukasa & Kaindl 2008). As a consequence, several approaches have used them for starting the web development process (Bouchrika et al. 2013; Milosavljevic et al. 2013; Rivero et al. 2014).

In a study, Ferreira, Noble, and Biddle (2007) analyse the integration of UI design in Agile development processes. The results obtained from conducted interviews reveal that UI mock-ups guarantee the usability of the applications and are more effective than ‘user stories’ for describing requirements. User stories are mostly used by agile processes to gather requirements where a user story describes the functionality that is valuable to either the user or the purchaser of the system (Cohn 2004).

This section focuses on study of several mock-up-based tools for application development. These include MockupDD by Rivero et al. (2014), Kroki by Milosavljevic et al. (2013), INSPECTOR by Memmel and Reiterer (2008), WED by Störrle (2010), AppForge by Yang et al. (2008), RAINBOW by Ramdoyal, Cleve, and Hainaut (2010), Ebase Xi rebranded as verj.io² by Ebase Technology (*Introduction to Ebase Xi* 2017) and XIDE by Vuorimaa, et al.(2016).

Rivero et al. (2014) propose MockupDD tool for agile web development starting from user interface mock-ups. Here end-users create the mock-ups during the requirements analysis phase by using mock-up tools such as Balsamiq³. Then the mock-up is manually enriched with tags to add semantics to the UI elements. The tagged mock-up is then translated to platform-independent UI specification from which further conversions are made to get domain, navigation and presentation models of a web application. Thus, mock-ups are used both for requirements gathering and model derivation. However, this approach is not holistic in the sense that the mock-up designer only specifies the structural UI requirements, whereas the enrichment of the mock-up for concerns such as for navigational semantics requires intervention by Information System developers. Finally, after a series of transformations, an executable prototype is obtained from the mock-up.

Similarly, Coyette and Vanderdonckt(2005), Rivero et al.(2010), Rivero, Grigera, Rossi, Luna, and Koch (2011) exploit mock-up based approaches to auto-generate code that implements user interface skeleton, while the rest of the application is implemented manually.

Bouchrika et al.(2013) propose mock-up-based navigational diagrams to formalize the outputs of requirements analysis. These diagrams contain essential components relating to user interaction and navigational information. The diagrams are graphs in which the nodes either represent the pages or business entities, whilst the edges linking the nodes signify transition events. The mock-up diagram can be utilized with appropriate heuristic algorithms to generate UML diagrams and test cases.

² <http://verj.io/>

³ <https://balsamiq.com>

Kroki is another mock-up based tool for end-user participatory development of business applications (Milosavljevic et al. 2013). Kroki mock-ups are created by active participation of end-users during requirements solicitation and are enriched with implementation details during design and implementation phases for code generation of business applications. Hence web applications development through Kroki requires both end-user as well IS professionals.

UI mock-ups have also been used in conjunction with other artifacts to obtain mixed models. As an example, Cohn (2004) links appropriate segments of mock-ups with 'user stories' for finer granular details. Similarly Homrighausen, Six, and Winter (2002) link mock-up segments with UML Use Cases to derive prototypes capable of updating themselves whenever requirements change. A use case is a list of actions defining the interactions typically between a user and a system to achieve a goal. Conversely, the UC Workbench tool auto-generates mock-ups from Use Cases which are themselves auto-created from user stories. Thus an agile team for example can create new mock-ups as soon as new use cases are identified which can reduce client-analyst miscommunications (Nawrocki & Olek 2005).

The INSPECTOR tool developed by Memmel and Reiterer (2008) integrates and interconnects several types of modelled artifacts in an interactive UI mock-up providing good support for engineering at any level of design stage. Though this tool is not meant for auto-generation of artifacts, it provides an integrated tool to reduce miscommunication among diverse engineering personnel, for example in the car design industry.

Störrle (2010) uses window-event diagrams (WED) representing a combination of user interface models and state charts. WED uses GUI elements and a variant of UML state transition diagram to focus on the logic and functionality of the desired system. It follows a three-step process to obtain prototypes of UIs with navigations. In the first step, prospective states are proposed, collected and systematized. The result of the first step is a draft of the GUI, containing only the main windows and arcs. In the second step, the draft is refined into a complete WED, by specifying detail structure details of the windows along with arcs and menus for refined interaction details.

Finally, in the third step it generates the prototypes of the UI. By integrating WED with storyboards and UML, it provides a common artifact that both designers and software engineers readily understand.

The visual mock-up literature also discusses some approaches for automatically deriving database schema from user interface specifications. For example, AppForge is a What You See Is What You Get (WYSIWYG) application development platform for users to graphically specify the components of web pages inside a Web browser, and to automatically generate the corresponding database schema and application logic. AppForge allows users to continuously refine their applications giving instantaneous feedback but it constrains them to follow non-intuitive rules during the construction of the web pages. As an example, the views are always presented as in table form, while the relationships between entities are derived and presented only as nested views. Moreover, no explicit navigation links can be used in the application. These constraints conflict with the way non-programmers generally view web pages.

A similar approach is followed by RAINBOW, a tool for discovering content models from form-based user interface prototypes (Ramdoyal, Cleve & Hainaut 2010). Brogneaux, et al.(2005) discuss a methodology to draw user interfaces using third party software, and converting them to an XML format using a language called USIXML. The database conceptual schema is then deduced based on the form components within the interface, using a semi-automated approach.

Ebase Xi by Ebase Technology is a rapid web application development platform comprising: (i) a WYSIWYG tool editor, (ii) a Business Process Manager component for designing workflow-enabled business processes, and (iii) an Integration Server for the development and deployment of web services (*Ebase Technology - Getting Started* 2005). Web services extend the web infrastructure (such as HTTP, XML, JSON, SOAP/REST) so that one software can utilize the services of another software application without worrying about how the invoked web service is implemented. Ebase Xi uses a single technology platform covering the entire scope of WebApps to simplify the development process. Architecturally, Ebase includes a designer and a server component. The former runs stand-alone on the desktop and is used during

development and testing phases. The developed applications are deployed to the server. Ebase also supports integration with external resources, such as databases or web services.

XIDE is a mark-up language based platform built on a reusable component-based architecture to support XHTML programmers in the development of database-driven web applications(Vuorimaa et al. 2016). XHTML stands for Extensible Hyper Text Mark-up Language and is a family of XML Mark-up languages that mirror or extend versions of the widely used Hypertext Mark-up Language (HTML). XIDE uses XForms and XHTML on the client side, together with XForms and XFormsDB Processors on the server side. The users of this system are expected to have knowledge of XHTML and XForm for specifying advanced functionalities in the web application.

Table I summarizes the major features of some of the tools discussed in this section with a perspective on the required skills of targeted users of the tool. It may be observed that each of them has some strengths and weakness regarding the auto-generation of web applications from visual mock-ups. Some support the automatic generation of the database, others require knowledge of non-intuitive scripting languages, while some others require knowledge of existing components. Additionally, some only capture structural UI and navigation information without capturing the functional behaviour while some require expert developers to specify the behaviour at later stages, and others enforce design restrictions on the users of the tool. Hence it can be concluded that the mock-up tools currently available are not suitable for BAs to take on additional roles as developers of applications. The next sub-section provides a summary of the web modelling discussions in this chapter and its implications on BAs as developers of applications.

Table 1: Skills required for some mock-up tools

	XIDE	Ebase Xi	MockupDD	AppForge
Targeted Users	Advanced end-users	Advanced end-users	End-users	Advanced end-users
Holistic Tool	Yes. Fully integrated development tool	Yes. Fully integrated development tool	No. Needs several tools for development	No. Requires manual effort.
Knowledge of Scripting language	Yes. Requires XHTML and XForms knowledge	Yes (scripting language)	Yes (tagging language)	No
Knowledge of renewable component	Yes	Yes	No	No
Support in design for look and feel	Poor. Depends on the availability of reusable components	High	High	Poor
Intuitiveness for non-programmers	No. Depends on the quality of the available components	No, not for non-programmers	No, not for non-programmers.	Yes, but enforces prescribed rigid structure
Automatic derivation of database (DB) from mock-up	No	No. DB must be pre-defined	No. DB must be pre-defined	Yes. The table structure of the form defines the DB
Automatic derivation of behaviour from mock-up	Yes. May require coding if required components are not found	Yes	Yes. It requires intervention by professionals	Yes, but limited to operations on tables

2.4.6 Summary on modelling of web applications and its implications for BAs

Chapter 1 highlighted that existing agile approaches of software development are not adequate for SMEs and recommended using the services of BAs both for the analysis as well as the developmental activity. However, the literature also drew attention to

the fact that BAs do not get adequate support to take on additional developmental responsibilities. Hence the aim of this research was identified - to support BAs in building RIA for SMEs. To do this, a review of the prevailing approaches to web application development was conducted in this chapter. Specifically, the chapter drew attention to the need for RIA application architecture to be not just flexible to quickly adapt to changing business requirements of SMEs but also easy to use by customers and easy to develop by BAs. In addition, the current literature review revealed that traditional RIA development methods are cognitively challenging because they are largely based on OODD approach. Hence, they are not suitable for BAs as developers. Furthermore, currently available technological tools either require deep knowledge of database structures or business objects to create fully functional web systems which are beyond the realm of BAs. The chapter also threw light on visual mock-up approaches to create prototypes and in some cases functional applications. The WYSIWYG effect of visual mock-up approaches mean that they are less cognitively demanding than other approaches of web development and are suitable for BAs. Thus, BAs may specify the requirements of the system as a visual mock-up and then a tool could be used to auto-generate the RIA from it. However existing visual tools for auto-generation of web apps are not holistic, they require many models to be created and manually integrated. Another drawback of the existing visual tools is that they do not auto-generate the database structure and the basic functionality required for common SME operations. Rather in most cases the database structures are required to be pre-defined manually. In addition, most of the existing visual languages are not intuitive to learn and use. These factors hinder the uptake of existing visual mock-up tools by BAs for web development activity.

3 RESEARCH QUESTIONS AND RESEARCH DESIGN

Chapter 2 highlighted the lack of support available to BAs to take on software developmental roles. So, a mock-up-based tool to auto-generate SME application is recommended. This chapter first discusses the research question on how to develop such a tool and then discusses a research method to guide the design of the tool. Specifically, Section 3.1 addresses the research question, Section 3.2 discusses Design Science Research in IS (DSR in IS) as a research method to generate innovative designs and Section 3.3 provides a summary of the chapter.

3.1 Research question(s)

The main research question is:

How to design a tool to help BAs to develop a fully functional Rich Internet Application for small to medium enterprises, holistically from visual UI requirement specifications using a visual mock-up language?

The above question deals with three main themes: a visual mock-up language to capture SME application requirements, auto-generation from a visual mock-up and the design of a tool to develop a fully functional RIA. Hence the above question can be split into the following three investigative questions:

Research Question 1 (RQ1): What is a suitable visual mock up language to fully capture the SME application requirements?

Research Question 2 (RQ2): How is a Rich Internet Application for a Small to Medium Enterprise auto generated from a visual mock-up?

Research Question 3 (RQ3): How can the auto-generating tool be validated?

RQ1 deals with the expected features of a suitable visual mock language and how it needs to be integrated within the tool to fully capture the major features of SME applications. Hence the first issue is to find the main (generic or common) features of SME applications. Knowing the requirements of the generic features will help in defining the second issue which is identifying the features of the mock-up language to express the requirements. Finally, the third issue is the suitability of the visual mock-up language for BAs. This issue is important because the language and the tool are primarily meant for BAs who are traditionally considered to have low technical skills. Hence the language should not be cognitively challenging. Furthermore, it should not be too abstract. Rather if it mimics the WYSIWYG approach to succinctly capture the structure, behaviour and navigation of the system then BAs should find it usable. Secondly the language and the tool should be easily integrated to avoid extraneous cognitive load on BAs during its usage. Extraneous cognitive load refers to the extra cognitive effort required for activities not directly related to the problem solving task (Sweller 1988). An example of this is the actual hands-on effort required to physically place the widget within a page, using the tool.

Hence RQ1 is further sub-divided as follows:

RQ1.1: What are the generic requirements of SME applications?

RQ1.2: What are the features of a visual mock-up language that make it suitable to fully express the requirements?

RQ1.2: How are the features integrated into a tool?

RQ2 attempts to identify how the various components of RIA are auto-generated from a visual mock-up. From the background literature review in Chapter 2 it is evident that a RIA requires a database and client and server-side components to manage the application logic and database operations. Hence these components need to be auto-generated. The client and server-side components to be auto-generated should also include the Models, Views and Controllers in an optimal RIA architecture discussed in Section 2.2.2.

Hence RQ2 is further sub-divided as follows:

RQ2.1: How is the database structure of the RIA auto-generated from a visual mock-up?

RQ2.2: How are the client side and server-side components of the RIA auto-generated from a visual mock-up?

RQ2.3: How is the database logic for Create, Retrieve, Update, and Delete (CRUD) operations auto-generated from a visual mock-up?

RQ2.1 and RQ2.3 focus on the modelling of the database structure and how CRUD operations on the database are auto-generated. Since all components of a web application are integrated together at the database level, for successful functioning of an application, the auto-generation of a correct database model is paramount. If the database models are correct, the Models, Views and the Controller components can function correctly as well. RQ2.2 deals with the design of the architectural structure of the RIA to be auto generated. That is the architecture has a direct impact on the behavioural model of the application. The literature review discussion in Section 2.2.2 highlights the importance of an optimized architecture for low latency of response during client-server communication in RIAs. Behavioural model refers to the modelling of the components required for client-server communication to process a business request. This involves the modelling of the Model, View and Controller components discussed in Section 2.2.2. Thus, this question deals with how the Models, Views and Controllers can be auto-generated from the information provided in the mock-ups. Figure 9 highlights the auto-generating components desired to answer research question 2, once a BA provides the mock-up of a SME application. It indicates three components are desired, each corresponding to the investigative sub-questions RQ2.1, RQ2.2 and RQ2.3 respectively.

Finally, RQ3 deals with the validation of the tool. This can be perceived with respect to three areas: the usability of the mock-up language, the usability of the auto-generated RIA, and the usability of the tool. The usability of the mock-up language needs to be validated for specifications of typical operations of SME applications. Validating usability of the auto-generated RIA means the auto-generated application's functional and non-functional features are deemed usable by SMEs.

Finally, the validation of the auto-generating tool ensures the tool is usable as an integrated system.

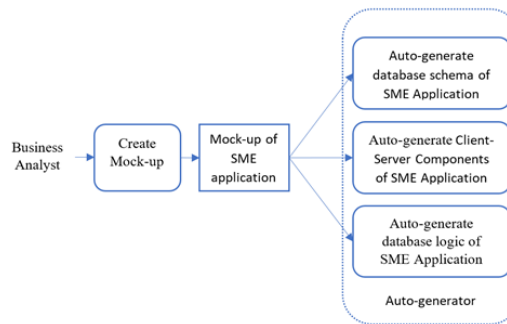


Figure 9: Auto-generating components desired to answer research question 2

Hence RQ3 is further sub-divided as follows:

RQ3.1: How is the usability of the mock-up language validated?

RQ3.2: How is the usability of the auto-generated RIA validated?

RQ3.3: How is the usability of the tool validated?

Table 2 provides an integrated view of all the research questions.

Table 2: Integrated view of the research questions

Research Questions	Sub Questions
RQ1: What is a suitable visual mock up language to fully capture the SME application requirements?	RQ1.1: What are the generic requirements of SME applications
	RQ1.2: What are the features of a visual mock-up language to fully express the requirements?
	RQ1.3: How are the features integrated into a tool?
RQ2: How is a RIA for a SME auto generated from a visual mock-up?	RQ2.1: How is the database structure of the RIA auto-generated from a mock-up?
	RQ2.2: How are the client side and server-side components of the RIA auto-generated from a mock-up?
	RQ2.3: How is the database logic for Create, Retrieve, , Update, and Delete (CRUD) operations auto-generated from a mock-up?
RQ3: How is the auto-generation tool validated?	RQ3.1: How is the usability the mock-up language validated?
	RQ3.1: How is usability of the auto-generated RIA validated?
	RQ3.3: How is the usability of the tool validated?

3.2 Research Design

Any research requires a research design method to conduct the research in a systematic manner. This section discusses Design Science Research in Information Systems as a suitable method to guide this research.

The design and development of the expected tool from which a fully functional RIA is generated is innovative in nature. It covers new ground in the form of a new visual modelling language and transforming UI mock-up-based requirement specifications into full fully functional transactional web applications for SMEs. The transformation results in the creation of UI models, navigation models, MVC models and data models from the visual mock-ups. It is innovative because existing mock-up-based tools are not holistic in nature. To-date no tool is found in the literature that derives the behavioural and database model from a single conceptual level mock-up model. The creation of innovative Information Systems designs can be aided by following the Design Science Research (DSR) in Information Systems (DSR in IS). DSR is a research method where knowledge and understanding of a wicked problem and its solution is gained while designing an artifact and during the application of that artifact. A wicked problem is a problem that is difficult to solve because of incomplete, contradictory, or changing requirements that are often difficult to recognize, and an artifact refers to something that is created artificially by humans either in the form of *constructs* (vocabulary), *models* (abstractions), *methods* (algorithms) or *instantiations* (March & Smith 1995). In this research all four types of artifacts are produced though the focus is on *instantiations*. DSR in IS a popular research method employed to guide IS design and to capture the knowledge created during the design process (Hevner & Chatterjee 2010a). Its main goal is to improve the effectiveness and utility of IT artifacts in solving business problems where creativity and innovations are often necessary to improve organizational effectiveness and efficiency (Baskerville, Pries-Heje & Venable 2009). DSR is like action research method in certain respects. In both methods, the researcher is not an independent observer but is an active participant intentionally modifying a setting and carefully evaluating the result. However, an action researcher's aim is creation of change in an existing set-up to fix social illness

whereas the DSR aim is creation of an artifact, though both strive to discover new knowledge in the process of achieving their aim. Furthermore, methodologically action research is set up in client situations whereas DSR is set up in laboratory situations (Baskerville, Pries-Heje & Venable 2009).

3.2.1 Design Science Research in IS

IS artifacts developed using DSR methodology are normally prototypical innovations that define the ideas, practices and technical capabilities (Hevner, March & Ram 2004). DSR in IS guides the design process and discovers the utility of the artifact in a business organization. DSR in IS consists of three cycles of research, namely the Relevance Cycle, the Design Cycle and the Rigor Cycle as illustrated in Figure 10.

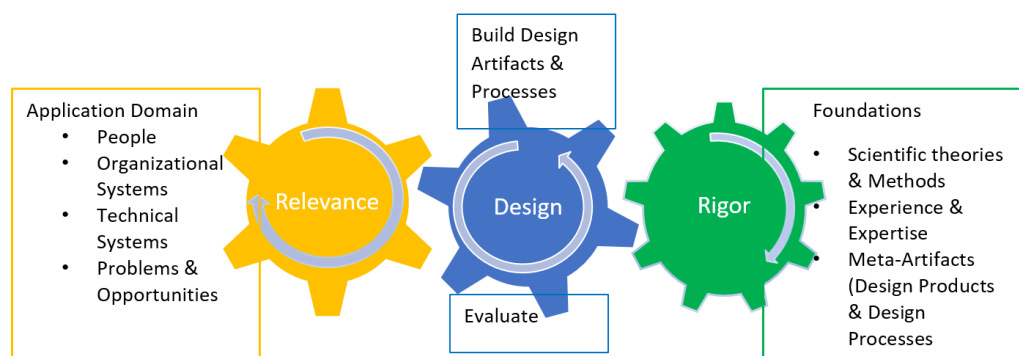


Figure 10: Design Science Research cycles

The Relevance Cycle identifies the design requirements in relation to the research question. That is, it identifies the SME application requirements, features of the mock-up language that are useful from a BA perspective, how it is to be field tested and what metrics are used to demonstrate the successful use of the artifact by BAs. “Together these define the business needs or problem as perceived by the researcher” (Hevner, March & Ram 2004, p.79). The Rigor Cycle ensures that the design is based on scientific theories and methods to produce a new knowledge base of artifacts that are useful to the society. The Design Cycle iterates between the core activities of building and evaluations of the design artifacts. It identifies how the artifact is represented, what design process or heuristics are used to build the artifact

as well as what evaluations are performed during the design. It also identifies design improvements based on feedback obtained from usability validations during the Relevance Cycle.

Figure 11 highlights the research plan based on DSR in IS. In this figure the Relevance Cycle activities are shown in orange coloured boxes and the Design Cycle activities are shown in blue coloured boxes. The Rigor Cycle activities are mostly related to information flow from the knowledge base during the Relevance Cycle and Design Cycle activities. However, the Rigor Cycle also has a distinct activity to disseminate the knowledge gained because of the research. This is shown by the red coloured box in Figure 11. The information flow resulting from the Rigor Cycle activity is represented by green arrows in Figure 11. The figure also shows the process flow during the various cycles of the research. This is represented by dotted black arrows.

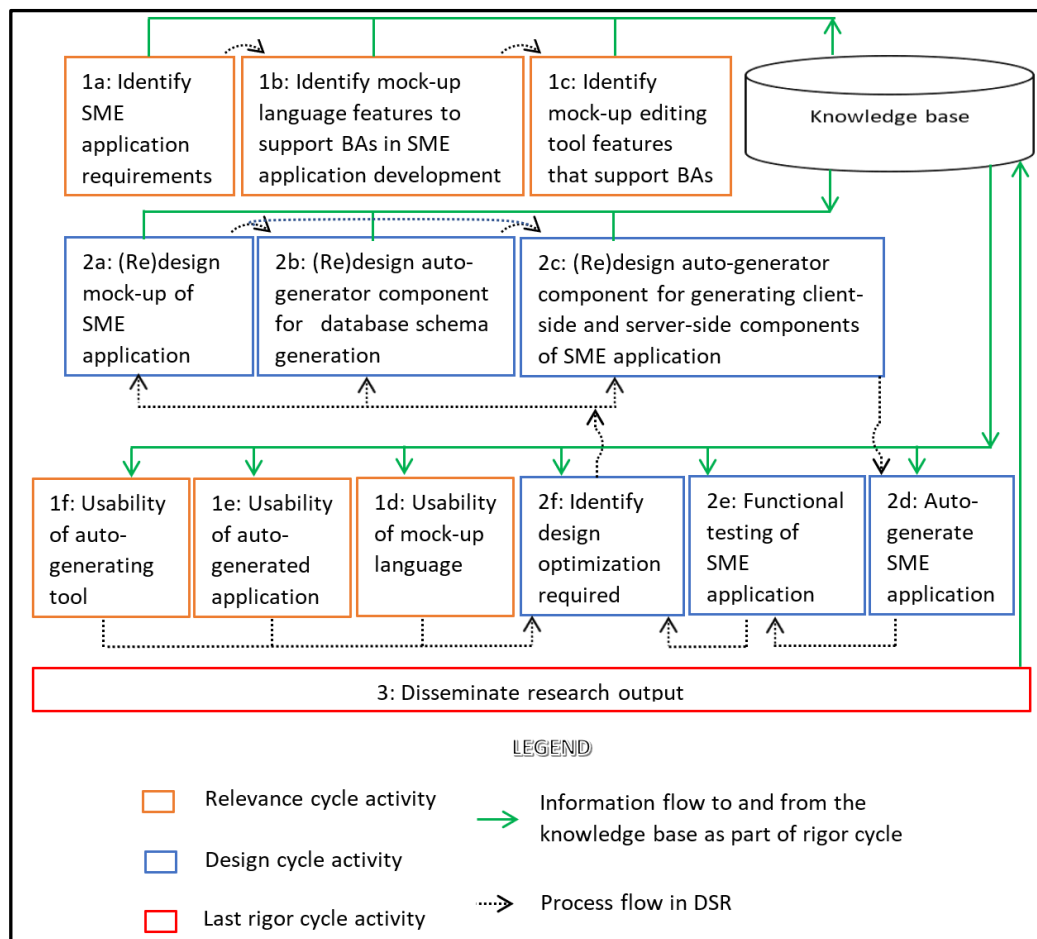


Figure 11: Research plan based on DSR in IS

In the Relevance Cycle, the SME application requirements are identified (shown as 1a in the figure), this in turn leads to the identification of the mock-up language features, to support a BA in SME development (activity 1b in the figure). Next, the desirable features of a mock-up editing tool are identified to ensure that BAs are not burdened by the tool during the creation of the mock-up (activity 1c in Figure 11). It may be observed that there is a close relationship between the Relevance Cycle activities 1a, 1b, 1c and the corresponding research sub questions RQ1.1, RQ1.2, RQ1.3 respectively. The last three and perhaps the three most important activities of the Relevance Cycle are to ensure the artifact is useful to the stakeholders (BAs). These activities focus on the usability aspects of the artifact. A systematic approach to usability testing of the auto-generated application, the mock-up language and the tool, is necessary to ensure the tests are objective, consistent and repeatable. Specifically, block 1d in Figure 11 aims to validate the usability of the mock-up language, while block 1e deals with the usability of the auto-generated application and block 1f considers the usability of the tool in general, as perceived by the BAs. These three activities of the Relevance Cycle ensure that the auto-generating tool is satisfactorily validated, thereby providing answers to research questions RQ3.1, RQ3.2 and RQ3.3. The artifact is optimized based on the usability test results until acceptable usability ratings are obtained. However, before the usability of the tool is validated, designs of the research artifacts need to be produced. This activity is performed in the Design Cycle and is discussed in the next paragraph.

Once the desired features of the mock-up language are identified, the Design Cycle kicks in with the design of the mock-up of a SME application (see block 2a in Figure 11). This is followed by the design of the auto-generating component for deriving the database schema of the SME application from the mock-up (see block 2b in Figure 11). Once the database design is derived, the components for auto-generating the client-side and service application logic is designed. This is shown by block 2c in Figure 11. At this stage the artifact is ready for a series of internal testing cycles. This is done by auto-generating the SME application from the mock-up (block 2d in figure) followed by its functional testing (block 2e). Functional testing identifies the design optimizations required to accomplish satisfactory functional behaviour of the SME

application. Next the artifact is optimized based on a series of iterations of activities in blocks 2a through to 2e. It may be noted that block 2a is a Design Cycle activity wherein SME application requirements are used as examples to kick-start the design activity. It may also be observed that block 2b is related to research question 2.1, namely the derivation of database structure from a mock-up. Similarly block 2c is related to RQ2.2 and RQ2.3 regarding the auto-generation of application and database logic from the mock-up. Finally, when the artifact is deemed to functionally correct, it is ready for field testing by BAs. At this stage (block 2a) in the Design Cycle, BA designed mock-ups, are input to the auto-generating components in 2b and 2c. As mentioned earlier, blocks 1d, 1e and 1f in Figure 11, ascertain the usability aspects of the artifacts during the field testing process.

Finally, when the artifacts are deemed to be acceptable from BA perspective, dissemination of research outputs is done. At this stage results are disseminated via publications through research conferences and journal articles. This activity is represented by the red block in Figure 11.

The above discussion provided an overview of how DSR in IS will be used to plan this research. DSR in IS pioneers, Hevner, March and Ram (2004), have provided a conceptual framework with clear set of seven guidelines for understanding, executing, and evaluating DSR in IS. These set of guidelines will be introduced in the next section to highlight how DSR in IS will be followed in the context of this research.

3.2.2 Guidelines for conducting DSR in IS

Table 3 below provides the guidelines for conducting DSR in IS. These guidelines are popularly used for driving the research process.

Guideline 1 necessitates the creation of an innovative artifact to solve a specific problem which is identified by Guideline 2. Guideline 3 ensures the utility of the artifact for the stated problem. Guideline 4 safeguards the artifact created during the process truly represents new knowledge hitherto unknown to the world and hence this new knowledge is articulated and documented so that it can be exploited by the society in the future. Guideline 5 warrants the artifact is rigorously defined using

existing knowledge so that is built on solid foundations. Guideline 6 focuses on documenting the search or heuristic process by which the artifact is created and made useful so that it can be recreated. Finally, guideline 7 ensures that dissemination of new knowledge because of the artifact targets towards both technical and managerial audience(Hevner, March & Ram 2004).

Table 3: Design Science Research guidelines adopted from Hevner, March and Ram (2004)

Guideline	Description
Guideline 1: Design as an Artifact	Design-science research must produce a viable artifact in the form of a construct, a model, a method, or an instantiation.
Guideline 2: Problem Relevance	The objective of design-science research is to develop technology-based solutions to important and relevant business problems.
Guideline 3: Design Evaluation	The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods.
Guideline 4: Research Contributions	Effective design-science research must provide clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies.
Guideline 5: Research Rigor	Design-science research relies upon the application of rigorous methods in both the construction and valuation of the design artifact.
Guideline 6: Design as a Search Process	The search for an effective artifact requires utilising available means to reach desired ends while satisfying laws in the problem environment.
Guideline 7: Communication of Research	Design-science research must be presented effectively both to technology-oriented as well as management-oriented audiences.

In addition to the seven guidelines, Hevner and Chatterjee (2010b) provide a checklist of eight questions to ensure the researcher has carried out all the activities in the three cycles. A brief introduction to the checklist is provided in the next section.

3.2.3 Checklist for DSR in IS

Table 4 highlights a list of checklist questions that can be applied to assess whether the DSR method has been successfully applied in this research. Answers to these questions will be addressed in the last chapter (Chapter 7).

Table 4: Checklist for DSR in IS

DSR Checklist questions
What is the research question (design requirements)?
What is the artifact? How is the artifact represented?
What design processes (search heuristics) will be used to build the artifact?
How are the artifact and the design processes grounded by knowledge base? What, if any, theories support the artifact design and the design process?
What evaluations are performed during the internal Design Cycles? What design Improvements are identified during each Design Cycle?
How is the artifact introduced into the application environment and how is it field-tested? What metrics are used to demonstrate artifact utility and improvement over previous artifacts?
What new knowledge is added to the knowledge base and in what form?
Has the research question been satisfactorily addressed?

3.3 Summary

This chapter introduced the research question and the investigative sub-questions in a quest to help Business Analysts in developing business applications using an auto-generating tool. DSR in IS chosen a suitable research method to guide the design of the auto-generating tool due to the innovative approach of using a UI mock-up of the business application as the input to the tool. A set of guidelines for DSR in IS are introduced to organize the research activities. Finally, a set of checklist questions are introduced to ensure the research activities conform to DSR in IS.

4 MOCK UP LANGUAGE SPECIFICATIONS

A tool that uses the mock-up language for BAs should be easy to use as well have adequate expressive power to satisfy SME web application requirements. This objective led to the first research question namely, “What is a suitable visual mock up language to fully capture the requirements?” This question was further divided in terms of investigative sub-questions RQ1.1, RQ1.2 and RQ1.3. RQ1.1 deals with finding the essential features of SME applications so that RQ1.2 can be answered to find the features of the mock-up language to express them. Finally, RQ1.3 is associated with how the features of the language are easily integrated with the features of the auto-generating tool. Easy integration means the tool should permit the mock-up designer to easily edit the mock-up before the auto-generation process. This chapter provides answers to these three sub-questions raised in chapter 3. Section 4.2 provide answers to RQ1.1 while Section 4.2 - Section 4.4 deal with RQ1.2 and Section 4.5 addresses RQ1.3. It may be observed that three sections are dedicated for RQ1.2. This is because based on the discussion in Section 4.2, a meta-model of the language is identified in Section 4.3 and a summary of the visual mock-up language is provided in Section 4.4.

4.1 Essential features of SME web applications

The development of an auto-generating tool for SME application is contingent on identifying SME application requirements. This section discusses how SME application requirements are identified. With regards to the broader research plan shown in see Figure 11, it discusses how activity 1a in the research plan is conducted. The essential features of web applications for SMEs are found by researching examples used in existing literature on modelling of business web applications. It is assumed that the common examples found in the literature focus on the essential components of business web applications and since SME applications are the

simplest of all business applications, it is reasonable to consider them to be the common features of SME applications. Moreover, this fits with DSR principles regarding design artifacts: “artifacts constructed in design science research are rarely full-grown information systems that are used in practice”(Hevner, March & Ram 2004, p.83). So, the features commonly referred to in the literature on the modelling of web applications can be considered to provide a good approximation of the answers to address RQ1.1, regarding the generic features of SME application requirements. Specifically, three examples of web application modelling and development are studied from the literature, which are discussed in the following paragraph.

In a study to illustrate design requirements of transactional web applications using WebML, Brambilla and Fraternali (2014) use an example of a simplified web content management system consisting of a product catalogue, a web interface for content publishing as well as for public transactions such as querying the catalogue and selecting products. Here the important business entities required are products and catalogues. The web content management system allows users to add new products to catalogues, delete products, update existing products in the catalogue as well as query the content management system for creation of reports. Similarly Koch et al., (2008) use a simple web application of a Conference Review System to allow users to add, update, delete, review conference papers by creating reports. Here the business entities are conference papers, authors and reviews. Yet another popular example of transactional web application found in the literature is the online car rental system where users can search, view (report), book and update details of rental cars. This example has been used by Valverde and Pastor (2009) to explain a Model Driven Engineering approach to develop of a web application.

The above three examples highlight five essential operations of transactional web applications. The essential operations are: The users of the application should be able to create (**insert**) new business entities or **update** them or **search** or create a **report** or **delete** them. All these business operations require a database with corresponding functionality at the database level. Hence a similar example is used in this thesis to illustrate the design research process. Specifically, an example of a transactional web

application for managing travel deals is considered. The application in the example allows users to create (insert) new travel deals, search travel deals, book travel deals, update travel deals or delete travel deals. Further details of the example are provided in the next section while dealing with the expected features of the mock-up language.

4.2 Mock-up language features to express the requirements of SME applications

The features of the mock-up language should be able to express the requirements of SME applications. The previous section highlighted the common set of behaviours exhibited by modern SME web applications as the ability to: create to new entities, update or delete existing entities, display and traverse through existing entities, search for entities or to create a report of entities. In addition, since business processes commonly operate in transactional mode, the mock-up language should have adequate expressive power to support these common behaviours. This section discusses how activity 1b in the research plan in Figure 11 is addressed. That is, it identifies the expected mock-up language features to support BAs in SME application development, as an answer to RQ1.2. At a conceptual level the expected features of the language can be perceived as the ability of the language to capture the layout structure and navigation for the common set of behaviours of SME applications. That is the mock-up language should have adequate expressive power to capture the layout structure and navigation for the creation, search & retrieval, update, deletion and report generation in a web application.

The visual mock-up of a Travel Deals web application for creating, searching, and updating, deleting, booking or reporting travel deals is chosen as an example to illustrate the expressive power of the language. The visual mock-up consists of one or more web pages. An overview of the mock-up of this application is shown in Figure 12. In general, the mock-up has a page to create a new travel deal. It also has a page to search a travel deal with an option to either update or delete or book a deal. If the booking option is chosen the mock-up allows a user to enter customer details,

address details and payment details to complete a booking transaction. Furthermore, the booked travel deal can be viewed and updated if necessary.

Specifically, the mock-up in in Figure 12 has 10 pages. It includes an "Add Administrator" page to create a new administrator entity. An "Add Travel Deal" page to create a new travel deal entity. A "Login" page for the administrator to login. A "Deal Management" page to manage administrator operations. A "Travel Deals" page to search and display travel deals. An "Order Deal" page to enable a customer to order a deal and do payments. A "Booking Confirmation" page to create a report of the order. It also has three update pages, namely, an "Update Travel Deal" page to update a travel deal, an "Update Customer Details" page to update customer details and an "Update Payment Details" page to update payment details.

Each web page is considered to have unique name and the visual language is case insensitive. In addition, widgets in a container are considered to have unique labels and a widget can be expressed using a fully qualified name. A fully qualified widget name can be expressed using a hierarchical path specification of the form: "widget w in a container c in a page p". Furthermore, each web page is made of one or more *Containers*. A *Container* allows specification and grouping of widget layout information in the mock-up. Each *Container* is considered to have a unique name when used in context of creation of new entities. A *Container* can be of one of the following types: *Database Field Yielding Containers*, *Search Containers*, *Search Result Containers*, *Data View Containers*, *Update Containers*, *Report View Containers* and *Navigation Only Containers*. Each of these will be discussed in detail on encountering them in the example.

Figure 13 represents the corresponding use cases of the Travel Deals example. It has use cases such as Create Travel Deal, Update Travel Deal, Order Travel Deal, Search Travel Deal, Add Customer Details, Add Customer Address, Add Payment Details, Update Customer Details and Update Payment Details. Actor roles are not shown in Figure 13 since user access control is not within the scope of this thesis. However user access control using mock-ups is published in a journal paper in collaboration with other researchers(Caruccio et al. 2015).

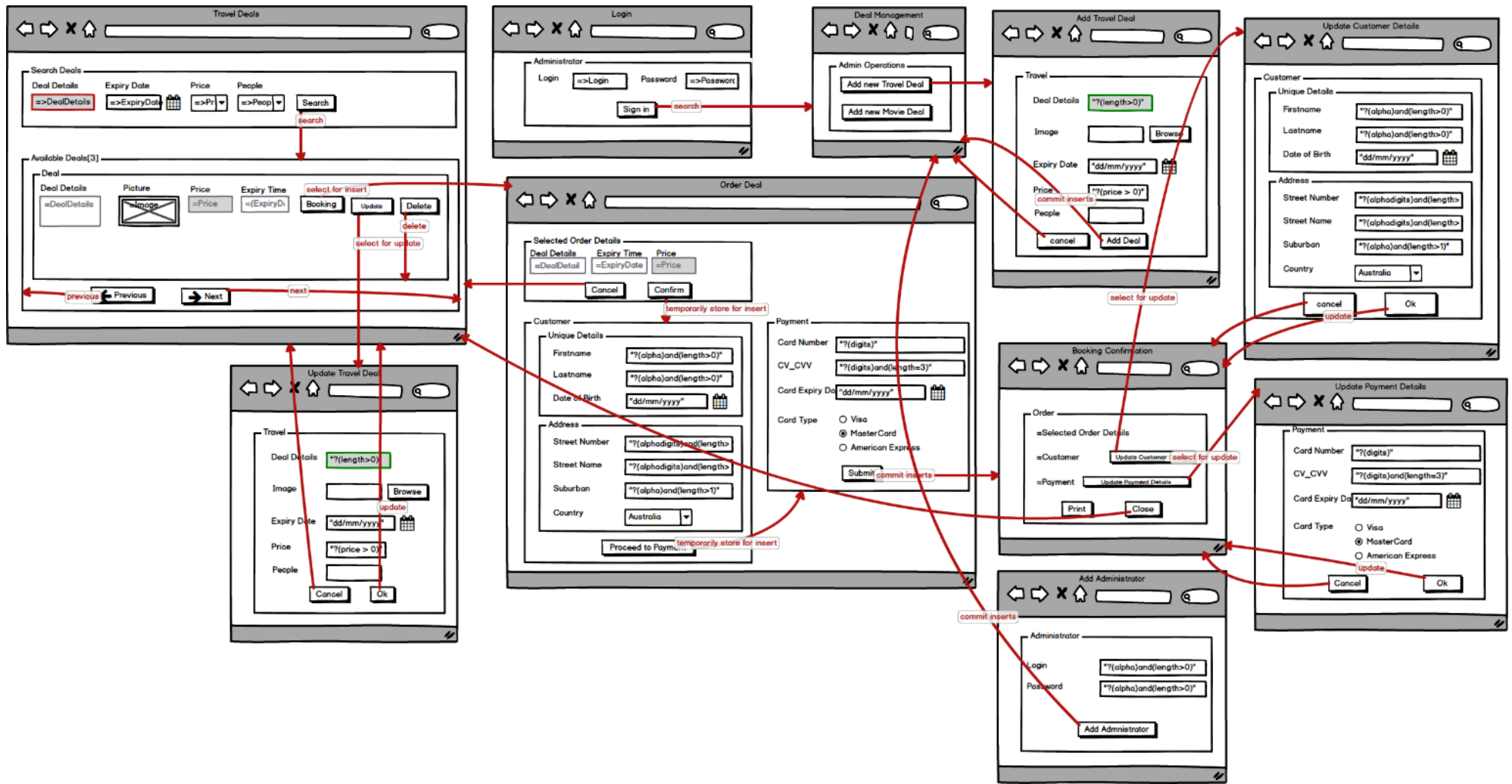


Figure 12: Mock-up for a Travel Deals web app

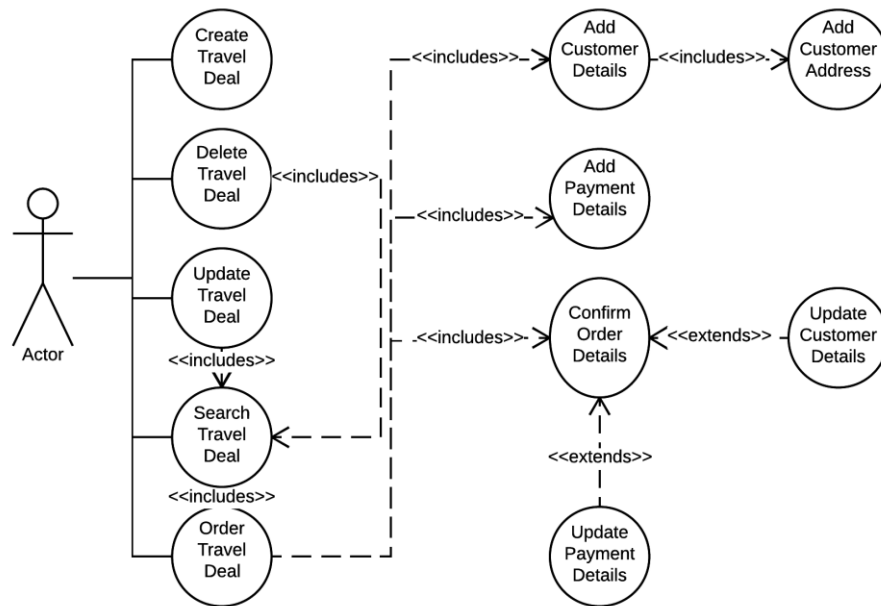


Figure 13: Use Cases in the Travel Deals web app

This section is organized in terms of five sub-sections. Section 4.2.1 to Section 4.2.5 are devoted to explaining how the features of the language are identified to create mock-ups of the web application to satisfy SME application requirements discussed in the previous section. Specifically, Section 4.2.1 addresses the features of the language to support creation of new business entities, Section 4.2.2 addresses how to support searching and subsequent display of business entities, Section 4.2.3 deals with the language features for managing an insert business transaction, while Section 4.2.4 deals with the features for managing report generation and finally Section 4.2.5 addresses how an update operation is supported.

4.2.1 Mock-up segment for creating new entities

Creation of new entities is one of the foremost operations in web applications. Once they are created, other operations such as searching, updating and deleting can be performed. This sub-section focuses on the mock-up language features for the creation of new business entities. Several types of business entities are needed in the Travel Deal example. These include, “Travel” representing a travel deal, “Administrator” with the responsibility for managing the operations on behalf of

clients, “Customer” representing a customer, “Address” representing a customer’s address and “Payment” representing a payment for booking a deal. However, in this sub-section, only the “Travel” and “Administrator” entities are considered since the same principles are used for other entities.

Figure 14 illustrates a part of the mock-up in Figure 12 for the creation of a new Travel Deal and a new Administrator entity in the database. It is assumed that Travel Deal entities and an Administrator entity is required in this system and the two entities are not related to one another. It also includes a mock-up segment for navigation control and sign-in.

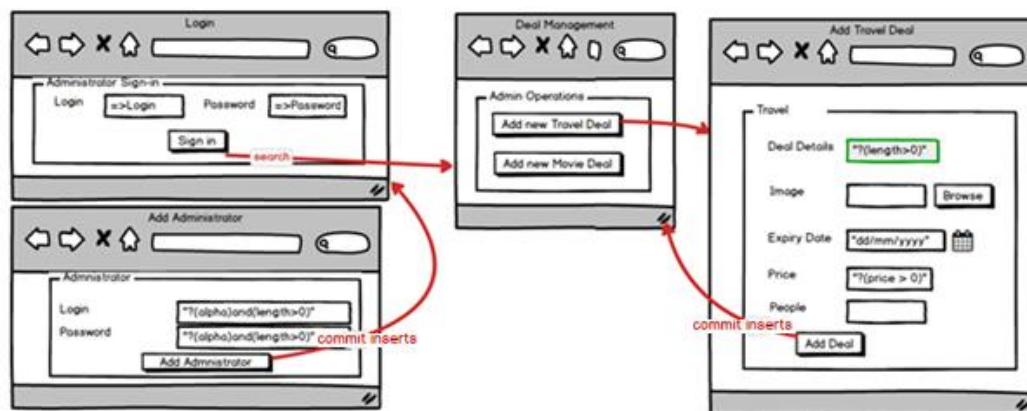


Figure 14: Mock-up segment for creation of Travel entity and Administrator entity in Travel Deal web app

Following features of the visual language will be introduced in this section:

- Use of unique names (or identifiers) for each page and for other widgets in a mock-up
- Case insensitiveness of the language.
- Use of a navigation widget with or without annotations
- Use of grouping widget called Container for organizing related elements of a business entity
- Identification of a specific type of data input widget called *Database Field Yielding Widget (DFYW)* and its relationship to a database field
- Specification of simple validation or formatting of input data

- Specification of search criterion in a type of data input widget called *Look-up widget*
- Difference between *DFYW* and a *Look-up widget*
- Specification of database table's fields using a type of container called *Database Field Yielding Container (DFYC)*.
- Specification of the insert behaviour in a database using the "**commit inserts**" annotation on a navigation widget.
- Specification of search criteria using a type of data input container called *Search Container*.
- Initiation of a search operation using the "**search**" annotation on a navigation widget
- Specification of a type of container called *Navigation Only Container* for grouping navigation widgets

In Figure 14, "Administrator" and "Travel" are examples of *Database Field Yielding Containers (DFYC)*. A *DFYC* is a container from which a database table's schema can be derived for an entity in the system. A *DFYC* has a unique label name in the mock-up and contains at least one or more data input widgets. Unique label name implies that if two or more *DFYCs* exist with the same name, the data input widgets in them should be similar. In our example, the "Travel" *DFYC* has five data input widgets with labels such as "Deal Details", "Image", "Expiry Date", "Price" and "People" and if any other "Travel" *DFYC* exists in the system, they would have the same set of data input widgets. Each data input widget in a *DFYC* is called "*Database Field Yielding Widget*" (*DFYW*) because it is used to identify a field in a database table. Each *DFYW* has a unique label in the mock-up. Based on the label text of a *DFYW*, the name of the database field and its data type is identified. If the data type cannot be identified, the default type for any field is assumed to be String.

As discussed briefly in Section 4.2, the mock-up language also captures anticipated behaviour. This is done using annotations on navigation links. For example, in Figure 14 the "**commit inserts**" annotation on the navigation widget linking the "Add Travel Deal" page and "Deal Management" page specifies a record is to be inserted in the database followed by navigation to the "Deal Management" page. Specifically,

“commit inserts” is a mock-up language key phrase used to commit one or more records into one or more tables in a database. In the mock-up, navigational information is modelled using an arrow and represents a transition between two widgets. Navigation is said to occur when the starting point of the arrow refers to a widget that is different from the destination widget of the arrow. For example, clicking the “Add Administrator” button widget on the “Add Administrator” page will cause navigation to the “Login” page.

The language also allows specification of validation strings in a mock-up. Simple validations of data in input widgets meant for storing new data in a database is represented by a string starting with a double quotation marks. A question mark operator “?” follows the first quotation mark to indicate that it is a validation string in a DFYW. For example, the “Add Administrator” page in Figure 14 contains the string “?(alpha) and (length>0) ” to formulate that the input data should contain one or more alphabets. The validation string may have one or more input conditions. Observe that each condition is set within the () brackets. Furthermore an **“and”** keyword is used to combine two or more conditions together. Examples of other validation conditions can be “?(digits) ” to specify that the input elements can only be digits and “?(digits) and (length=3) ” to indicate it should be a three-digit input. Other examples are: “?(price>0) ” to indicate that the input value of a corresponding “price” data input widget should be greater than 0. The “Add Travel Deals” page also exhibits another feature of the language for format specification for input data. For example, the “Expiry Date” Text Input widget in the “Add Travel Deals” page contains the “dd/mm/yyyy” format specifications for inputting the date.

In addition, Figure 14 illustrates other mock-up features though not related to the creation of new data entities. Brief explanations of these features are provided below for completeness of the discussions based on the figure. For example, it contains *“Admin Operations” Navigation Only Container*. The purpose of a *Navigation Only Container* is to contain links or buttons for navigational purpose. Such containers are normally used as headers or footers in web page designs. In addition, the *“Administrator Sign-in” Container* has widgets that are like the widgets in the *“Administrator” Container*. However, the visual notation used in the *“Administrator”*

is different from that in the “Administrator Sign-in”. The principal difference is “Administrator” is an example of a *Database Field Yielding Container* but “Administrator Sign-in” is a *Search Container*. At run time when the user clicks the sign-in button in “Administrator Sign-in” *Container*, the data entered is looked-up (searched) against existing records in the database and if a match is found the user navigates to the next page. The specification of the “**search**” annotation on the navigation widget that links the “Sign-in” button and the “Deal Management” page in Figure 14, is used to capture the trigger of a search operation initiated by a user. A *Search Container* is used to specify searching criteria while the search annotation causes the search.

The “=>looked-up widget” notation is used to look-up and assumes the looked-up widget has a unique name. A data input widget that contains the “=>looked-up widget” notation to specify a search criterion is called a *Look-up widget*. For example, “=>login” text in the “Login” *Look-up widget* in the “Administrator sign-in” *container* is used to search for a login value in the “login” field of a database table. That is, a *looked-up widget* is used to specify the data for a search criterion. This specification is used to define the search behaviour. In other words, at run time the data entered in the login and password text inputs in “Administrator sign-in” is used to search against the login and password fields in a database table. It is important for the mock-up designer to remember that though this discussion explains how the data from widgets is related to database concepts to provide insights to the thought process, they do not need to perceive the visual model in terms of database concepts during the modelling process. Rather all “=>looked-up widget” references are strictly perceived in terms of data input widgets (*DFYWs*). The next section discusses these features in more detail for managing search operations in SME applications.

4.2.2 Mock-up segment for search management

Searching of entities is yet another frequent operation in web applications. Continuing with the quest to answer RQ1.2 regarding the features of mock-up language to fully express the requirements, this sub-section discusses searching and

management of search results in detail. The previous sub-section briefly introduced the usage of a *Search Container* to specify search criteria and the specification of a “**search**” keyword to capture the trigger of a user-initiated search operation. Additional features of the language are introduced in this section for search related operations. The following features of the visual language are introduced in this section:

- Usage of a type of *Container* called *Search Result Container* to manage search results
- Usage of a type of widget called *Data View Widget* to display a database field value
- Usage of type of *Container* called *Data View Container* within a *Search Result Container* to display search result
- Usage of “**previous**” and “**next**” annotations on navigation widgets within *Search Result Containers* for traversal of search result sets
- Usage of the “**delete**” annotation on navigation widget within *Search Result Container* to trigger a delete operation
- How to separate traversal of search results from deletion in the *Search Result Container*.
- Specification to pre-populate a combo box
- Specification of *Search Containers* containing *look-up widgets* linked to more than one *Database Field Yielding Containers*

To explain the above-mentioned features of the language, another sub-section of mock-up of the Travel Deal example in Figure 12 is considered. This sub-section is shown in Figure 15 to highlight the essential features of the mock-up for managing search and search results. It principally contains two main *Containers*, namely “Search Deals” and “Available Deals [3]”. The arrows in the figure represent navigations to other sections of the mock-up and out-going arrows with no targets simply indicate that the targets are beyond the scope of this figure.

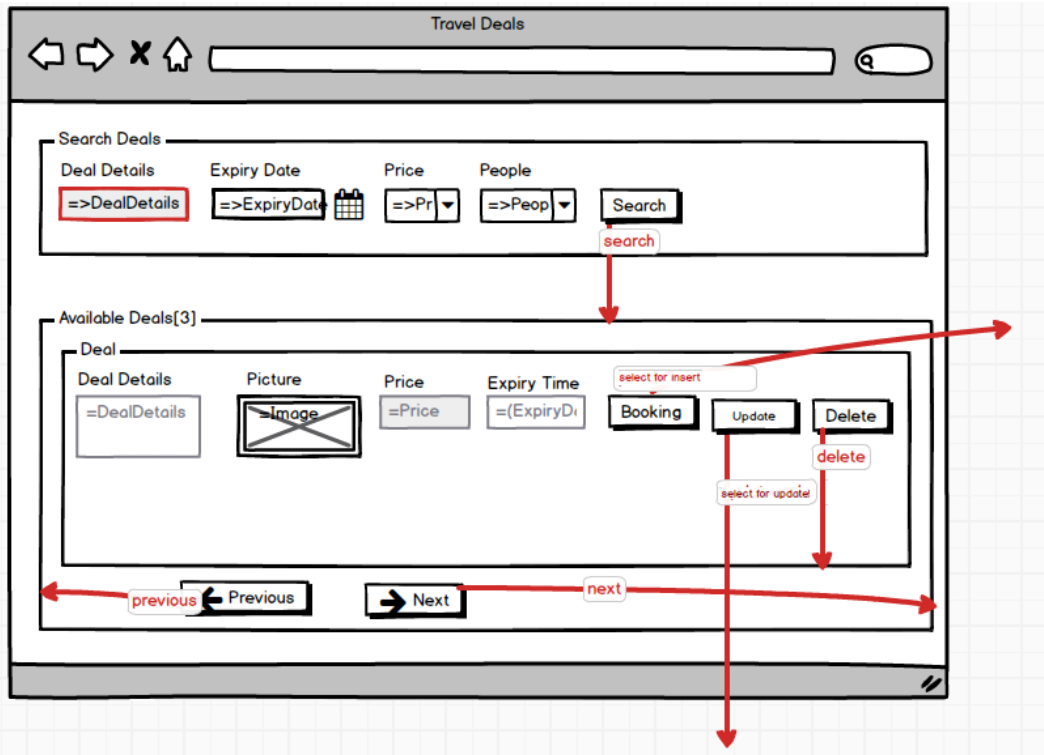


Figure 15: Mock-up segment for managing search and search results in Travel Deal web app

The “Search Deals” *Search Container* consists of data input widgets to specify the data to be searched. It uses the previously discussed “=>looked-up widget” notation to specify the search criteria. However, when the “=>looked-up widget” notation is used within a combo box, the visual language specifies the pre-population of the combo box using the source field of the looked-up widget. For example, the “=>People” notation in the “Search Deals” *Search Container* specifies the “People” combo box needs to be populated with unique user entered data for “People” in the “Add Travel Deal” page in Figure 14. At the database level, this corresponds to the “People” field in a “Travel” database table. In addition, when a user selects a value from the combo box, the selected data will be searched against “People” field in the “Travel” table.

A web page may include a *Search Result Container* to manage the search results. Since a search can result in multiple set of matched entities, a *Search Result Container* contains a nested container to specify the display elements of each result set. This is important because only selected elements in a result set may need to be displayed. The outer container is used to manage traversal through the result sets. In Figure 15

“Deal” is the inner container and “Available Deals [3]” is the outer container. Each container is used for a specific purpose. The “Deal” container uses the “=reference widget” notation to display a field value of a selected result. This notation is used to specify the source of data to be displayed in a data view widget. The visual language terminology for a widget such as “Deal Details” in the “Deal” container is *Data View Widget*. A *Data View Widget* is a widget for displaying existing data from the database. In the example, *Data View Widgets* in the “Deal” container display details such as a textual description of the travel deal, a picture, a price and an expiry date. “Deal” is an example of a *Data View Container*. A *Data View Container* is a *Container* that holds one or more widgets for displaying data where the data to be displayed already exists in storage. The search results may yield many sets of travel deals. In Figure 15 “Available Deals [3]” models all the travel deals yielded by the search. The “[x]” notation is used to specify pagination of deal results. In our example; the visual model specifies three sets of deal results to be displayed at a time. So, a maximum of 3 deals will be displayed at a time in the “Deal” *Data View Container*. If more than 3 deals exist, the “Previous” and “Next” buttons in the “Available Deals[3]” container will be enabled and the “**previous**” and “**next**” annotations on the navigation elements associated with the buttons are used to trigger the service for their navigation. Here “**previous**” and “**next**” are keywords of the language and are contained within “Available Deals [3]” and not within “Deal”. “Available Deals [3]” is an example of a *Search Result Container*. A *Search Result Container* contains the results of a search operation and the “**previous**” and “**next**” annotations on navigation widgets for search result traversal.

The *Data View Container* within a *Search Result Container* may contain at least one among “**select for insert**”, “**select for update**” and “**delete**” annotations on navigation widgets for selecting and further processing of a selected search result set. For example, in Figure 15 when the “Delete” button in “Search Results” *Container* is clicked, the selected “Travel” in search results will be deleted. The “**delete**” annotation on the navigation widget triggers the deletion of the selected results. That is “**delete**” is a keyword in the visual language for deletion of selected elements. Though not shown in Figure 15, further processing of selected travel deal may also

result in an “update” or other activity such as insertion of selected deals using “**select for insert**” annotation on a navigation widget in an insert business transaction. Here an insert business transaction refers to a business transaction resulting in creation of one or more database records. For example, Figure 15 illustrates additional activities related to the selected items in the search result.

The visual mock-up language allows multiple types to be searched. Figure 15 illustrated searching a single *Database Field Yielding Container*, namely the fields from “Travel” *Database Field Yielding Container*. However, in Figure 16 it is assumed that the “First Name” *look-up widget* searches a “Customer” table and the “Travel Deal” *look-up widget* searches a “Travel” table. In this example, each search result set will contain records that match the two looked-up criteria. Further processing of a search result set may affect both “Customer” and “Travel” table. For example, if the “**delete**” operation is activated on the search result set, then the corresponding “Customer” record and “Travel” record will be deleted in one operation. In Figure 16 out-going arrows with no targets simply indicate that the targets are beyond the scope of this figure. This completes the discussion on the language features for search operation. The next section discusses the features for managing insert business transactions in SME applications.

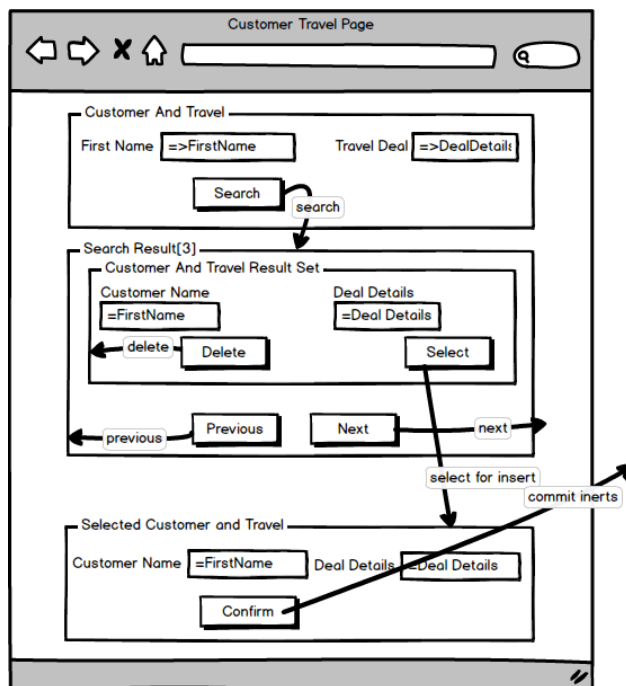


Figure 16: Mock-up segment illustrating searching multiple entity types in a search container

4.2.3 Mock-up segment for insert business transactions

Transaction processing is an essential feature of most business operations. This section discusses how an insert business transaction involving several physical transactions can be visually modelled in the mock-up.

A “transaction” in transactional web applications implies each business transaction is performed as a series of physical transactions. The visual language supports transactions using annotations such as “**select for insert**”, “**temporarily store for insert**” and “**commit inserts**” on navigational widgets. Specifically, this sub-section discusses how these annotations are used in a mock-up to support an insert transaction process. The following features of the visual language will be introduced in this section:

- Nesting of *Database Field Yielding Containers* to specify database relationships among tables
- The use of “**select for insert**” annotation on navigation widgets to select business entities for further processing in a transaction.
- The use of “**temporarily store for insert**” annotation on navigation widgets to temporarily store selected records in a business transaction.
- The use of “**commit inserts**” annotation on navigation widgets to specify completion of an insert business transaction.
- The optional use of “**unique details**” annotations in *Database Field Yielding Containers* to identify a set of *DFYWs* to be treated as a unique composite key in the database.

The previous sub-section provided hints about how selections can be made from search results for further processing using the “**select for insert**” annotation on a navigation widget. Figure 17 illustrates how this can be accomplished. This figure is a sub-part of Figure 12 representing the Travel Deals example. Here selected “Travel” deals are processed further for booking and reporting. It consists of three pages, namely “Travel Deals” for searching a “Travel” deal (only a partial view of this page is shown on the left-hand side of the figure), “Order Deal” page for linking the selected

“Travel” deal with customer’s details and “Booking Confirmation” to provide a report of what was ordered.

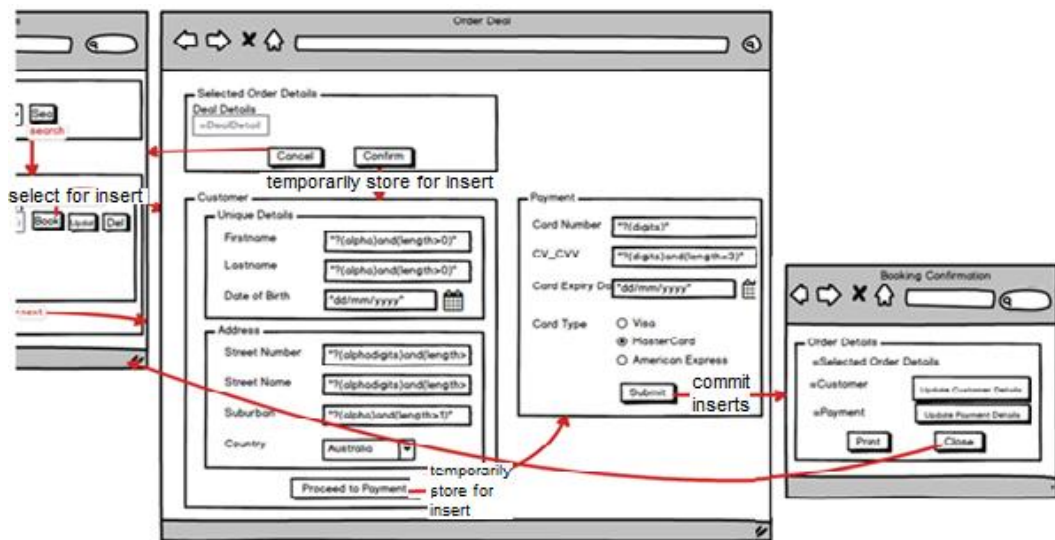


Figure 17: Mock-up segment for insert transaction processing in Travel Deal web app

The processing of the selected deals may either be done on the same “Travel Deals” page or on a different page. Figure 17 illustrates the selections are sent to a “Order Deal” page and displayed in the “Selected Order Details” *Data View Container* using the previously described “=reference widget” notation. The “=DealDetail” text in the *Data View Widget* in “Select Order Details” *Data View Container* is used to display the selected “Travel” deal entity which was originally created using the “Add Travel Deal” page (see sub-section 4.2.1). On clicking the “Confirm” button in the “Selected Order Details” *container*, the “Customer” *Database Field Yielding Container* is enabled to input customer details. The “Customer” *container* contains various *DFYW*’s to input a customer’s first name, last name, date of birth and address. The customer’s first name, last name and date of birth are together treated as unique in the model by using a special container whose text starts with the “Unique Details” key phrase. In Figure 17 “Unique Details” annotated *container* signifies that its *DFYW* elements are to be treated together as a composite key. That is “**unique**” is a keyword in the visual language and is used to specify a composite key, where a composite key is key made of two or more elements and may be used to uniquely identify an entity”. Furthermore, since a customer has an address and since “Address” has its own

details, it is nested with “Customer” as a *Database Field Yielding Container*. The nesting of one “Address” *Database Field Yielding Container* within “Customer” *Database Field Yielding Container* internally identifies a database relationship between a “Customer” table and an “Address” table.

In addition, relationships are also identified through **“temporarily store for insert”** annotations on navigation widgets between containers. For example, the **“temporarily store for insert”** annotation on the navigation link between “Selected Order Details” *Data View Container* and “Customer” *Database Field Yielding Container* in the “Order Deal” page enables the identification of database relationship between “Travel” and “Customer” entities. This is because the “Selected Order Details” *Data View Container* refers to the “Travel” *Database Field Yielding Container*. Similarly, **“temporarily store for insert”** annotation on the navigation widget between “Customer” and “Payment” *Database Field Yielding Containers* identifies yet another relationship. In other words, database relationships, can be established from the source and target containers of **“temporarily store for insert”** links. The source and target of the **“temporarily store for insert”** annotated navigation widget can be *Data View Containers* or *Database Field Yielding Containers*. The target can also be a *Search Container*. While establishing relationships among entities, if a *Data View Container* is involved it is assumed it targets the corresponding *Database Field Yielding Container* it references. Similarly, relationships among entity types are established from *Search Containers* by finding the *Database Field Yielding Containers* from the search criteria. Further details of identification of database relationships are discussed in Chapter 5.

The discussion on the Travel Deal example indicates that the “Order Details” mock-up contains the necessary information to capture values for a database insert transaction dealing with the following types of entities due to the **“temporarily store for insert”** annotations on navigation widgets:

a) “Travel” b) “Customer” c) “Payment”

In addition, the nesting of “Customer” and “Address” containers identify a “Customer-Address” relationship.

The visual language specifies the completion of an insert business transaction by using a navigation widget with the **“commit inserts”** annotation. In the Travel Deal example, the **“commit inserts”** service is triggered when the user clicks the “Submit” button in the Payment *DFY container* to confirm the order. This means that the “Customer” and “Address” record for example, should will not be committed to the database table when the “Proceed to Payment” button is clicked rather it is committed to the database when the button linking the **“commit insert”** annotated navigation widget is clicked. Hence to visually separate the actual insertion from the temporary storage of the various records **the “temporarily store for insert”** annotation is used during the intermediate navigations. This feature is especially useful if the **“commit inserts”** is invoked after traversing a series of pages. Furthermore, generally, after a transaction is completed, most business applications are expected to display a summary of the transaction. The mock-up language displays the result using a *Report View Container*, which is discussed in the following section.

4.2.4 Mock-up segment for managing report generation

Generation of reports is another essential feature of SME applications. This section discusses how report generation may be modelled in the visual mock-up, in the ongoing quest to answer RQ1.2 on the expressive power of the visual language.

A report represents a summary of a transaction. The mock-up language facilitates this by using short cut notation to reduce the time required for designing reports. This section discusses the language features for creating report views. The following features of the visual language are introduced in this sub-section:

- *Report View Container* to display a report of a business transaction
- **“=reference widget”** notation to reference a *Database Field Yielding Container* in a *Report View Container*
- **“=reference widget”** notation to reference a *Data View Container* in a *Report View Container*

An example of the mock-up for a report view in the “Travel deals” case is illustrated in Figure 17. Specifically, the “Booking Confirmation Page” is provided to display all the details of an immediately completed transaction. The details to be displayed include the selected travel deals, customer details and payment details. The visual language provides an optional short hand notation to display all the details of a container instead of specifying each data widget in the container. Figure 18 shows the actual view of the corresponding “Booking Confirmation” page illustrated in Figure 17. Observe that the actual view includes all the attributes of the entities as defined in their corresponding *DYFCs*. If the mock-up designer only wants specific attributes to be displayed, then it should be explicitly modelled in the mock-up instead of following the short cut notation. The data in the report is displayed in read only mode but may be updated using the update buttons provided in the report container. In Figure 17 the short cut is specified using “=*reference widget*” notation in a label widget. For example, in the “Order Details” container a label element with the text “=Selected Order Details” will display all details of the selected travel deals in the “Selected Order Details” container (in the “Order Deal” page). Here “Order Details” is an example of a *Report View Container*. A *Report View Container* displays report of data from one or more entities associated in a business transaction in read only mode. A *Report View Container* is used to view data just like a *Data View Container* except that it may be optionally modelled using a short-cut notation, so it can be treated as special type of *Data View Container*. In Figure 17, “=Selected Order Details” is a reference to a *Data View Container* whereas “=Payment” and “=Customer” are references to *DFY Containers*. In other words, a *Report View Container* may have references to either or both *Data View Containers* and *Database Field Yielding Containers*. Furthermore, if a *DFY Container* has nested containers then the actual report view also includes the corresponding nested containers. This is illustrated by the presence of Address *DFYC* within the Customer *DFYC* in Figure 18 though Address is not present in the mock-up shown in Figure 17. This completes the discussion on the language features for report generation. The next section discusses the language features for managing update operations in SME applications.

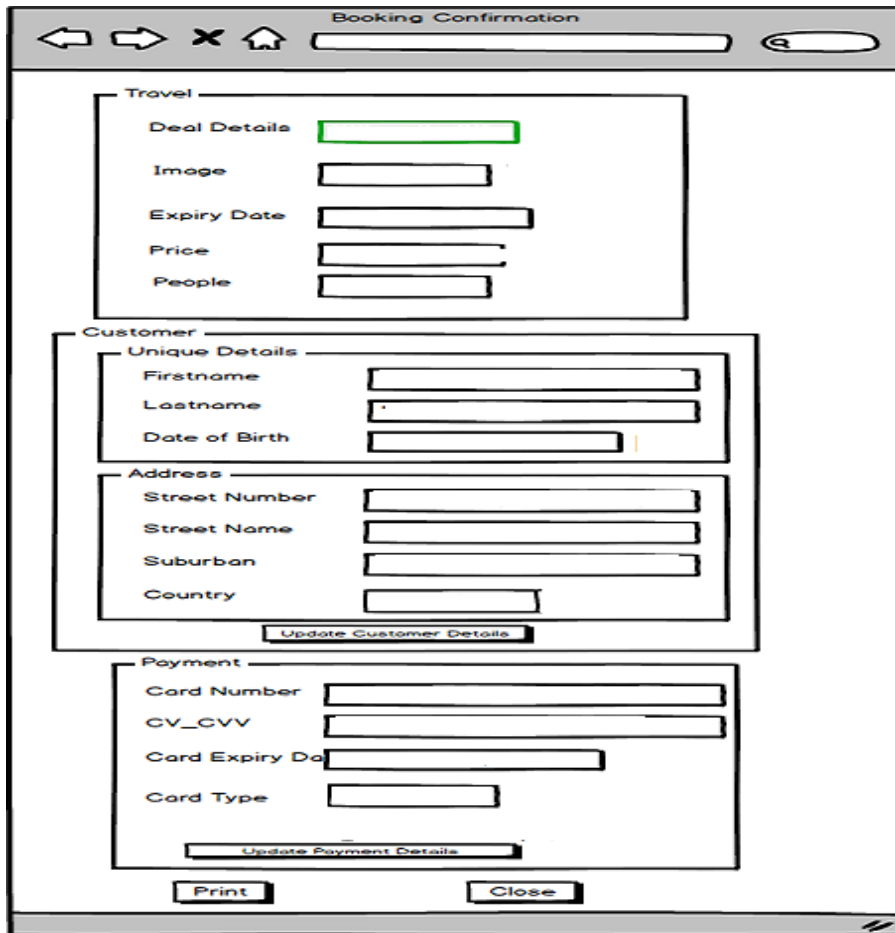


Figure 18: Expanded view of the mock-up segment for booking confirmation page in Travel Deal web app

4.2.5 Mock-up segment for update operation

Updating existing entities is yet another common operation performed in SME applications. This section focuses on how update operations can be visually modelled and represents the last part in the series of discussions on RQ1.2 regarding the expressive power to the language.

The mock-up language supports visual modelling for update of existing business entities. By going through the example in this section the following features of the visual language are introduced:

- How to select data in a Data View Container or a Report View Container using the **“select for update”** annotation on a navigation widget.
- How to specify update of selected data using the **“update”** annotation on a navigation widget.

Figure 19 illustrates how update operations are specified in the visual mock-up in the Travel Deal example. Figure 19 is a sub-part of the complete mock-up given in Figure 12 and is shown in a two-column diagram. The left column illustrates the mock-up section for updating the “Travel Deal” entity and the right column represents another sub-section of the mock-up for updating of “Customer” and “Payment” entities. The left column in Figure 19 illustrates that data for the update operation is selected from a *Data View Container* whereas the right column shown how the data can be selected from a *Report View Container*.

The update operation is specified using the **“select for update”** and **“update”** annotations on navigation widgets. Firstly, updates are always done with respect to data that is presented in *Data View Containers*. The visual language uses the **“select for update”** annotation on a navigation widget to specify a selected set of data items are required to be transferred to the destination container for update. The destination container may either be on the same page or on a different page. The language uses the **“update”** annotation on a navigation widget to specify the data in a *Data Input Container* is to be updated in a database.

In Figure 19 the targets of the **“select for update”** annotated navigation widgets are on different pages. For example, the **“select for update”** navigation widget between the “Travel Deals” page and the “Update Travel Deal” page causes the selected travel deal data sets to be pre-populated in the destination page. The user may edit the data in the “Update Travel Deal” page and click the “OK” button linked to the **“update”** navigation widget to complete the update and navigation. The “Travel” container in the “Update Travel Deal” page is an example of an *Update Container*. An *Update Container* has similar widgets to its corresponding *Database Field Yielding Container* but contains an “update” annotation on a navigation widget instead of a **“commit inserts”** annotation and is pre-populated with data to be updated.

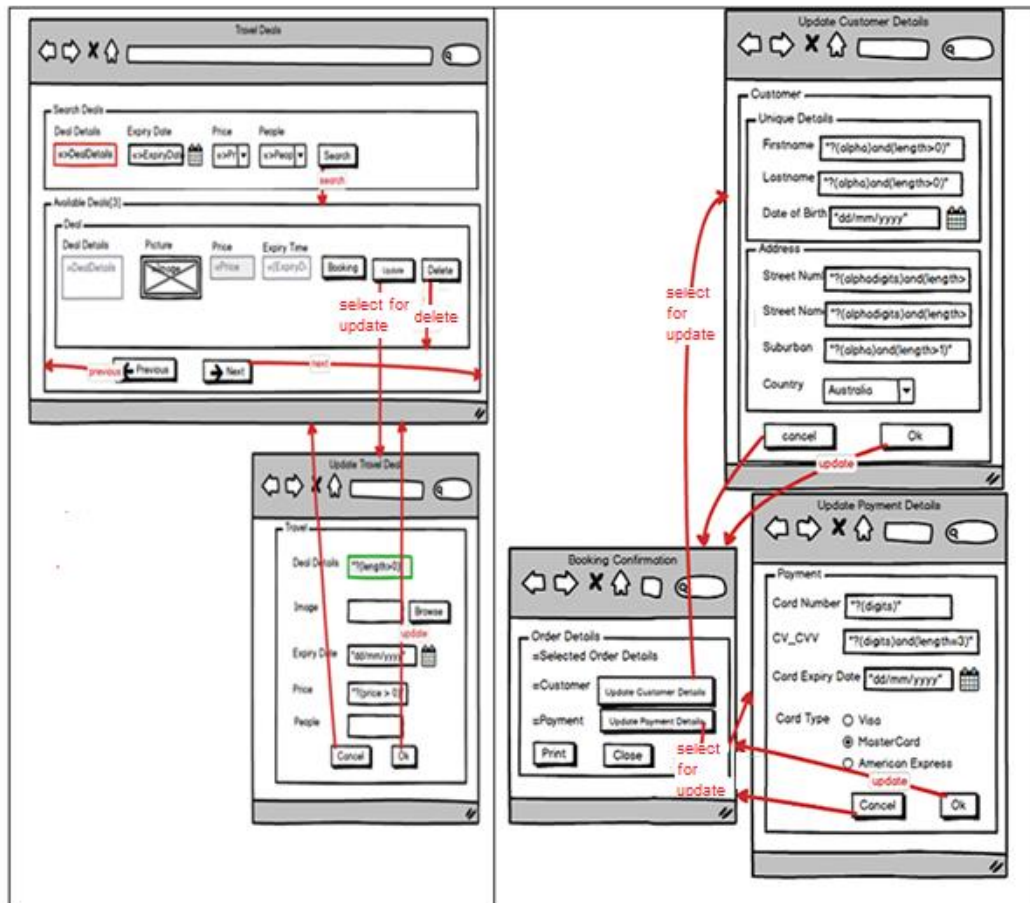


Figure 19: Mock-up segment for managing update operation in Travel Deal web app

“**update**” is a keyword of the visual language that causes the update. Additionally, Figure 19 illustrates how the “Update Customer Details” and “Update Payment Details” containers are every used to update customer and payment details in the “Order Details” Report View Container using the “**update**” annotation on a navigation widget and that the updated details are refreshed in the “Booking Confirmation” page.

Update can also be specified on search results containing more than one entity. Consider another example of a web application for the management of “To do” tasks by assigning them to “users”. Figure 20 contains the mock-up of such a system. The mock-up allows users to create “Todo Task” and “User” entities via *Database Field Yielding Containers*. Figure 20 also illustrates the mock-up permitting assignment of a “Todo Task” to a “User” following a search operation involving both entities. The resulting “User” and “Todo Task” entities that satisfy the search criteria are displayed in the “Search Result [3]” container. The user can then link any “Todo Task” entity

with a “User” entity by following the navigation widget with the **“select for insert”** annotation in the “Search Result[3]” container. Such linked pair of entities can be updated together by following the navigation widget with the **“select for update”** annotation. In Figure 20 such an update operation involving more than one type of entity is performed in the “Update User Todo Page”.

In summary Section 4.2 discussed several features of the visual mock-up language to fully express the structure and behaviour of a web application. The language supports creation of new business entities, searching and traversal through search results, linking existing business entities (e.g. Travel) with new entities (e.g. Customer), linking pre-existing entities (e.g. User and Todo Task in Figure 20), update and deletion of entities and creation of reports in web applications. The various features of the language are put together in the next section to get a meta-model of the language.

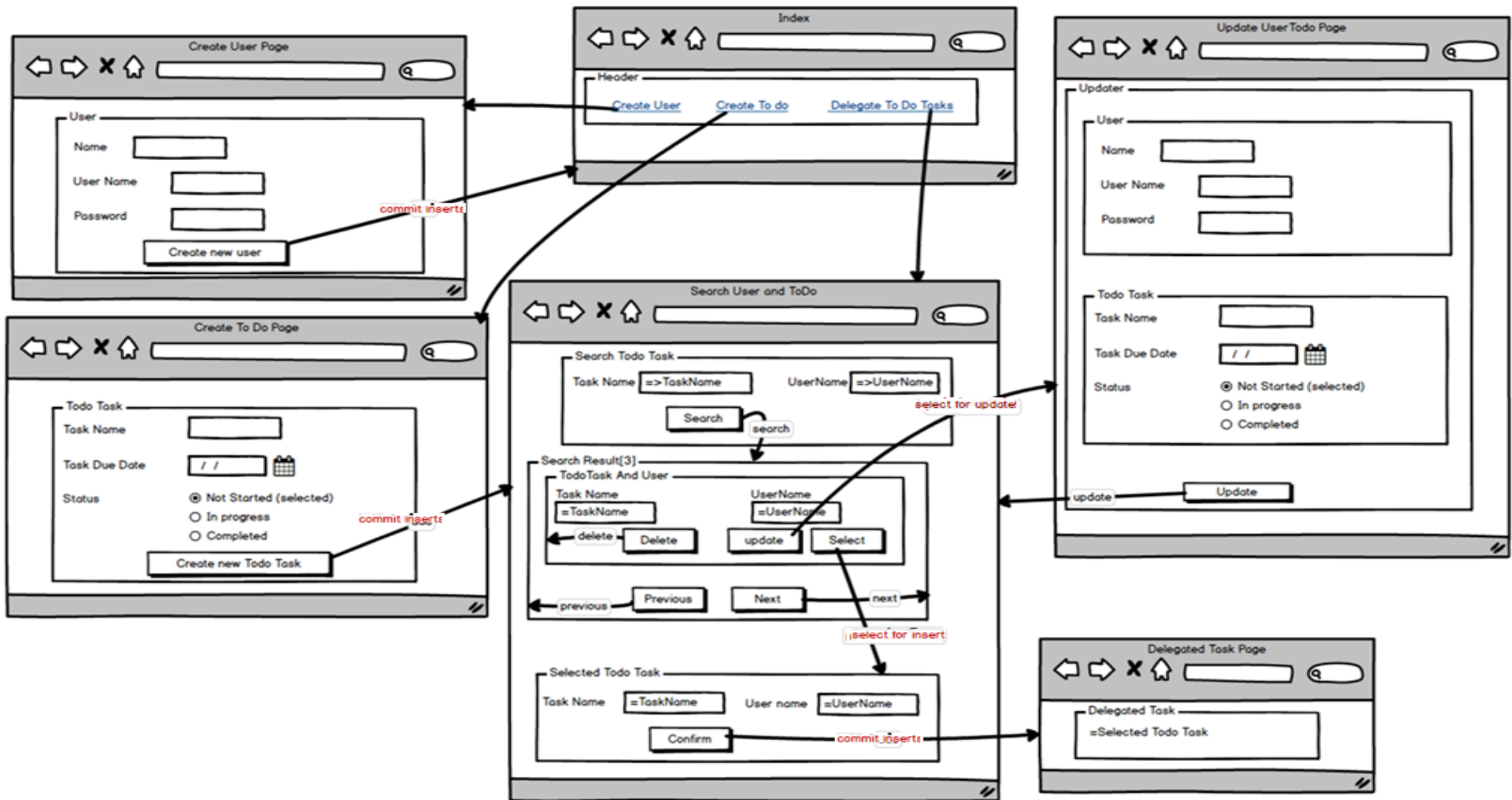


Figure 20: Mock-up to highlight update of multiple entity types in an update container

4.3 Meta-model of the mock-up language

The foundation of the visual mock-up language is based on Valverde and Pastor's (2009) RIA meta-model discussed in Section 2.4.3. The meta-model utilizes few basic widgets and behaviours to capture the visual model of any web application. Hence it makes itself amenable to mock-up modelling by BAs who as discussed in the literature review chapter are traditionally considered to have weak technical skills. A fully functional web application can be auto-generated from such a mock-up if it sufficiently captures the structure and behaviour of the application using Valverde and Pastor's UI meta-model. In order to do this Valverde and Pastor's (2009) UI meta-model has been modified as illustrated in Figure 21. The modifications are mainly to manage the specification of certain distinctions among input widgets and layout widgets, which are necessary for the auto-generation of the application from the mock-up.

A UI mock-up model has a root object that is an instance of the Mock-up class. A Mock-up is composed by one or more web pages, which in turn contains a collection of Structural Element, representing a UI mock-up component (generally referred to as a widget). A structural element can belong to one of the five basic types of widgets identified by Valverde and Pastor (2009) as, Data Input, Data View, Event Service, Navigation, and Layout (Container). Since these are already discussed earlier in Section 2.4.3.1, the discussion in this sub-section will focus on the extensions required for the meta-model to facilitate creation of mock-ups for auto-generation. The extensions are displayed as classes with yellowish-orange background colour in Figure 21. Further details of the meta-model will be discussed with the help of the travel deals case study (see Figure 12) that was introduced in Section 4.2

Navigation. The navigation widget is used to change the point from which the application's UI is perceived by the user. The navigational information is visually modelled using an arrow and represents a transition between two widgets. In the meta-model, this is represented by the Navigation class. Attributes, *source* and *target* respectively represent the source and target widgets during the navigation process

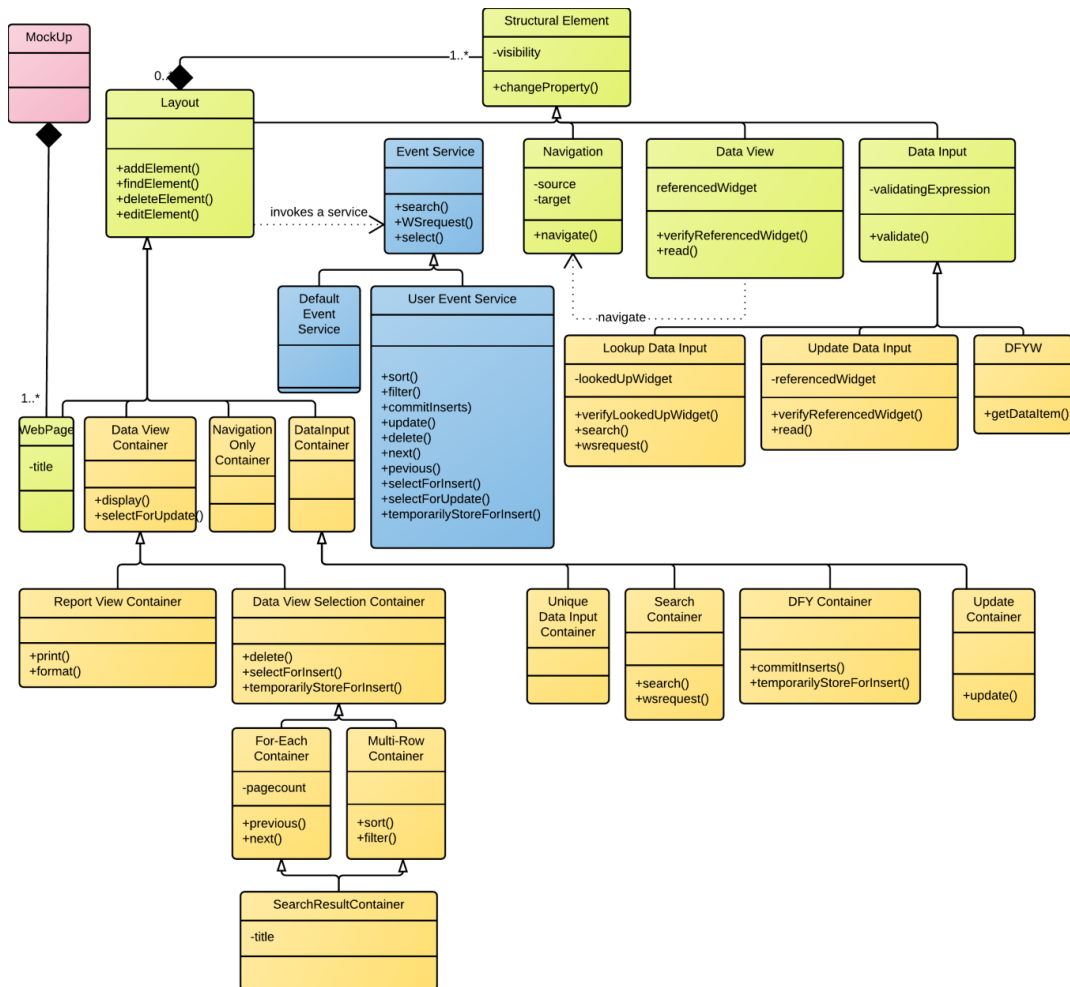


Figure 21: Meta-model of the mock-up language

Event Service. A service widget triggers the invocation of a service action on the triggering of an event. There are two types of service actions - one is triggered by user interaction events and the other by default events such as an on-load event, or a timer event. In the meta-model, these service actions are identified by classes *Default Event Service* and *User Event Service* respectively. A service action is captured visually by annotating a *Navigation* arrow. The target of the arrow can either be the source widget itself or a different widget. If the arrow connects two different widgets, then it also signifies navigation following the service invocation. On the other hand, the annotation of a self-pointing arrow on a widget represents functional invocation without navigation. Services related to self-pointing arrows are generally triggered by default events, while other widget pointing arrows are always triggered by user events. For example an annotation, “**WSRequest**”, of the self-pointing arrow could be specified on the “country” widget in the “Order Deal” page to indicate an on load

web service action to be invoked for the population of the country names whereas the annotation, “**commit inserts**”, on the arrow between the “Submit” button in the “Order Deal” page and the “Booking Confirmation” page models the invocation of a user initiated event service resulting in the committing of the order details to a database, following which it navigates to the “Booking Confirmation” page. A service action may be executed by local methods or by secondary web service invocations. In the visual model, web service invocations are identified by a URL specification beginning with the keyword “**WSRequest**”. Apart from this difference, the visual model does not differentiate between *Default Event Service* actions and *User Event Service* actions. That is, when “**WSRequest**” is present on a navigation widget it indicates the no user action is required to trigger the service. A service widget’s action method may require zero or more contextual arguments to be specified. The contextual information is collected from the source and destination widgets of the arrow or from the container enclosing the source/destination widget of the arrow. For example, in the “Travel Deals” page, the “**search**” arrow between the “SearchDeals” and the “AvailableDeals” container, invokes the search method but it requires three arguments representing the search criteria. This is obtained from the data in the widgets: “Deal Details”, “Expiry Date” and “Price”. In these widgets, the “=>looked-up widget” notation is used to indicate a search criterion. Since these widgets exist in the same container as the service widget, the auto generator has sufficient information regarding the search criteria from the visual mock-up. Further, the search behaviour needs additional information regarding the selection of display data. That is, not all information that matches the search needs to be selected for display. This information is obtained from the widgets in the destination container of the “**search**” arrow. In the destination container, the “=*reference widget*” notation is used to indicate the selection. Further details regarding the “=>looked-up widget” notation and the “=*reference widget*” notation are discussed later in this section.

Data Input. A data input widget enables the user to input data. The input data may need to be validated. Hence some form of validation behaviour can be associated with data input widgets. In the visual model, a data input widget is represented by an

appropriate input widget with optional specifications for validations and formatting as explained in Section 4.2.1.

Lookup Data Input. A specific kind of data input widget whose input data is used to look-up against existing data is called *Lookup Data Input* widget. For example, during a search operation or a sign-in operation, the data input in a *Data Input* widget is not used for storage in a database, rather it is used to look-up against existing data. In the visual model, lookup is specified using the “=>looked-up widget” notation. The “=>looked-up widget” notation assumes the looked-up widget has a unique name. If not, a fully qualified name should be used. A fully qualified name can be expressed using a hierarchical path specification of the form: “widget w in a container c in a page p”. If a look-up widget is a multi-value widget such as “Price” in the “Search Deals” container of Figure 15, then the look-up notation is also used by a *service* action method to get the criteria for the on-load population of the widget. For example, the “=>Price” notation within the “price” drop down box is used to specify two types of searches. One, to specify the search criteria for the population of the widget on load and two, specify a search criterion for the button click event by the user. Hence in the meta-model, a *Lookup Data Input* widget is shown to have **search** and **WSRequest** methods. These methods will invoke a corresponding method from a *Service* class. From the above discussion, it is evident that dependency can exist between a lookup data input widget and looked up data input widget and that any change in the specification of the looked-up data input widget will have a referential effect on the corresponding lookup widget. Identification of such dependencies is important during the management of the evolution of the application. Please read Section 4.2.2 for further illustrations of how the *Lookup Data Input* widget is used.

Database Field Yielding Widget (DFYW). A *DFYW* is a data input widget whose data needs to be persisted in a database table. That is, the data input in a *DFYW* is essentially ends up as a value of a database field. Please refer to Section 4.2.1 for further illustrations of how this is used with respect to the “Travel Deals” example.

Update Data Input. This is a *Data Input* widget that is used to update existing data in a database. In the visual model, *update data input* widgets are always found in a

container (layout) which contains the source of an “update” navigation widget. At the meta-model level, this widget is represented by the *Update Data Input* class, which is a sub-class of the *Data Input* class. Since the *Update Data Input* class needs to first read the data before an update, this class has a read method that behaves like the read method of the Data View widget. Please read Section 4.2.5 to learn how to use *update data input* widgets in a visual mock-up.

Data View. A *data view* widget is used to display existing data by reading it from a database. The term “existing” refers to data that was previously input using a data input widget. Thus, the model of the data view widget should also specify the corresponding *data input* widget. In the visual model, the “=*reference widget*” notation is used to specify the source of data to be displayed in a data view widget. For example, in the “Selected Order Details” container of the “Order Deal page” in Figure 17, “=DealDetails” specifies that the *data view* widget should display data by referencing a “Deal Details” *data input* widget. The reference may either be to a *DFYW* or a *DFY Container*, or a *Data View Container*. A *Data View* widget specification results in a read operation from storage when the container in which the data is to be displayed is enabled. The meta-model includes a *read* method in the *DataView* class. A *data view* widget may also exhibit navigation behaviour. Though the case study does not include an example of this, in some cases navigation behaviour is also required to be specified for a displayed data item. For example, a page may contain a navigation link for each item in a list of available travel deals. So, each item in the list will be a *Data View* widget as well as a *Navigation* widget. In the visual model, an arrow is used along with the “=*reference widget*” notation to specify a *data view* widget that is also a *navigation* widget.

Layout Widget. A layout widget is a container that is used to organize a group of logically related widgets. The usage of containers eases the specification of the organization of the UI. As discussed in Section 4.2 the mock-up language distinguishes several types of layouts (containers). These are discussed next.

Data View Container. This widget is a type of *Layout* widget that contains at least one *data view* widget. A *Data View Container* is a special type of *layout* container to

predominantly display data using *Data View Widgets*. Such a container helps the mock-up designer to organize a group of related *data view widgets* together. Generally, the visual model will use a *data view container* to specify the *Service* widget action to be performed on all the data view widgets it. For example, the visual model in Figure 12 uses the “**select for update**” service in the “Deal” container in the “Travel Deals” page to specify the list of data items to be selected for an update operation in the future. Hence, in the meta-model in Figure 21 a *Data View Container* is shown to have *display* and “**select for update**” methods. These methods will then invoke the corresponding methods from the *Service* class. Specialized versions of *Data View Container* can exhibit other optional actions such as “**delete**”, “**select for insert**”, “**temporarily store for insert**”, “**format**” data etc. These are considered in several sub-classes of *Data View Containers*: *Report View Container*, *Data View Selection Container*, *For-Each Container*, *Multi-Row Container* and *Search Result Container*.

Report View Container. A special type of data view container optionally used for displaying data in a report format. Hence the *Report View Container* class in the meta-model has a *format* method. The “Order Details” container in the “Booking Confirmation” page in Figure 12 is an example of a *Report View Container*. Section 4.2.4 contains further details about the usage of *Report View Containers*.

Data View Selection Container. This is an abstract *data view container* with optional actions for “**delete**” and/or “**select for insert**” and/or “**temporarily store for insert**”. Abstract means that such a container may only be instantiated by its sub-classes. The “Deal” container in Figure 12 is an example of an instance of such a container. The “**delete**” action triggers deletion of the selected item from the *Data View Container* and from the database. For example, the “**delete**” annotation on the navigation widget in Figure 12 deletes the selected “Travel” entity from the “Deal” container. The “**select for insert**” and “**temporarily store for insert**” are used to specify that the operation is to be performed as a part of a series of operations in an insert business transaction. The “**select for insert**” operation is used for selecting and temporarily storing a data set from a *Data View Container* whereas the “**temporarily store for insert**” annotation stores and links previously selected data from a *Data View*

Container or from a *Database Field Yielding Container*. This distinction can be explained with respect to Figure 12 where an existing “Travel” deal is linked with a new “Customer” for ordering a deal as part of an insert business transaction. When a button linked to “**select for insert**” annotated navigation widget is clicked an existing “Travel” deal is selected from a search result and put in temporary storage. This data may be used in several different contexts. The mock-up in Figure 12 specifies a context in which new-details of “Customer” and “Payment” are to be captured and linked with the selected “Travel” data as a part of a series of operations in an insert business transaction. Here “**temporarily store for insert**” operation is used to temporarily store and link previously selected data or new data in an insert business transaction and is distinct from “**select for insert**” which is used only used to select and store data from a *Data View Container* in a *Search Result Container*. The end of an insert operation occurs when a “**commit inserts**” operation on a navigation widget is encountered. Thus “**temporarily store for insert**” is used to store linked data whereas “**select for insert**” is used to store a selection without linking in an insert business transaction.

For-Each Container. This is a special type of *Data View Container* that is used to create a repeated set of data view containers, for pagination effect on the client side. For example, in the “Available Deals” container of the “Search Deal” page in Figure 12 the presence of the digit 3 in square brackets next to the “Available Deals” container specifies three sets of “Deal” containers to be displayed at a time. That is the page size is 3, for displaying three sets of records at a time. Correspondingly the meta-model of a *For-Each Container* is shown to have previous and next methods. These methods will then invoke the corresponding methods from the *Service* class to traverse back and forth among the list of selected data items. For example, when the “Previous” or “Next” button in the “Available Deals” container is clicked, it will invoke the corresponding methods in the *Service* class.

Multi-Row Container. This is yet another special type of *Data View Container* for presenting repetitive information generally as rows in a HTML table. Each row of the table represents a repetition of a set of widgets. A special functionality of such a widget is to be able to request the *Service* widget to sort or filter the data in container.

Data Input Container. This is a generic class to represent a collection of *Data Input* widgets. This type of container normally also contains a *Service* widget to invoke a service action such as “**search**”, “**WSRequest**”, “**commit inserts**” or an update.

DFY Container. This type of layout is used to group a set of related *DFYWs* to specify the definition of a new business entity, which consequently results in a new data entity to be stored in a database. A *DFY Container* represents the specification of a container for a logical grouping of *DFYWs*. Since a new instance of the business entity is created from such a container, in the meta-model the annotation, “**temporarily store for inserts**” or “**commit inserts**”, is associated with the container to specify that the *Service* widget’s corresponding method will be invoked by a service widget in the container. As an example, in Figure 12 the “**commit inserts**” arrow linked to the “Add Travel Deal” page specifies that newly created “Travel” entities will be stored in the database with details corresponding to the *DFYWs* in the *DFY Container*.

Unique Input Data Container (Unique Container). This is a special type of *Data Input Container* to specify that the data from two or more *Data Input* widgets should be treated as unique. Such a container can be used to uniquely identify a business entity. At the data model level, a unique input data widget container helps in identifying compound primary keys in a database table. For example, while ordering a travel deal a requirement can be to uniquely identify a customer using the first name, last name and the date of birth. The *Unique Details* container of the “Order Deal” page in Figure 12 illustrates how this is specified in the visual model. The meta-model definition for such a container is simply to prefix any *Data Input Container* with the keyword unique. No “unique” *service* behaviour is associated with such a container. This is because uniqueness is only a design time feature that can be identified from the name of the container.

Update Data Input Container (Update Container). This is a *Data Input Container* that can be used to specify the data in its *Data Input Widgets* should be updated. That is existing instances of one or more business entity is updated in such a container. Hence in the meta-model the annotation, “**update**”, is associated with the container to specify the *Service* widget’s update method will be invoked by a service widget in

such a container. As an example, in Figure 12 the “**update**” annotation on navigation widget from the “Update Travel Deal” page specifies an existing “Travel Deal” entity’s fields will be updated in the database with details corresponding to the *Update Data Input* widgets in the container.

Search Container. A Search Container is a *Data Input Container* to contain a list of *Lookup Data Input* widgets. For example, in the “Travel Deal” page in Figure 12, the “Search Deals” container encapsulates the *Lookup Data Input* widgets for the search criteria. Data from the *Search Container* is used by the “search” *Service* widget where the “search” service widget is represented by the “**search**” annotation on a navigation widget. Hence, in the meta-model, the *Search Container* class has a search method what will invoke the search method in the *Service* widget class.

Search Result Container. A *Search Result Container* is used to contain the results of a search operation. In the meta-model, it is shown as a container that inherits its properties and operations from the *Data View Container*, *Data View Selection Container*, *For-Each Container* and *Multi-Row Container* since it can exhibit all the of the behaviours associated with the inherited classes. Please refer to Section 4.2 for illustrations of its usage.

Navigation Only Container. A Navigation Only Container is a container that does not contain *Data View Widgets* or *Data Input Widgets* but contains at least one navigation widget. For example, the “Admin Operations” container in the “Deal Management” page in Figure 12 represents a *Navigation Only Container*. *Navigation only containers* are generally used as headers or footers in web applications to provide high-level navigation visibility. The next section provides a summary of the features of the visual mock-up language.

4.4 Summary of the mock-up language specifications

A visual mock-up is made of one or more web pages where each web page is considered to have unique name and the visual language is case insensitive. Further each web page is made of one or more containers and associated with one or more

operations. Table 5 and Table 6 provide a summary of the containers and annotations on navigation widgets that are used to specify the behaviour of the application.

Table 5: Summary of definitions of container types

Container Type	Container Definition
Database Field Yielding Container (DFYC)	A data input container that groups a set of DFYWs, the data from which can potentially be inserted as a record in a database table
“Unique” Container	A nested container within a DFYC to specify a composite key
Search Container	A container that groups a set of look-up widgets for entering search criteria using the “=>looked-up widget” notation
Search Result Container	A container that groups a set of data view widgets to manage the traversing through the result of a search operation using “previous” and “next” annotated navigation widgets
Data View Container (DVC)	A container that groups <i>Data View Widgets (DVW)</i> , where a DVW is a widget that displays existing datum in read only mode
Update Container	A container that groups pre-initialized data input widgets for potential update
Report (View) Container	A DVC to display a report of data collected from one or more entities associated in a business transaction

Table 6: Summary of annotations for behavioural specifications

Annotation (Behaviour)Type	Annotation (Behaviour) Definition
“search”	A construct to trigger the search behaviour
“previous”, “next”	Constructs to trigger the behaviour of traversal of research results
“select for insert”	A construct to trigger temporarily storage of a selected data set from a Search Result Container (without linking with another data set) as a part of an insert business transaction
“temporarily store for insert”	A construct to trigger linking of data sets in two containers and to temporarily store them as part of an insert business transaction
“commit inserts”	A construct to trigger commitment of one or more temporarily stored data sets onto one or more database tables, at the end of an insert business transaction
“select for update”	A construct to trigger temporarily storage of a selected data set for potential use in an Update Container
“update”	A construct to trigger update of data in an Update Container on to one or more tables in a database
“delete”	A construct to trigger deletion of a selected data set from a Search Result Container and from one or more tables in a database
“WSRequest”	A construct to trigger a web service invocation

The next section discusses how a suitable mock-up editing tool is found to exploit the features of the visual mock-up language.

4.5 Features of the tool for easy integration with the language

The tool for editing the mock-up using the visual language features discussed in Section 4.2 should not be cognitively challenging for BAs. The tool should allow easy integration with features of the language. Some of the features of the tool that support integration with the language are discussed in this section. This discussion corresponds to block 1c in the research plan shown in Figure 11.

Easy integration the tool should help a BA to perform activities to: easily select or deselect a widget, copy, paste, undo, annotate element for learn ability and import or export of the mock-up. As discussed during the literature review in Section 2.4.5, several types of tools are researched for visual mock-ups. However most of them are not suited for BAs since they are cognitively challenging to operate. Please refer to

Table 1 in Section 2.4.5 to learn about some of the challenges. Hence some popular wireframing tools were analysed to find common features make them easy to use by non-technical users such as BAs. Specifically, five tools were studied, namely, Balsamiq, Mockplus⁴, Axure⁵, Just in Mind⁶ and UXPin⁷. By studying these tools some of the common features of the tool that potentially make them amenable for easy integration with the language were identified. These features were found to be:

- Easy to search for widgets for quick insertion
- Drag and drop of widgets on to a canvas.

⁴ <https://www.mockplus.com/features>

⁵ <https://www.axure.com/>

⁶ <http://www.justinmind.com/>

⁷ <https://www.uxpin.com/>

- Easy to change the properties of a widget. For example, it should be easy to send a widget to a background or to group a set of widgets.
- “Copy” and “paste” features to reduce duplication effort.
- “Erase” and “undo” features.
- Easy to customise and annotate the widgets as desired.
- Easy to zoom in and out to focus on segments of the mock-up.
- Auto-save feature to prevent accidental loss of the mock-up.
- Features for exporting or importing mock-ups.
- A toolbox of readymade widgets for most commonly used web apps.
- Generate output in form of XML or JavaScript Object Notation (JSON) code from the visual mock-up. That is the output of the visual mock up is not HTML or any other proprietary code. JSON is a data format that uses human-readable text to transmit data objects consisting of attribute–value pairs and array data types. XML and JSON are popularly used in the industrial applications and several pre-defined software libraries exist to process them.

To choose a suitable tool, mock-ups like the Travel Deal mock-up shown in Figure 12 were created using each of the above-mentioned tools. Though all the five tools provided good support for most of the common features, Balsamiq was found to be the most effortless to use and it had support for all the features. Hence it was chosen as a suitable tool for integration with the language features discussed in Section 4.2. The next chapter discusses how a visual mock-up created using Balsamiq as a mock-up editing tool and the language features discussed in this chapter, can be used to auto-generate a SME application.

5 DESIGN OF AN AUTO GENERATOR OF WEB APPS FROM VISUAL MOCK-UPS

Design of an artifact as an innovative solution to a wicked problem is the main activity of DSR in IS. This chapter discusses the design activities for an auto-generator of a SME web application from a visual mock-up. That is, it provides answers to research question 2, as a part of the DSR in IS research plan activities envisaged largely in block 2b (auto-generation of database schema) and 2c (auto-generation of client and server-side components) in Figure 11. Other design related activities shown in Figure 11 are block 2a (design of mock-up), block 2d (auto-generation), block 2e (functional testing) and block 2f (design optimizations). These activities are performed routinely during the internal design process. It may be noted from Section 3.1 that research question 2 is made of three sub-questions respectively dealing with auto-generation of: the database structure (RQ2.1), the client side and server-side components of RIA (RQ2.2), and the database logic (RQ2.3). Correspondingly this chapter is organized as follows. Section 5.1 discusses the algorithms for the database schema auto-generation thereby providing answers to RQ2.1. Section 5.2 to Section 5.6 provides answers to RQ2.2 and RQ2.3. Section 5.7 provides a general discussion on client-server communication structure in the auto-generated application, highlighting that the generated application has quick-response times and low-design complexity due to the chosen architecture. Section 5.8 provides a summary of all the auto-generating components required for the various pages of the Travel Deals case study. Finally, Section 5.9 discusses how a trial evaluation of auto-generating tool was performed as a part of the internal design process. It may be noted that five sections (Section 5.2 to Section 5.6) are devoted to RQ2.2 and RQ2.3 because they deal with the five common behaviours of SME applications identified in Section 4.1 in the previous chapter, namely, **“search”**, **“insert”**, **“report”** generation, **“update”** and **“delete”**. Specifically, Section 5.2 discusses auto-generation component required for **“search”** operations, Section 5.3 considers **“insert”** operations, Section 5.4 deliberates on

“report” generation, Section 5.5 discusses “update” operation and the focus of Section 5.6 is from a “delete” operation perspective.

5.1 Algorithms for database schema generation from mock-up

The thought process behind the derivations of algorithms for database schema or the Entity-Relationship (E-R) model generation from a visual mock-up is discussed in this section. Detailed versions of the corresponding algorithms are given in the appendix.

From the meta-model in Figure 21 it is evident that the UI of a web application is composed as pages with groupings of conceptually related widgets within containers. The process of deriving the E-R model is driven by certain inference rules, which principally deal with the identification of groupings of data widgets within containers and the relationships among such containers. The containers help in finding entity types and the relationships between the entities in the E-R model are derived from the nesting of containers or from navigational behaviour between containers. A summary of the diverse types of widgets, containers and behaviours is provided in Section 4.4 in the previous chapter. Specifically, from the previous chapter it is known that Database Field Yielding Containers (DFYCs) identify entity types and DFYCs principally contain data input widgets in the form of *Database Field Yielding Widgets* (DFYW). So, the derivation of E-R models from a UI mock-up involves identifying: *DFYW*, *DFYCs*, and the relationships between these DFYCs. Consequently, this section is organized as follows. Sub-section 5.1.1 discusses the algorithm for identifying DFYWs in a mock-up, sub-section 5.1.2 discusses the algorithm for identifying DFYCs, sub-section 5.1.3 discusses how to identify entity-relationships among DFYCs and sub-section 5.1.4 provides a summary of how the auto-generation algorithms are applied to the Travel Deal example.

5.1.1 Identifying DFYWs

From the discussion in Section 4.2.1 it is known that a *DFYW* is a data input widget that results in the storage of its input data in a field of a database table. Hence, an E-

R generating component of the auto-generator needs to identify whether any data input widget in the mock-up is a *DFYW*. From the meta-model in Figure 21 it is known there can be three types of data input widgets, namely *Look-up*, *DFYW*, and *Update Data Input*. Thus, the algorithm to find whether a widget is a *DFYW* simply requires verifying whether a *Data Input widget* is neither a *Look-up* nor an *Update Data Input* widget. From the discussion in Section 4.2.2 it is known, a *Look-up widget* is identifiable by the existence of the “=>looked-up widget” notation. Similarly, from Section 4.2.5 it is known than update widgets are identifiable by them being housed in the same container as that of the source of an “**update**” annotated navigation link. Hence it is easy to identify and eliminate these two types of data input widgets on encountering them in the mock-up in the quest to find *DFYWs*. The detailed version of this algorithm can be found in Appendix 1.1.

5.1.2 Identifying *DFY* Containers

DFY Containers are strongly related to corresponding database tables. From the discussion in Section 4.2.1 it is known that groupings of *DFYWs* help in identifying a database table, and such a grouping is termed as a *DFY Container*. This sub-section discusses how to identify *DFY* containers.

Two essential features of *DFY* containers are that they are *Data Input Containers* and are sources of either a “**commit inserts**” or a “**temporarily store for insert**” annotated navigation widget. Here, source refers to the widget associated with the beginning of the navigation widget. The Travel Deal example is used to explain how the above-mentioned features can be used to identify *DFY Containers*.

The *Data Input Containers* in the Travel Deal example are: “Search Deals” in “Travel Deals” page, “Administrator” in “Login” page, “Travel” in “Add Travel Deal” page, “Customer”, “Unique Details”, “Address” in “Update Customer Details” page, “Customer”, “Unique Details”, “Address” in “Order Deal” page, “Payment” in “Unique Payment Details” page and “Administrator” in “Add Administrator” page. Using the logic in the previous paragraph to identify *DFY Containers* among the *Data Input Containers*, the “Search Deals” container in the “Travel Deals” page and the

“Administrator” container in the “Login” page are not *DFY Container* candidates even though they have data input widgets because they are not sources of either a **“commit inserts”** or a **“temporarily store for insert”** navigation widget. Similarly, the “Unique Details” container in the “Order Deal” page is not considered because as seen from Figure 21, it is a *Unique Data Input Container* that is used to represent composite primary key fields within a data base table. Similarly, the containers in the “Update Customer Details” page and in “Update Payment Details” page are not considered because they are not sources of either **commit inserts”** or a **“temporarily store for insert”** navigation widget. The rest of the containers in the case study are ignored because they are not *data input containers*.

Moreover, since a visual model may contain nested *DFY containers* (in Figure 12 for example, “Address” DFYC is nested within “Customer” DFYC), the E-R generator should be able to derive potential database tables from such nested containers too. Hence “Address” is identified as a *DFY Container* within the “Customer” *DFY Container* in the “Order Deal” page.

By following the above algorithm, the following database tables are identified from the Travel Deal example: A “Travel” table from the “add travel deal” page, a “Customer” table, an “Address” table, and a “Payment” table from the “Order Deals” page. The detailed version of the algorithm to identify *DFY Containers* in a visual mock-up is available in Appendix 1.2.

In summary, a container in the visual model is identified as a *DFY container* if it is not a unique data input container and contains at least one *DFYW* and is the source of either a **“commit inserts”** or a **“temporarily store for insert”** navigation widget or is a *Data Input Container* nested within a *DFY Container*. The next section discusses how relationships among database tables can be identified from the mock-up.

5.1.3 Identifying Entity Relationships (E-Rs) among database tables

Once the database tables are identified, the second important aspect is the derivation of entity-relationships (E-Rs) among the tables from the mock-up model. E-Rs can be identified from the following: nested of DFYCs, look-up widgets in Search

Containers and “temporarily store for insert” annotated navigation widget linkage among containers. These are discussed in the following sub-sections.

5.1.3.1 Identifying E-Rs from nested DFY Containers

In the real (business) world, an implicit way of visually illustrating “has-a” relationships among business entities is to show them as nested entities. For example, the Invoice template in Figure 22 below, can be thought of being composed of the four nested components, which are obviously related to an invoice.

DESCRIPTION	TAXED	AMOUNT
[Service Fee]		230,00
[[Labor: 5 hours at \$75/hr]	X	375,00
[Parts]		345,00

Subtotal	\$	950,00
Taxable	\$	345,00
Tax rate		6,250%
Tax due	\$	21,56
Other	\$	-
TOTAL Due	\$	971,56

Figure 22: Perceiving a UI as a group of containers

In the figure, the four components have been represented by the enclosing bands (in red on colour prints) for: the header, billing address, order items description, and total payment details. The visual model captures such relationships in the form of nested DFY containers. Since a *DFY Container* corresponds to a database table, nested *DFY Containers* imply relationships among the corresponding database tables. With respect to the Travel Deal example, the Customer container in the Order Deal page includes an Address container and a Unique Details container. The discussions in Section 4 indicates that Unique Details container is ignored while considering tables because it represents a composite data field within the Customer Details container. So only the Address container is considered as being nested within the Customer Details *DFY Container*. Hence a relationship can be established between a *Customer* table and the *Addresses* table (see Figure 23). The detailed version of this algorithm is available in Appendix 1.3.

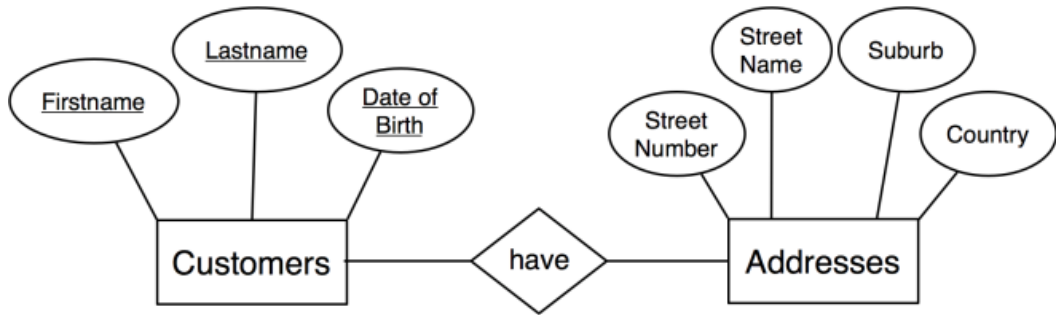


Figure 23: E-R Model due to nested DFY Containers

5.1.3.2 Identifying E-Rs from Search Containers

Entity-Relationships can be identified from look-up widgets in Search Containers. As discussed in the mock-up language specification for search activity in Section 4.2.2 and in the meta-model in Section 4.3, a *Search Container* may contain look-up widgets from several different *Database Field Yielding Containers*. An example of this was provided in Figure 15 in Section 4.2.2 where the “Customer And Travel” *Search Container* has the “=>FirstName” look up widget referencing the “Customer” *Database Field Yielding Container* and the “=>DealDetails” look up widget referencing the “Travel” *Database Field Yielding Container*. The prevalence of such search criteria from multiple types of *Database Field Yielding Containers* in a mock-up indicates the existence of relationship between their corresponding database tables. Hence from Figure 15, it is evident that an E-R could be established between “Customer” and “Travel” database tables which is illustrated in Figure 24.

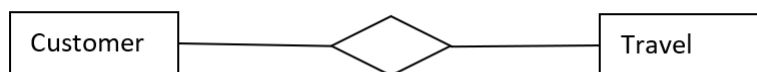


Figure 24: Customer-Travel relationship from criteria in Search Container

The above discussion yields a generic algorithm for identifying relationships from search containers: Find all unique *DFY Containers* referenced by *look-up widgets* in the *Search Container*. If more than one *DFY Containers* are found, create relationships among their associated database tables. For example, if $T_1, T_2 \dots T_n$ are the tables associated with *DFY Containers* referenced by the *look-up widgets* in a *Search*

Container, then create a relationship table: T_{lookup} equal to a set of foreign keys from $T_1, T_2 \dots T_n$ where a foreign key is a reference key which uniquely identifies a record in another database table. That is the set of foreign keys in T_{lookup} refer to primary keys in $T_1, T_2 \dots T_n$. The detailed version of this algorithm is found in Appendix 1.4.

5.1.3.3 Identifying E-Rs from “temporarily store for insert” annotated navigation widgets

Entity-Relationships can be identified from “temporarily store for insert” annotated navigation links among containers. The discussion on the “insert” business transaction in Section 4.2.3 introduced the idea of finding database entity relationships by identifying containers linked by the “temporarily store for insert” annotated navigation widgets. Specifically, in Figure 17 the “Selected Order Details” *Data View Container* is linked to the “Customer” *DFY Container* which in turn is linked to the “Payment” *DFY Container* via the “temporarily store for insert” annotated navigation widget, as a part of an insert business transaction. Since the visual model signifies real world links between entities such as “Travel” (via the “Selected Order Details” container) and “Customer” and between “Customer” and “Payment”, correspondingly E-R can also be established from it. Thus, a “Travel-Customer” relationship and a “Customer-Payment” relationship is identified if not already found elsewhere in the model. In our example since “Customer-Travel” relationship is already identified in the previous section the only new relationship found is between “Customer” and “Payment”. Hence the consolidated E-R diagram will be as shown in Figure 25.

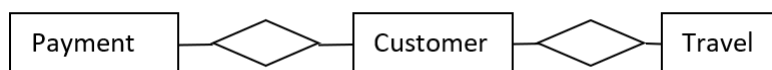


Figure 25: Payment-Customer-Travel entity relationships from “temporarily store for insert” annotations

Though the above discussion illustrated how a “temporarily store for insert” annotated navigation widget provides links between a *Data View Container* (as its source container) and a *DFY Container* as its target container(i.e. between “Selected

Order Details” and “Customer”) and also between two *DFY Containers* (i.e. between “Customer” and “Payment”), it did not illustrate that a *Search Container* could have been a target of a “temporarily store for insert” annotated navigation widget. Consider a scenario where an existing (rather than a new) customer needs to book and pay for a travel deal. In such a case, the mock-up model could be as shown in Figure 26 where the “Selected Order Deal” *Data View Container* in the “Order Deal” page links with “Search Customer” container via the “temporarily store for insert” link. Further the selected customer in “Selected Customer” *Data View Container* is linked with the “Payment” *DFY Container* through the “temporarily store for insert” link.

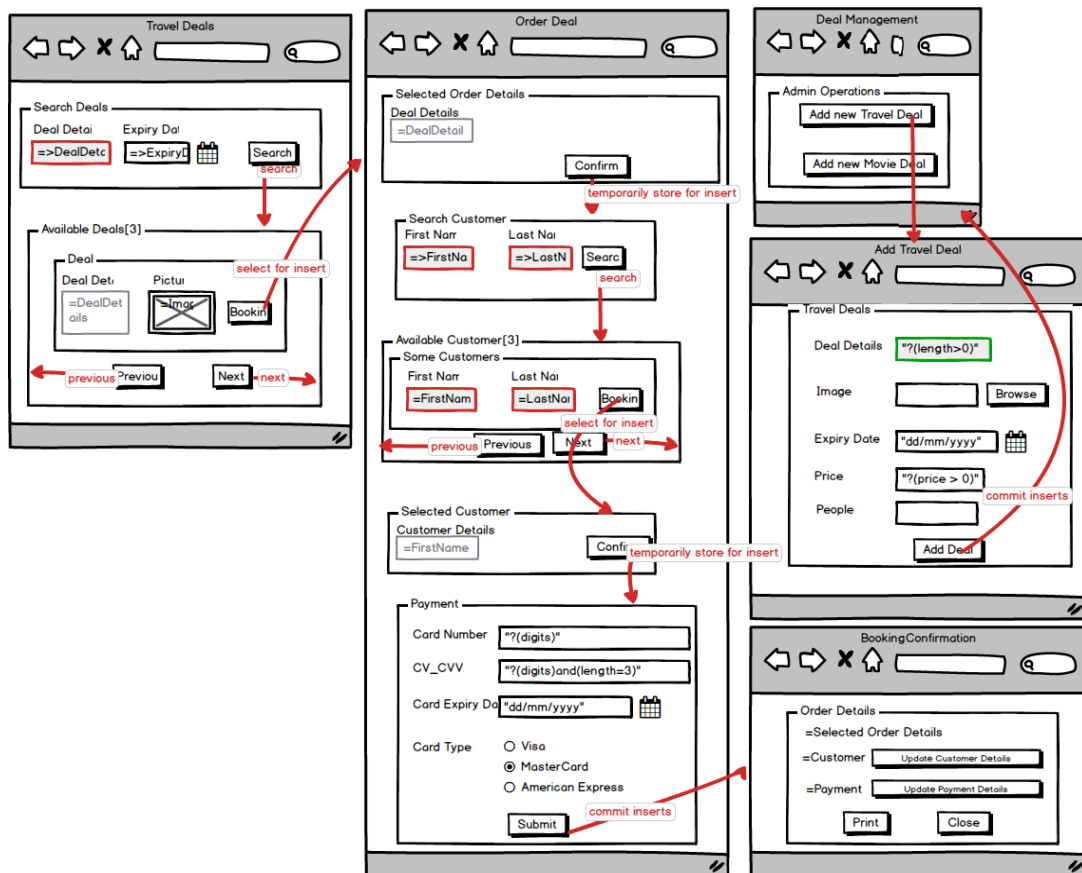


Figure 26: Search Container with "Temporarily store for insert" annotations among containers

In such a case, the “Travel” entity referenced by “=DealDetail” in “Selected Order Details” container should be linked to the Customer entity referenced by “=FirstName” in the “Selected Customer” *Data View Container*. That is an E-R relationship needs to be established between Travel table and Customer table. In addition, the “temporarily store for insert” annotated navigation link between

“Selected Customer” *Data View Container* and “Payment” *DFY Container* indicates a relationship between a “Customer” and a “Payment” entity.

The above discussion yields the following generic algorithm for identifying relationships from “**temporarily store for inserts**” annotated navigation widget among containers:

Find the first container that is a source of a “**temporarily store for insert**” annotated navigation widget. The source container can either be a *DFY Container* or a *Data View Container*. The target of the “**temporarily store for insert**” annotated navigation widget can either be a *Data View Container* or a *DFY Container* or a *Search Container*. If the target is a *Search Container* replace it by the *Data View Container* used for displaying the select research result. Similarly replace any *Data View Container* by the corresponding *DFY Container(s)* it references in both the source and target containers. Establish entity relationships among the *DFY Containers* in the source and target containers. Continue this process until a “**commit inserts**” annotated navigation widget is found among the linked list of navigation widgets. The detailed version of this algorithm is available in Appendix 1.5.

5.1.4 The generated E-R model of the example application

This section provides a consolidated summary of how database tables and their relationships were identified from the mock-up of the Travel Deal example. Specifically, the following database tables are identified: “Travel”, “Customer”, “Addresses”, and “Payment Detail because they correspond to *DFY Containers*. Similarly, the following relationships were identified: “Customer-Address”, “Customer-Payment” and “Travel-Customer”. That is “Customer-Address” relationship is found due to the nesting of the “Address” *Data Input Container* within the “Customer” *DFY Container*. The “Customer-Payment” relationship is identified due to the “**temporarily store for insert**” link between the “Customer” and “Payment” *DFY Containers*. Similarly, the “Travel-Customer” relationship is identified from the “**temporarily store for insert**” link between the “Selected Order Details” *Data View Container* and the “Customer” *DFY Container*. Note that the “Selected

Order Details” container indirectly refers to a “Travel” entity. Hence the relationship is between “Travel” and “Customer” and not between “Selected Order Details” and “Customer”. The complete E-R model of the “Travel Deal” case study is shown in Figure 27.

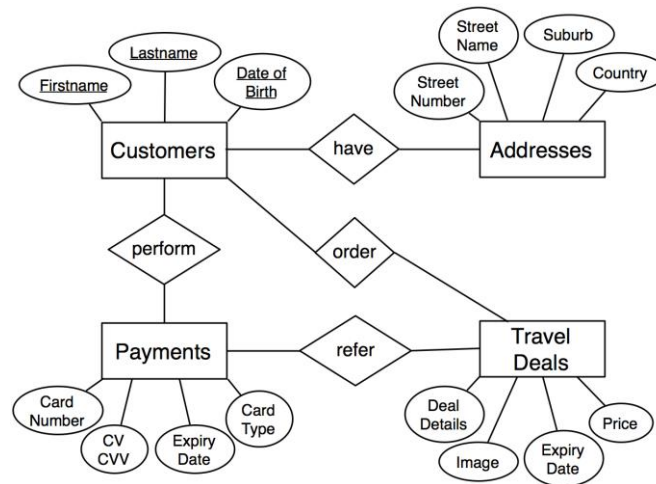


Figure 27: E-R Model of the Travel Deal case study

This completes the discussion on the algorithms for deriving the database schema from the visual mock-up. The next section discusses the algorithms for deriving the components to manage client-server communications, from the visual mock-up.

5.2 MVC-MC Generator for search operations

Searching is one of the behaviours requiring both client and server-side components in SME applications. This section discusses algorithms for deriving client-side and server-side components of the auto-generator from the mock-up, to manage search operations. Since the server-side logic is closely associated with database operations, it also deals with the retrieve operations of the database logic for the corresponding search operation. This section provides answers to RQ2.2 and RQ2.3 from a “search” operation perspective. Search operation is a part of the “retrieve” database logic in the “Create Retrieve Update Delete” (CRUD) family of operations. The other aspect

of “retrieving” data happens during report generation, which is dealt separately in Section 5.4.

The auto-generated SME application follows the MVC-MC architecture based on the MVC pattern recommended by Reenskaug(1979). As discussed in Section 2.2.2 since RIAs have both client-side and server processing components, the acronym MVC before the hyphen in MVC-MC refers to the client-side components, whereas the MC following the hyphen refers to the server-side components (D’Souza & Ginige 2010). The MVC-MC generator will utilise the data model discussed in Section 5.1 and the visual mock-up to auto-generate the client and server-side components. The algorithms in the MVC-MC generator can be better explained using scenarios from the Travel Deals example illustrated in Figure 12. The following paragraphs provide an overview of how MVC-MC components are used to manage search operation. Once this is explained, the derivations of the MVC-MC components will be discussed.

Let us first consider the client-side MVC components. The “Travel Deals” page in the Figure 12 has two main containers, namely “Search Deals” *Search Container* and “Available Deals[3]” *Search Result Container*. Here “[3]” represents 3 sets data are to be displayed per search result page. A runtime instance of this page is generated by the Client-Side View (CSV) and the CSV is defined by the MVC-MC generator. Here CSV is the logical unit that manages presentation of data and UI on the client-side. The MVC-MC generator utilizes the data model and the visual mock-up to define the View logic to handle user interactions at runtime. For example, when a user enters search details in the “Search Details” *Search Container* and then clicks the “Search” button to get a resultant list of entities, the MVC-MC generator should be able to find the actual entities that are required to be displayed, using pagination effects. These resultant entities should be managed using Client-Side Models (CSMs). A CSM is a logical unit for managing business entities on the client-side. Similarly, traversal of the resultant entities is also required. As discussed in Section 4.2.2 the “previous” and the “next” button in the “Available Deals” *Search Result Container* are used for traversing through the full result list. For successful completion of these operations the View should manage the rendering of the page based on the correct subset of data in the CSM. Further, the *Search Container* and the *Search Result Containers* are

linked through the “**search**” annotated navigation widget. For example, the “**search**” annotated navigation widget links the “Search Deals” source container with the “Available Deals[3]” target container. From this discussion, it is evident that a series of actions are required to be set in sequence for managing the search operations. The MVC-MC architecture utilizes a Client-Side Controller (CSC) to manage client-side operation such as “**search**”. A Client-side Controller (CSC) is a client-side control logic unit to manage client-side presentation layer and client-side model layer. The following paragraphs discuss the search operation in terms of the MVC-MC Components required for searching using the *Search Container* and for managing the search result using the *Search Result Container*.

Let us consider the “Search Deals” container in more detail to study how the behavioural logic in a *Search Container* can be derived from the mock-up and the data model. The look-up widgets in the “Search Deals” container specify the references from where the data needs to be searched using the look-up notation. From the discussion on look-up widgets provided in Section 4.2.2 it is clear that each widget in a look-up container requires finding of a corresponding DFYW. Similarly, from the discussion on the derivation of the data model in Section 5.1, it is evident that a DFYW corresponds to a field in a database table. In other words, the table names and the field names associated with look-up widgets in a *Search Container* specify the search criteria in a database. In Figure 12 it is assumed that the field names are unique across the whole database, so the table names are omitted in the lookup widgets in the *Search Container*.

The search operation also requires information regarding the selection of data to be presented from the search result. This information is got from the target container of the “**search**” annotated navigation widget. In the example, “Available Deals[3]” *Search Result Container* provides this information. In particular, “Available Deals[3]” contains data such as “Deal Details”, “Picture”, and “Expiry Time”. Observe that some of this information is different from the search criteria in the “Search Details” *Search Container*. That is, the search criteria may be different from the search result display criteria.

Each of the auto-generated MVC components has a well-defined task. For example, a CSM for search operation should contain attributes for each *look-up widget* in the *Search Container* as well attributes for the data to be displayed in the *Search Result Container*. In addition, the CSV logic of the *Search Container* and the *Search Result Container* should contain presentation information from the associated containers in the mock-up. The CSV and CSM operations are managed together by a CSC. An example of this is the management of traversal through the search result list where appropriate values from the CSM are sent to the CSV for display based on the user's needs. In addition, Controllers and Models are also required on the server side.

Figure 28 is a sequence diagram for search operation using MVC-MC components.

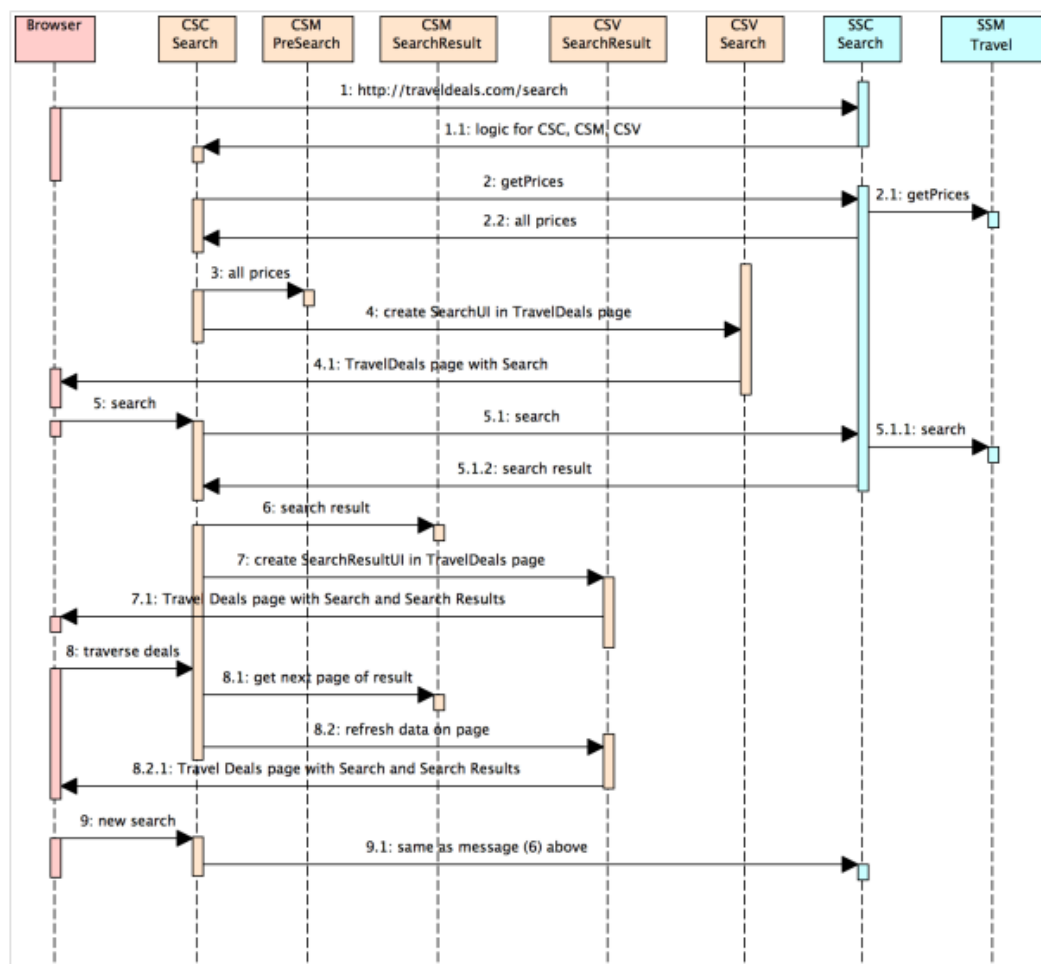


Figure 28: Sequence diagram showing interaction among web components for a search operation in Travel Deal web app

In the sequence diagram, the first call identified by message 1 occurs at the initial load time of the page. Since no client-side components exist at this stage the search

request is processed by a Server-Side Controller (SSC) dealing with search. This results in sending the Client-Side Controller (CSC), Client-Side View (CSV), and Client-Side Model (CSM) components to the client. In MVC-MC RIA architectures, html pages are always generated by CSV on the client-side on behalf of CSC. In Figure 28 the CSC component is termed “CSC Search”, while the CSV component is termed “CSV Search”. Users can view the “Search Deals” container in the “Travel Deals” page because the “CSC Search” requests “CSV Search” (message 4 in Figure 28) to execute the presentation logic once message 1 is completed. Observe that the “price” combo box in the “Search Deals” *Search Container* requires a list of all currently available prices to be populated from the corresponding “price” field in a database table, before the page is rendered. This information is got by the “CSC Search” requesting the “SSC Search” to retrieve all prices via message 2 in Figure 28. The above process results in the “Search Details” *Search Container* to be displayed in the *Travel Deals* page. Now the user can enter the search details. Message 5 represents a user’s click of the submit button on entering the search details. The search entry details are captured by “CSC Search”, which then invokes message 5.1 to request server-side data using “SSC Search”. The latter then requests via message 5.1.1 the appropriate Server-Side Model (SSM) to retrieve the requested data from the database where a SSM represents the information (the data) of the application and the business rules required to manipulate the data on the server side.

“SSM Travel” retrieves the requested data from the database and sends it to “SSC Search”. The response (message 5.1.2) is then passed to the CSC. The CSC then requests (message 6) the services of CSM, that is “CSM SearchResult”, to create a list of available travel deals. CSC also requests (message 7) a CSV, namely “CSV SearchResult” to execute the presentation logic for displaying data in the “Available Deals[3]” *Search Result Container*.

Subsequent requests for traversal (using the “previous” and “next” buttons in the “Available Deals[3]” container) through the list of deals that are already loaded are handled only on the client-side. This is illustrated by messages 8, 8.1 and 8.2 in Figure 28. These messages result in refreshing the CSVs with appropriate selections of data from “CSM SearchResult”.

In the visual model (see Figure 12), an annotated navigation widget is used to specify the source that initiates a behavioural action and its target. Correspondingly during the runtime of a RIA, the source and the target of a navigation widget are generated by appropriate CSVs and the control action is provided by a CSC. It can be observed from the sequence diagram that a CSC requires data and data processing logic on the client-side which is represented as CSM in the sequence diagram. Further, if the data is not available on the client-side, the CSC also needs to communicate with the SSC to access server-side data through the services of SSMs since client-server communication is maintained only between CSC and SSC.

From the above discussion, it is evident that principally five types of components are sufficient to handle any client-server and client-only communications: CSC, CSV, CSM, SSC and SSM. Further, since the communication process is always initiated by a CSC on behalf of the user, the complexity of deriving a web application can be reduced to the derivation of these five components from the mock up and the data model. Section 5.2.1 discusses the auto-generation of these components in further detail from a search operation perspective.

5.2.1 Component generation for a search operation

In the running example the search operation is carried on in the “Travel Deals” page, which requires the auto generation of:

a) A CSV containing a HTML form for entering the travel deal search criteria. The structure of this form is derived from the source container of the “**search**” annotated navigation widget. In Figure 12, the container is called “Search Deals” and its runtime equivalent is generated by the “CSVSearch” object in the sequence diagram.

b) SSMs to find “Travel” entity related to the search criteria by making appropriate type of queries to the database. At runtime, the “Travel” entity and its fields to be searched are identified from the key-value pairs in the server request where each key represents the name used to represent a *looked-up widget* in the Search Container. At runtime each *looked-up widget* name is of the form “DFY Container Name__DFYW Name”. For example, a “Travel__DealDetails” key would identify the “DealDetails”

field to be searched in the “Travel” table. “SSMTravelDeals” is an example of a SSM instance is in the sequence diagram.

c) CSMs to manage the “Travel” entities received from the server-side. The CSM should also have appropriate client-side processing code for traversing the “Travel” entities search result list. At runtime, the server responds with key value pairs where the keys are the names of the entities along the names of the fields, from which the CSMs are derived. In Figure 28, “CSMSearchResult” is a representation of such a CSM.

d) A CSV template for the runtime generation of “Available Deals[3]” container in Figure 15. The template is automatically derived from the structural model of the target container (i.e. “Available Deals[3]” *Search Result Container* is the target) of the “**search**” annotated navigation widget in the mock-up. In addition, the CSV should also contain the processing logic for rendering new data in “Available Deals[3]” *Search Result Container* when users traverse through a search result list. “CSVSearchResult” represents a runtime object for such a CSV in the sequence diagram.

e) A CSC to manage asynchronous calls to the server and to invoke appropriate functionality of CSM and CSVs during user interaction in the “Travel Deals” page. The CSC is derived by registering the various events related to the behavioural elements in the *Search Container* and *Search Result Container* with corresponding event handlers in CSMs and CSV. In addition, at runtime it should invoke appropriate functionality when automatically generated events are triggered. For example, in the “Travel Deals” page, the CSC registers user event handlers for behaviours associated with “**search**”, “**previous**”, “**next**”, “**select for insert**”, “**select for update**” and “**delete**” annotated navigation widgets. In addition, since the “price” and “people” combo boxes in the “Search Deals” container are pre-populated with unique values of “price” and “people” from the “Travel” database table, the CSC invokes a function when an event for document loading complete is automatic triggered by JavaScript. In Figure 28,, the “CSCSearch” is an example of a CSC object in the sequence diagram.

f) A SSC to assemble the components required for the “Travel Deals” page when a user requests the search page the first time in a session. That is the SSC assembles a

page with components for CSVs, CSMs, and CSCs. The desired information for each component is derived from the mock-up of the “Travel Deals” page. In the sequence diagram “SSCSearch” represents the *Server-side Controller* for the “Travel Deals” page and uses the services of the server-side model, named “SSMSearch”, to get the desired search results from the database.

The MVC-MC auto-generator uses Knockout.js⁸, JQuery.js⁹ and HTML on the client side and PHP¹⁰, CodeIgniter¹¹ and MySQL¹² on the server side to define the MVC-MC components. Sub-section 5.2.1.1 to sub-section 5.2.1.5 discuss how the MVC-MC components are auto-generated using these technologies, from a search operation perspective. In particular, Section 5.2.1.1 deals with auto-generation of CSV for search; Section 5.2.1.2 discusses auto-generation of SSM for search; Section 5.2.1.3 discusses the auto-generation of the CSM to manage search results; Section 5.2.1.4 deals with the auto-generation of CSC for search and finally Section 5.2.1.5 dwells on the auto-generation of SSC for search.

5.2.1.1 Auto-generation of Client-Side Views for Search

This section illustrates how the technologies mentioned above are utilized by the MVC-MC Generator for the auto-generation of the CSV for search. A client-side view for search needs two components, one for entering the search details and the other for displaying the result (see (a) and (b) in Section 5.2.1 for further details). In Figure 12, they are represented by “Search Deals” *Search Container* and “Available Deals[3]” *Search Result Container*, respectively. The auto-generator for CSVs derives the structural details of these components directly from the mock-up. However, the behavioural logic of the views requires additional effort. For example, in the “Search Deals” container, “Price” is a combo-box that is required to be pre-populated with the list of available prices from the “Travel” database table. This behavioural

⁸ <http://knockoutjs.com/>

⁹ <https://jquery.com/>

¹⁰ <http://www.php.net/>

¹¹ <https://codeigniter.com/>

¹² <https://www.mysql.com/>

requirement specification is gleaned from the data model and from the property for the pre-population of the “Price” look-up widget in “Search Deals” container. In, Figure 28 messages 2 and 2.1 represent pre-population requests to a SSM via the SSC for a list of available prices in the “Travel” table. This implies that whenever a widget requires to be pre-populated with values from the database the auto-generator should derive the processing code for a select query on the database.

In general, pre-population of a data widget requires a corresponding SSM code of the following type:

```
`Select tablem.fieldxName where tablem.fieldxName LIKE ''` .
```

This code is like the generic select statement for a “search” operation. This implies that default search methods should be available for each attribute associated with a corresponding database field in the SSM to pre-populate the combo boxes. Correspondingly on the client-side, the CSV requires data-binding of CSM objects with appropriate widgets. For example, the “price” widget in Figure 12 contains values from “CSMSearch”, the *client-side model* for search. Data-binding is the process that establishes a connection between the application UI (User Interface) and Business logic in Knockout.js JavaScript library. Similarly, the CSV needs to store user-selected data. Hence, the auto-generator should specify data binding of the widgets in the CSVs with appropriate CSM attributes. In addition, the CSV requires the definition of the widgets to manage the results of a search. For example, the “Available Deals[3]” *Search Result Container* should manage the display of appropriate data entities when a user clicks either the “Previous” or the “Next” button in Figure 12.

Figure 29 is a snapshot of the generated “CSVSearch”. It highlights two aspects of the auto-generator, firstly the id and name of each element in the *Search Container* is derived from the container name and the widget name, secondly it highlights how some of the widgets are data bound to CSMs using Knockout.js.

The green boxes in Figure 29 illustrate the names of the widgets include both *DFY Container* name as well as the widget name in the form “DFY Container

Name__DFYW Name”. This notation is generated by associating each *looked-up widget* in the visual mock in Figure 12 with their corresponding *DFYW* names and *DFY Container* names. Consider the case of widget “Travel__DealDetails” by observing the first green box Figure 29. On comparing Figure 29 with the mock-up in Figure 12 it is evident that “Travel” in “Travel__DealDetails” is derived from the “Travel” *DFY Container* name and “DealDetails” from the “DealDetails” *DFYW* in the “Travel” *DFY Container* in the mock up. Further from the discussion in Chapter 5 it is known that “Travel” is a database table and “DealDetails” is a field in the table. Section 5.2.1.2 provides further information on how this link is obtained. When the data from an *HTML FORM* in Figure 29 is posted to the server side in the form of key-value pairs where each key of the form “DFY Container Name__DFYW Name”, it is easy for a CSC to sort the keys in terms of database table names. Further the maroon coloured underlined text in Figure 29 illustrates how the data in the *Client-Side View* is bound to the *Client-Side Model*. For example, the segment, “data-bind=options: dataPrice1” in

```
<select data-bind="options: dataPrice1, optionText:'Price',
optionsValue: 'Price1', value:selectedPrice1"
id="Travel__Price" name="Travel__Price"></select>
```

binds the “dataPrice1” attribute of “CSMPreSearch” (in Figure 28) to the “Price” widget identified by the code id="Travel__Price" name="Travel__Price" in “CSVSearch”.

```
<div id="TravelDeals_1-DIV"><B>Search Deals</B>
<form id="TravelDeals_1-FORM" action="" method="post" autocomplete="on">
<table>
<tr>
<td>DealDetails</td>
<td>ExpiryDate</td>
<td>Price</td>
<td>People</td>
</tr>
<tr>
<td valign="top"><input type="text" id="Travel__DealDetails" name="Travel__DealDetails" />
</td>
<td valign="top"><input type="date" id="Travel__ExpiryDate" name="Travel__ExpiryDate"/>
</td>
<td valign="top"><select data-bind="options: dataPrice1, optionText:'Price', optionsValue: 'Price1',
value:selectedPrice1" id="Travel__Price" name="Travel__Price"></select>
<span data-bind="text: selectedPrice1" id="PriceSpan" style="visibility:hidden; display:none"></span>
</td>
<td valign="top"><select data-bind="options: dataPeople1, optionText:'People', optionsValue: 'People1',
value:selectedPeople1" id="Travel__People" name="Travel__People"></select>
<span data-bind="text: selectedPeople1" id="PeopleSpan" style="visibility:hidden; display:none"></span>
</td>
<td><button id="Search1" name="button" type="submit" value="Search">Search</button></td>
</tr>
</table>
</form>
</div>
```

Figure 29: Snapshot of the auto-generated code for Client-Side View of a Search Container in Travel Deal web app

Similar discussion is valid for binding “dataPeople1” with “Travel__People”. Further, “Travel__People” and “Travel__Price” are combo boxes in the mock-up. User selected data from these combo boxes are bound to “selectedPeople1” and “selectedPrice1” respectively. Here “dataPrice1”, “dataPeople1”, “selectedPrice1” and “selectedPeople1” are the corresponding CSM attributes in knockout.js. The post-fix “1” indicates that these CSM attributes are for the first *Container* in the page. In general, “CSMPreSearch” has a list attribute to store the values of each multi-valued widget such as combo boxes and an attribute each to store selected items from such widgets. Please refer to Knockout.js website for more details of how data-binding mechanisms between CSMs and CSVs is carried out.

When the user clicks the “Search” button in the travel deals page an AJAX call is made to the SSC using jQuery. Consider the example of an AJAX search request from the client side:

```
request = $.ajax({
  url:
'http://localhost/Balsamiq_VINCIexampleSuccess/SSC/search',
  type: "post",
  data: $('#TravelDeals_1-FORM: input').serialize (),
  datatype: "json"
});
```

The URL in the above POST request is to a “CodeIgniter” MVC framework on the server side. CodeIgniter is configured to a MySQL server for server-side storage.

An example of the information sent to SSC when a user enters no data in the *Search Container* and clicks the “**search**” button is:

```
"Travel__DealDetails=&Travel__ExpiryDate=&Travel__Price=&Travel__People="
```

In the above code, “Travel” refers to the table name and the associated field names are found tokenizing the text between “__” and “=”. Here, since no search data was entered, no values are associated with any of the fields.

The response data representation is of JSON type and has the following form:

```
{
  "0": ["Deal Details: string", "Deal Picture: Image", "Expiry
Time: Date"],
  "1": ["Paris Travel Deal...", "http://paris.com/img.jpeg",
"20-6-2017"],
  "2": ["Italian Travel Deal...", "http://italia.it", "10-5-
2017"]
}
```

The object associated with index "0" identifies the data types of each field in the result set and the other objects represent the response from the SQL select query. The JSON response is mapped to a CSM ("CSMSearchResult" in Figure 28) containing attributes such as "allData2", "pageData2", "nextStartIndex2" and "pageSize2". Here, "allData2" is a list that is populated with the full search result response, "pageData2" is another list containing a subset of "allData2" pertaining to a page of data to be displayed within the *Search Result Container*. The postfix "2" for the various attributes indicates that the model is for the second main container in the page. Hence each main container in the page will have its own Model. In addition, "nextStartIndex2" holds an integer to manage enabling and disabling of the buttons associated with sources of "previous" and "next" annotations on the navigation widgets in the *Search Result Container* and "pageSize2" manages the number of records to be displayed at a time in the *Search Result Container*.

Figure 30 represents the auto generated code of the *Search Result Container*. Here the outer <div> corresponds to the "AvailableDeals [3]" *Search Result Container* and the inner <div> is equivalent to the "Deal" *Data View Container* in the mock-up. The inner <div> uses the *foreach* feature of Knockout.js to create instances for display and refresh by utilizing a template identified by the name "resultRowTemplate2". The blue box in Figure 30 contains the definition of the template. The *foreach* feature is linked to the "pageData2" attribute of the *Client-Side Model*. In this example "pageData2" will have three sets of items, where each attribute in the set corresponds to the "data_bind" field in the blue box in Figure 30. That is each *data view widget* in the "Deal" *Data View Container* will have an associated "data_bind"

field associated with a CSM element. A maximum of three sets of values will be displayed at a time in the inner <div> since “AvailableDeal[3]” has 3 within the square brackets.

```

<div style="display: none" id="TravelDeals_2-DIV"><!-- RRRAvailableDeals[3]-->
<div class="content2"><B>Available Deals[3]</B>
  <table data-bind="template: (name:'resultRowTemplate2', foreach:pageData2)">
  </table>
</div>
<script type="text/html" id="resultRowTemplate2">
<tr>
<td>DealDetails</td>
<td>Picture</td>
<td>Price</td>
<td>ExpiryTime</td>
</tr>
<tr>
<td valign="top"><textarea rows="6" cols="16" data-bind="text: Travel_DealDetails" readonly ></textarea></td>
<td valign="top"><img alt="Travel_Image" data-bind="attr:{src: Travel_Image}"/></td>
<td valign="top"><input type="text" data-bind="value: Travel_Price" readonly /></td>
<td valign="top"><input type="text" data-bind="value: diffTime(Travel_ExpiryDate, 'TimeNow')" readonly /></td>
<td><button id="Booking2" name="button" type="submit" value="Booking">Booking</button></td>
<td><button id="Update2" name="button" type="submit" value="Update">Update</button></td>
<td><button id="Delete2" name="button" type="submit" value="Delete">Delete</button></td>
</tr>
</script>
<td><button id="Previous2" name="button" type="submit" value="Previous">Previous</button></td>
<td><button id="Next2" name="button" type="submit" value="Next">Next</button></td>
</tr>
</div>

```

Figure 30: Snapshot of the auto generated code for Client-Side View of a Search Result Container in Travel Deal web app

In general, the auto-generation of a CSV for a search operation requires a *client-side view* component for the rendering search input, a *client-side view* component for rendering search results, a *client-side controller* logic to pre-populate appropriate multi-valued widgets in the search input container and to manage a search request, and behavioural logic to manage and render search results.

5.2.1.2 Auto-generation of Server-Side Models for Search

Server-Side Models manage business operations on the server side. The auto generation of SSMs involves two types of activities: (a) Creation of a SSM for each database table in the data model and (b) Creation of a SSM for each Entity Relationship in the data model. The auto-generation process for individual database tables is quite simple. Each SSM will have a method to create a new record, select record(s), update record(s), and to delete record(s). This results in standard Create Read Update Delete (CRUD) operations on a single table. Each of these methods will have appropriate parameters to specify the selected fields, the conditions on which the selections are done, and the order of retrieval. Since the CRUD algorithms on

single tables are standard, no further details are being provided in this thesis. However, the auto-generation of SSM involving more than one table requires additional relationship information from the data model. The process of finding E-Rs is discussed Section 5.1.3. This section discusses the auto-generation of the SSMs involving one or more tables from a search operation perspective.

In the sequence diagram (Figure 28) “SSMSearch” is an example of a *Sever Side Model* for searching and retrieving appropriate data in the database. It contains a search method to retrieve selected data from one or more table fields based on a search condition. The search criteria include fields associated with one or more database tables and the selection criteria include selection of fields whose values need to be retrieved. With respect to Figure 12, the auto-generator finds the specification of the search criteria fields from the looked-up references in the *data input widgets* in the “Search Deals” *Search Container* and the specification of the selection criteria is found from the *data view widgets* in “Available Deals[3]” *Search Result Container* in the *Travel Deals* page. Comparing the mock-up in Figure 12 with the auto-generated *Search Container* code in Figure 30 provides evidence of how the names and ids of the widgets are auto-generated from the mock-up in terms of *DFY Container* names and *DFYW* names. Please refer to Section 4.2 for a review of the terminology used and to Section 5.1 for a review of how some of the widget names are related to database tables and database table field names.

Let us assume x fields from m tables are selected based on a conditional expression on y fields from n tables. From the visual model, a general query of the following form is defined by the auto-generator for the search method in *SSMSearch*:

```
"SELECT table1.field1Name, table1.field2Name,  
table2.field3Name..., tablem.fieldxName where table1.field1Name  
LIKE DATA1 and table1.field2Name LIKE DATA2 and  
table2.field3Name LIKE DATA3 and..., tablen.fieldyName LIKE  
DATAy"
```

In the given example table_m.field_xName details are got from “Available Deals[3]” *Search Result Container*, table_n.field_yName details are got from “Search Deals” *Search Container* in the “Travel Deals” page, and “DATA_y” is got at runtime from the user. The result of the query is returned as key-value pairs. Similarly, the result can also

contain the database meta-data such as table field names and its data-types. The detailed version of this algorithm is presented in the Appendix 2.1.

5.2.1.3 Auto-generation of Client-Side Models for Search result

This section discusses how CSMs for managing search results are auto-generated. Specifically, it explains how a component such as “CSMSearchResult” with respect to Figure 28 is auto-generated. CSMs come into picture on receiving a response from the server. From the discussion in Section 5.2.1.1 it is known that the server response is a JSON object representing a list of zero or more data-sets to be displayed in the *data view widgets* of the *Search Result Container*. Furthermore, from the discussion on *Search Result Container* in Section 4.2.2 it is also known that the container displays search results in paginated form. Hence the corresponding CSM should have a list attribute (say “allData”) to hold the response data-sets. In addition, it should have another list (say “pageData”) to represent each sub-page of results to be displayed. Moreover, the number of data-sets to be displayed in each sub-page needs to be defined. This can be defined in an attribute, “pageSize”. Another important attribute is the index of the next data-set within “allData” to be loaded into “pageData”, when a user clicks the “Previous” and “Next” buttons in the *Search Result Container*. This may also result in automatically enabling or disabling either or both buttons at runtime depending on the number of items in “allData”. The definition of each data-set is got from the mock-up of the inner container in the *Search Result Container* and “pageSize” value is obtained from the value within the “[]” brackets in the name of the *Search Result Container*. Further, whenever there is a list of entities, the corresponding standard algorithms for insertion, search, traversal, deletion, update, and sort can be added by default, though not all may be relevant to the search process. From the above discussion, an abstract class diagram of a CSM for managing search result is defined as shown in Figure 31.

CSM-SearchResult	
Entity{ Field ₁ name Field ₁ type Field ₁ value ... Field _n name Field _n type Field _n value }	
List of entities[] Index of the first record to display	
find(key)	#finds an entity
update(index, newValue)	#updates an entity
delete(index)	#deletes an entity
sort(order)	#sorts a list
insert()	#inserts an entity in a list
slice(count)	#returns a slice of entities
getNextSlice(order)	#returns the next set of entities
compare(entity1, entity2)	#compares two entities

Figure 31: An abstract Client-Side Model for search result

In the above definition of the CSM for search result, ‘Entity {...}’ defines all the attributes within a data-set to be displayed in the search result and ‘List of Entities’ defines a list of search result objects to be displayed. Read Appendix 2.2 for the further details regarding the attributes of CSM for managing search results.

No discussion is provided here for CSM for search operation. This is because user entered data for a “search” operation is generally not required to be processed on the client side. However, CSM for search will be required in some circumstances. For example, the “Search Deals” *Search* Container has a “Price” combo box that is pre-populated with a list of unique travel deal “prices” in the system. In this case a CSM for search too would be required. However, the general principles of deriving the CSM for search is same as that of CSM for search result. Hence no further discussion is provided here.

5.2.1.4 Auto-generation of Client-Side Controller for Search related operations

The CSC manages asynchronous calls to the server and invokes appropriate functionality of CSM and CSVs during user interaction. Specifically, a CSC for search related operations performs control services for:

- a) Invoking creation of CSM objects,

- b) Invoking CSM methods to pre-populate multi-valued look-up widgets in the *Search* container,
- c) Managing “search” behaviour using the services of CSMs and CSVs,
- d) Managing display of search results using the services of CSMs and CSVs.

The CSC is auto-generated by registering the various events related to the behavioural elements in the *Search Container* and *Search Result Container* with corresponding event handlers in CSMs and CSV. In addition, at runtime it invokes appropriate functionality of the event handlers when automatically generated events are triggered. CSC for search is defined by registering user event handlers for behaviours associated with “**search**”, “**previous**”, “**next**”, “**select for insert**”, “**select for update**” and “**delete**” annotated navigation widgets, where each event handler invokes appropriate CSMs and CSVs methods to perform the desired behaviour. Hence the algorithm for the derivation of CSC for search related operations is to register a function for each behaviour that exists in the mock-up. In addition, if drop-down boxes in *Search Containers* are required to be pre-populated with unique values from corresponding fields in a database table, then the CSC needs to invoke a function to accomplish the same. For example, the “Search Deals” container in the “Travel Deals” page requires “price” and “people” combo boxes in the “Search Deals” container to be pre-populated with unique values of “price” and “people” from the “Travel” database table. The CSC should manage this by invoking a function when the on-load event of the “Travel Deals” page is triggered.

Finer details of the auto-generation of CSC for search related operations are provided in Appendix 2.3. Specifically, Appendix 2.3.1 deals with a CSC helper function for search operations, Appendix 2.3.2 for a CSC helper function for search result navigation and Appendix 2.3.3 for a CSC helper function for managing on-load event operations.

5.2.1.5 Auto-generation of Server-Side Controller for Search related operations

The SSC for a search operation has two main responsibilities. During the first request, it should assemble all the desired client-side components and during subsequent requests, it must access appropriate SSMs on behalf of the CSC. The algorithm to manage the first request is:

```
Include appropriate client-side library code such as Knockout.js
and JQuery.js for each page request.
Include the code to manage CSMs, CSVs and CSCs for each page
request.
Send the files to the client-side.
```

The algorithm to manage subsequent requests is:

```
Get the requested key-value pairs from the request post.
Identify which Server-Side Models are required to be invoked.
(See Appendix 2.1.1 for finer details on the above two actions)
Identify the methods to be invoked from the Server-side Models
from the URL.
Return the results to the client in JSON form.
```

Consider the following request for the “Travel Deals” page in Figure 28:

```
'http://localhost/Balsamiq_VINCIexampleSuccess/ASSC/TravelDeals'
```

Here “Balsamiq_VINCIexampleSuccess” is the name of the web application and “ASSC” is the name of a Server-Side Controller and “TravelDeals” is the name of the page to be requested. In response, the SSC will load the file, “TravelDeals.html”, along with the necessary client-side JavaScript library files identified within the html file. The auto-generator will define a html page for each page in the mock-up. The JavaScript library files are required for each page in the mock-up to manage desktop like features by the client-side of RIAs. Next the code to manage CSMs and CSVs and CSCs are defined in “TravelDeals.html” as explained in Section 5.2.1.1 through to Section 5.2.1.4. On the initial request of a html page, the predefined html page is loaded on the browser along with the necessary JavaScript library files. Once the page is loaded the CSC will take control of client-server communication. Consequently, each request from the CSC to the server will be handled by the SSC.

A SSC is required for any operation that is dependent on server-side data. That is, SSCs exist for the following annotations on the navigation widgets: **“search”**, **“commit inserts”**, **“update”**, and **“delete”**. It may be noted that **“select for insert”**, **“temporarily store for insert”**, **“select for update”**, **“previous”** and **“next”** are not included here since these are primarily client-side operations. However, in some occasions **“temporarily store for insert”** will also invoke server-side functionality in addition to storing data temporarily on the client-side side during an insert business transaction. Specifically, if an intermediate operation during an insert business transaction requires file upload, an AJAX call is made to the server side and the URL of the uploaded file is stored on the client-side instead of storing the file itself. This is necessary to conserve client-side storage due to its limited size.

Apart from this exception, a SSC for search is derived from the **“search”**, **“commit inserts”**, **“update”**, or **“delete”** annotations on the navigation widgets in the mock-up and the expected key-value pairs from the requested URL. From the discussion in 5.2.1.1 it may be recalled that such an URL is of the form:

```
'http://localhost/Balsamiq_VINCIexampleSuccess/ASSC/search'
```

and as each key in a key-value pair is of the form **“DFY Container Name__DFYW Name”**, it is easy for a CSC to sort the keys in terms of database table names. That is the SSC can identify the desired SSM model and SSM method to be executed from the requested URL.

This completes the discussion on identifying the client-side and server-side components of the auto-generator from the mock-up for a search operation. The next section does the same from an insert business transaction perspective.

5.3 MVC-MC Generator for Insert operations

Insertion of new records is another business operation requiring client and server-side components in web applications. As discussed in Section 4.2.3 each insert business transaction generally involves several physical transactions, which may be carried on in a single web page or in a series of web-page interactions. In the mock-up language, annotations such as **“select for insert”**, **“temporarily store for insert”** and **“commit inserts”** are used on navigation widgets to manage the mock-up of an

insert business transaction. This section discusses how these navigation widgets facilitate in the generation of the client side MVC components for an insert business transaction. The generation of server-side MC components are not discussed for the insert operations since the principles are similar that of server-side search operations.

An insert business transaction may involve one or more physical operations. An insert business transaction involving a single physical operation occurs when data from one *DFY Container* is committed to the storage using the “**commit inserts**” annotation on the navigation widget without using intermediate “**select for insert**” or “**temporarily store for insert**” annotated navigation widgets in the process. Examples of such operations are discussed in Section 4.2.1 for the creation of “Travel” or “Administrator” entities.

The auto-generation of MVC-MC components for insert business transaction involving many physical insert transactions is discussed in this section since it represents a complex case. An insert business transaction can be specified in five ways in the mock-up:

- a) By linking existing record(s) with new record(s).
- b) By linking existing record(s) with other existing and new record(s)
- c) By linking existing record(s) with other existing record(s).
- d) By using existing record containing multiple entity types but not linking with any other record(s).
- e) By linking a series of new records

Correspondingly the auto-generator should be able to identify the above-mentioned scenarios while deriving the required MVC-MC components for the insert transaction.

Scenario (a) is used to discuss the MVC-MC component generation in detail since it involves both existing and new records. Additional features required for the other scenarios are then discussed generally. Figure 17 illustrates scenario (a) where an existing “Travel” record is required to be booked by a new “Customer”. This figure is a sub-part of the running case study illustrated in Figure 12. Figure 33 is the corresponding the sequence diagram illustrating the insert business transaction

process. It uses MVC-MC components with respect to “insert” operations. The transaction starts when the user generated event associated with “**select for insert**” annotated navigation widget is triggered on clicking the “booking” button in the “Available Deals [3]” *Search Result Container*. This is shown as message 1 in Figure 33. It may be noted that “CSCSearch” and “CSMSearchSelections” are already available on the client-side since they are loaded during the search operation (see Section 5.2.1). This results in the selected existing “Travel” record to be stored on the client side (message 2) using “CSMSearchSelections” for managing search result selections. Figure 32 is the auto-generated “CSMSearchSelections” code segment representing the “**select for insert**” event handler for storing user selected data from search results.

The algorithm for a “**select for insert**” event handler is: Get the CSM data-set bound to the clicked item on the CSV and store it on the client side until the insert business transaction is complete.

In Figure 32 “pageData2” is an attribute of “CSMSearchSelections” which is data-bound to the *data view widgets* in “Available Deals[3]” *Search Result Container*. From the auto-generated view of the *Search Result Container* (see Figure 30) it can be observed that each search result data-set is displayed in an html table row. The row number of the selected result set is found using JQuery and since the data in the row is bound to “pageData2”, the selected existing data-set is readily available for client-side storage. In Figure 32 *localStorage* represents the client-side storage.

```
function Booking2Handler(event) {
    var $row= $(this).closest('tr');
    var $table = $row.parent();
    var rowsInTable = $table.children().length;
    var clickedRowIndex = $row.index();
    var actualRowIndexOfItem = Math.floor(clickedRowIndex/2);
    var vmItem2=viewModel.pageData2()[actualRowIndexOfItem];
    var local={};
    if(typeof(Storage) !== 'undefined') {
        if(localStorage.OrderDeal){
            //append the selected data to the target as well
            for (var prop in vmItem2){
                local[prop]=vmItem2[prop];
            }
            localStorage.setItem('OrderDeal', JSON.stringify(local));
            var url= 'http://localhost/Balsamiq_VINCIexampleSuccess/post/nextPage/OrderDeal/';
            $(location).attr('href',url );
        }
    }
}
```

Figure 32: Snapshot of the auto-generated CSM code segment for storing user selected search results data-set

The selected data is not sent to the server-side for temporary storage to reduce time loss due to client-server communication in the MVC-MC architecture. Once an

existing “Travel” record is selected, “CSCSearch” requests “CSMSearchSelections” to store it in client-side storage to be used by the “Order Deal” page. Then it loads the “Order Deal” page as shown by message 3 in Figure 33. This results in the code for “CSCInsert”, “CSVInsert” and “CSMInsert” to be loaded in the browser. The algorithm for the derivation of these client-side components is not provided here since it follows the same principles as discussed in Section 5.2.1. This triggers the “on load” default event service (discussed in the meta-model in Section 4.3) to invoke message 4 to read the selected “Travel” record from client-side storage and to invoke message 5 to display it in the “Selected Order Details” *Data View Container*. As a result, a “Travel” record representing an existing record gets displayed for potential linkage with new (customer details) records. The details of how this is linked to new records are discussed next.

Figure 34 is the auto-generated view of “Selected Order Details” for the display of the existing “Travel” record. The structure of the “Selected Order Details” *Data View Container* is like its corresponding mock-up except that it is used with Knockout.js. Since it uses the previously described (refer to *Search Result Container* discussion in Section 5.2.1.1) Knockout.js technologies for data-binding with the CSM attributes, no further explanation of the figure is provided here.

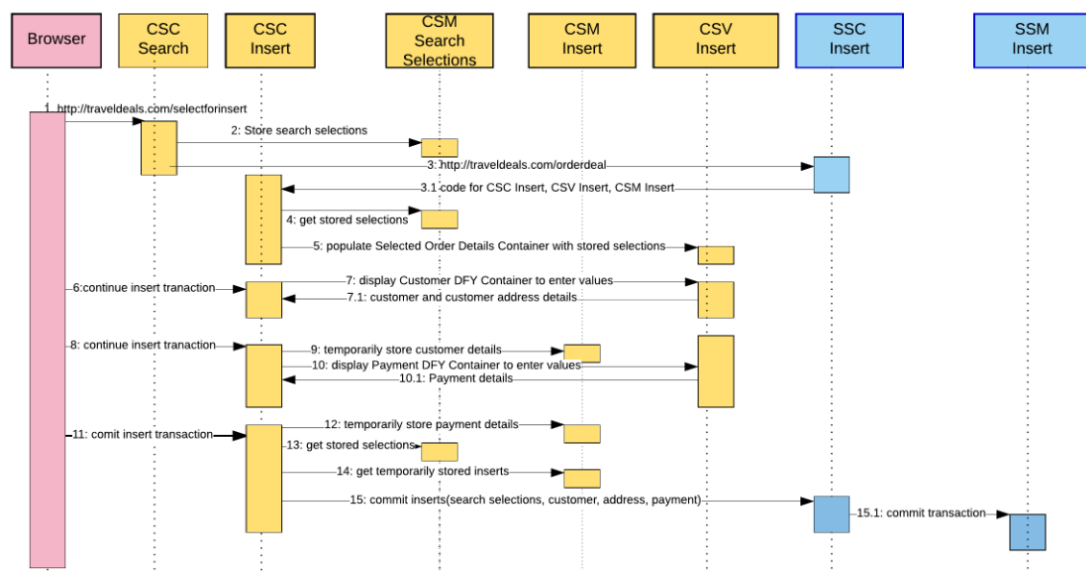


Figure 33: Sequence diagram highlighting component interaction in an insert business transaction

```

<div id="OrderDeal_1-DIV">
<div class="content1"><B>Selected Order Details</B>
| <table data-bind="template: {name:'resultRowTemplatel', foreach:listItems1}">
| </table>
</div>
<script type="text/html" id="resultRowTemplatel">
<tr>
<td>DealDetails</td>
<td>ExpiryTime</td>
<td>Price</td>
</tr>
<tr>
<td valign="top"><textarea rows="3" cols="14" data-bind="text: Travel_DealDetails" readonly> </textarea></td>
<td valign="top"><input type="text" data-bind="value: Travel_ExpiryDate" readonly /></td>
<td valign="top"><input type="text" data-bind="value: Travel_Price" readonly /></td>
</tr>
<tr>
<td><button id="Cancel1" name="button" type="submit" value="Cancel">Cancel</button>
</td>
<td><button id="Confirm1" name="button" type="submit" value="Confirm">Confirm</button>
</td>
</tr>
<br/>
</script>
</div>

```

Figure 34: Snapshot of auto-generated CSV of Selected Order Details container in Travel Deal web app

When the user confirms the continuation of the insert process via message 6, “CSCInsert”, the *Client-Side Controller* for insert invokes message 7 to display the “Customer” *DFY Container* for data entry of new customer record. “Selected Order Details” *Data View Container* and “Customer” *DFY Container* are linked via the “temporarily store for insert” annotated navigation widget in Figure 12 and the user’s (click) confirmation associates the existing “Travel” record with the newly entered “Customer” details. This is an example of creating links between an existing record and a new record as a part of an insert business transaction. Further details of how such relationship links are established is discussed in Section 5.1.3.3. The “Customer” details and the linked “Travel” details are stored on the client-side using CSMInsert (message 9) when the user via message 8 confirms the entered details. Any redundant data is removed from client-side storage during this process. Next, CSCInsert invokes message 10 to display the “Payment” *DFY Container*. “Customer” *DFY Container* and “Payment” *DFY Container* are linked using the “temporarily store for insert” annotated navigation widget. As discussed in Section 5.2.1 when a user clicks a navigation widget with “temporarily store for insert” annotation, it predominantly causes storage on the client-side but will cause sever-side storage if file uploads are required in the process.

Hence the algorithm for a “temporarily store for insert” event handler is: Get the CSM data-set bound to the clicked item on the CSV, do a file upload operation if necessary and store the data on the client-side until the insert business transaction is complete.

Finally, message 11 is invoked when the **“commit inserts”** user generated event is triggered causing messages 12 to 15 for “reading of the client-side stored records and their relationships” and for sending the data to the SSC for managing an insert business transaction in a transactional mode. That is, if any record is not successfully stored on the server, all the operations are in the transaction are aborted and the user is informed.

Scenario (b): Linking existing record(s) with other existing and new record(s) is not illustrated in the running example in Figure 12. However, this case is discussed in Section 5.1.3.3 during the algorithm design for entity-relationships from **“temporarily store for insert”** links. Specifically, Figure 26 in Section 5.1.3.3 illustrates the insert process where an existing “Customer” books an existing “Travel” deal and enters new “Payment” details for the booking. In Figure 26 the “Travel Deals” page searches a “Travel” entity and the “Order Deal” page links a selected “Travel” entity with an existing “Customer”. Here the existing “Customer” is found by another search operation. The selected “Customer” is then linked to a new “Payment” record. The only difference between this scenario and scenario (a) is that no new “Customer” record is created in this case.

On comparing the mock-ups for the two scenarios it is evident that the relationships among the records involved in an insert transaction can be established from the “source” and “target” container of the **“temporarily store for insert”** annotated navigation widgets. From Figure 26 it is seen that the target of **“temporarily store for insert”** can be a *Search Container* in which case it is required to find the corresponding *DFY Container* of the selected search result data-set. The discussion in Section 5.1.3.3 covers how to identify the corresponding *DFY Containers* in such circumstances. Hence the only change in the algorithm will be to replace any targeted *Search Containers* of **“temporarily store for insert”** annotated navigation widgets with the *DFY Containers* from the selected search results.

Scenario (c) is a simpler version of scenario (b) because it does not involve new records while linking. An example of this is same as Figure 26 but without the “Payment” *DFY Container* and with the **“temporarily select for insert”** link in the

“Selected Customer” container being replaced by a **“commit inserts”** link. Since it is a simpler version, there will be no change in the algorithm.

Scenario (d) is the simplest scenario because it assumes each search result data-set consists of multiple types of records. An example of is discussed in Section 4.2.2 and illustrated in Figure 16. It may be observed that each search result data-set in Figure 16 is made of “Customer” and “Travel” entities that match the search criteria. Hence as in the previous scenarios, on selecting a search result set via the **“select for insert”** annotated navigation widget in the “Selected Customer and Travel” *Data View Container*, the user may complete the transaction without using any intermediate **“temporarily select for insert”** annotated navigation widgets. Such a transaction will result in an entity-relationship being established between the entity types in the “Selected Customer and Travel” *Data View Container*.

Finally, scenario (e) represents the case where a series of *DFY Containers* are linked via **“temporarily store for insert”** annotated navigation widgets to store new records. Since no existing records are encountered in the process, no *Search Containers* are used. A new record is stored in the database table corresponding to each *DFY Container* and entity-relationships are established among the tables associated with the linked *DFY Containers*.

As discussed earlier an insert transaction may also involve file uploads. If a file upload operation is involved in any *DFY Container*, an AJAX call is immediately made to the server to upload the file to a temporary location and its URL is stored in the client side. This action is undertaken since most browsers allow limited client-side storage size.

From the above scenarios, a generic algorithm for an insert operation is derived:

The containers associated with an insert operation can be thought of as a linked list of containers from the mock-up and they can be in one of the four forms shown in Figure 35. The linked list will be of form 1 (in Figure 35) if only one new entity is used in the transaction. An example of this is the “Travel” *DFY Container* in the “Add Travel Deal” page in Figure 12. The linked list will be of form 2 if a data set consisting of two

or more types of existing entities need to be linked as a part on insert transaction. An example of this is the “Search Result” container being linked to the “Selected Todo Task” container in the “Search User and ToDo” page in Figure 20, for an existing “User” entity to be linked to an existing “Todo Task” entity. The linked list will be of form 3 if one or more existing entity types are linked to either new entities types or other existing entities on the same web page. Finally, the form 4 is like form 3 except that an existing entity is selected from one web page and the other entities can be from a series of other web pages. An example of this is the linking of a selected “Travel” entity from the “Travel Deals” page to the “Customer” and “Payment” entity in the “Order Deal” page in Figure 12. Once the nodes in the hypothetical linked list is known, traverse through each node in the list and store the associated entity or entities in each node on the client-side along with the entity type to which each node is related. Finally, when the last node is encountered, post all temporarily stored client-side data to the server-side to complete an insert business transaction.

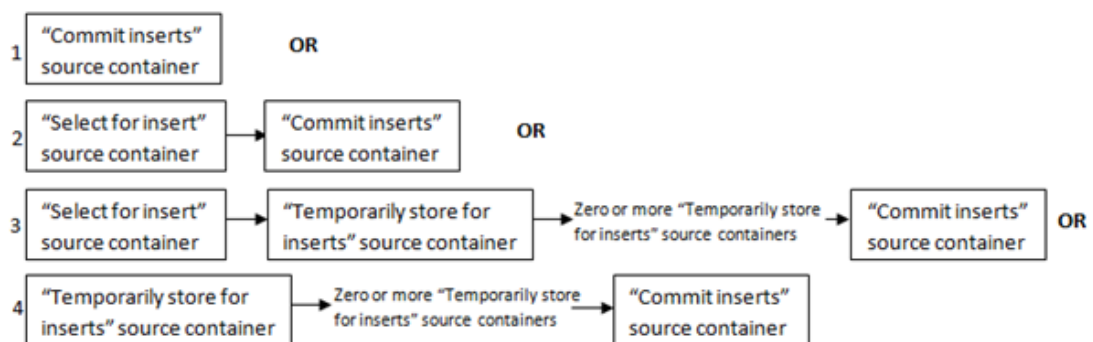


Figure 35: Various forms of linked list of containers possible in an insert transaction

Separate sections on the auto-generation of MVC-MC components for insert business transaction is not provided here since the general logic of identifying the components from the visual mock-up is very similar to that discussed for search operation in Section 5.2.1.2 to Section 5.2.1.5. The detailed version of insert business transaction algorithms is found in Appendix 3. In particular, Appendix 3.1.1 deals with the algorithm for CSC to manage “select for insert” action; Appendix 3.1.2 deals with the algorithm for defining CSM attributes to storing data related to a “select for insert” action; Appendix 3.1.3 discusses how to initialize the CSM attributes with selected data when the user clicks a navigation widget with “select for insert” annotation; and

finally Appendix 3.2 deal with algorithms to manage “temporarily store insert” and “commit inserts” annotations on navigation widgets. This completes the discussion on the answers to RQ 2.1 and RQ2.3 for the auto-generation of client and server-side components for insert operations.

The next section provides answers to research R2.1 and RQ2.3 from the perspective of the auto-generation of client and server-side components for creating reports.

5.4 MVC-MC Generator for Report generation

Generation of reports is an operation that can be performed by client-side components once data is available. Reports are required for making management decisions and are obtained from stored data. This section discusses how the auto-generator derives the MVC-MC components required for report generation. However, the discussion is limited to client-side MVC components only since the generation of server-side MC components follow the same principles as in the search operations for retrieval of data.

From the discussion in Section 4.2.4 it known that the mock-up of “reports” utilizes a short cut notation using only container names to specify the data to be displayed in the report, instead of the expanded version. Specifically, with respect to the “Travel Deals” example the “Booking Confirmation” page in Figure 17 is the mock-up and Figure 18 is the corresponding expanded version of a report view. On comparing the mock-up with the expanded version, it is obvious that a MVC-MC generator needs to be aware of not only the mock-up of the *Report View Container* but also of the *DFY Widgets* in the mock-up of the corresponding *DFY Containers* to generate the expanded view and the data is displayed in read only format in the *Data View Containers*. Secondly it should identify the actual data to be populated in each *Data View Containers*. Moreover, reporting does not require server-side functionality once the data is available on the client side from earlier operation(s). Figure 36 is the sequence diagram for the generation of the “Booking Confirmation” report view page. Here message 0 represents the assumed data that is available on the client side because of the previous insert transaction in the “Order Deal” page (see Figure 17).

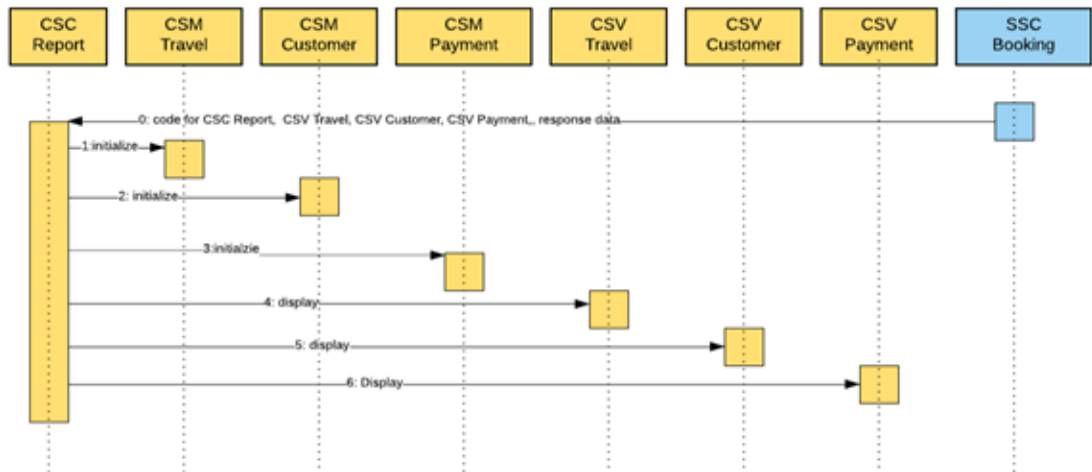


Figure 36: SequencedDiagram highlight component interaction for generation of a report

“CSVTravel”, “CSVCustomer” and “CSVPayment” are the auto-generated CSVs for “Travel”, “Customer” and “Payment” *DFY Containers*. The auto-generated CSV code segments for each “=reference widget” in the “Order Details” *Report View Container* in the “Booking Confirmation” page in Figure 17 is shown in Figure 37, Figure 38 and Figure 39 respectively. In the auto-generated code, “listItem1”, “listItem2” and “listItem3” are attributes of the corresponding CSM objects, namely “CSMTravel”, “CSMCustomer” and “CSMPayment”. The CSVs exhibit the now familiar Knockout.js features discussed in Section 5.2.1. On observing the three views it is apparent a “foreach” feature of knockout.js is required and that for each “foreach” feature a list of data-sets is required in the client-side model. Comparing the similarities in the auto-generated code in Figure 37, Figure 38 and Figure 39 with the auto-generated code for the *Data View Container* within *Search Result Container* in Figure 30 in Section 5.2.1 it can be observed that a *Report View Container* is a special type of *Data View Container*. Furthermore, in the sequence diagram in Figure 36, messages 1 to 3 are automatically invoked on loading of the report page to initialize “listItem1”, “listItem2” and “listItem3” attributes. Once the CSMs are initialized messages 4 to 6 are invoked to display the report using the expanded version described earlier. From the sequence diagram, it is obvious that the definition of the CSC for report only requires passing messages 1 to 3 to CSMs to retrieve the data from their attributes and to pass messages 4 to 6 to CSVs to populate them.

```

<div id="BookingConfirmation1-DIV">
<div class="content1"><B>Selected Order Details</B>
  <table data-bind="template: {name:'resultRowTemplate1', foreach:listItems1}">
  </table>
</div>
<script type="text/html" id="resultRowTemplate1">
<tr>
<td>DealDetails</td>
<td>ExpiryTime</td>
<td>Price</td>
</tr>
<tr>
<td valign="top"><textarea rows="3" cols="14" data-bind="text: Travel_DealDetails" readonly> </textarea>
</td>
<td valign="top"><input type="text" data-bind="value: Travel_ExpiryDate" readonly />
</td>
<td valign="top"><input type="text" data-bind="value: Travel_Price" readonly />
</td>
</tr>
</script>
</div>

```

Figure 37: Snapshot of auto-generated code segment of the CSV for Displaying Selected Travel Deal in Travel Deals web app

```

<div id="BookingConfirmation2-DIV">
<div class="content2"><B>Customer</B>
  <table data-bind="template: {name:'resultRowTemplate2', foreach:listItems2}">
  </table>
</div>
<script type="text/html" id="resultRowTemplate2">
<tr>
<td>Firstname</td>
<td>Lastname</td>
<td>DateofBirth</td>
<td>StreetNumber</td>
<td>StreetName</td>
<td>Suburban</td>
<td>Country</td>
</tr>
<tr>
<td valign="top"><input type="text" data-bind="value: Customer_Firstname" readonly /></td>
<td valign="top"><input type="text" data-bind="value: Customer_Lastname" readonly /></td>
<td valign="top"><input type="text" data-bind="value: Customer_DateofBirth" readonly /></td>
<td valign="top"><input type="text" data-bind="value: Address_StreetNumber" readonly /></td>
<td valign="top"><input type="text" data-bind="value: Address_StreetName" readonly /></td>
<td valign="top"><input type="text" data-bind="value: Address_Suburban" readonly /></td>
<td valign="top"><input type="text" data-bind="value: Address_Country" readonly /></td>
</tr>
</script>
<button id="UpdateCustomerDetails2" name="button" type="submit" value="UpdateCustomerDetails">Update Customer Details</button>
</div>

```

Figure 38: Snapshot of auto-generated code segment of the CSV for Displaying Customer Details in Travel Deals web app

```

<div id="BookingConfirmation3-DIV">
<div class="content3"><B>Payment</B>
  <table data-bind="template: {name:'resultRowTemplate3', foreach:listItems3}">
  </table>
</div>
<script type="text/html" id="resultRowTemplate3">
<tr>
<td>CardNumber</td>
<td>CV_CVV</td>
<td>CardExpiryDate</td>
<td>CardType</td>
</tr>
<tr>
<td valign="top"><input type="text" data-bind="value: Payment_CardNumber" readonly /></td>
<td valign="top"><input type="text" data-bind="value: Payment_CV_CVV" readonly /></td>
<td valign="top"><input type="text" data-bind="value: Payment_CardExpiryDate" readonly /></td>
<td valign="top"><input type="text" data-bind="value: Payment_CardType" readonly /></td>
</tr>
</script>
<button id="UpdatePaymentDetails3" name="button" type="submit" value="UpdatePaymentDetails">Update Payment Details</button>
</div>

```

Figure 39: Snapshot of auto-generated code segment of the CSV for Displaying Payment Details in Travel Deals web app

A general algorithm for the auto-generation of Report View is:

For each “=*reference widget*” notation found in the *Report View Container* define a CSM attribute where the attribute is in the form of a list. Each element of the list is data-bound to a *data view widget* in a CSV. The CSV is got by creating an expanded view of each “=*reference widget*” notation found in the *Report View Container*. This is done by replacing each “=*reference widget*” notation with the *data view widgets* in the corresponding *Data View Containers*. Next define the CSC by first passing messages to appropriate CSMs to initialize the attributes by reading corresponding data from client-side storage. Secondly pass messages to appropriate CSVs to populate the Views with data. Finer details of the client side MVC algorithms for *Report View* is discussed in Appendix 4. Specifically, Appendix 4.1 contains the algorithm for defining CSM for Report View from the mock-up; Appendix 4.2 deals with the CSV definition for *Report View* and Appendix 4.3 focuses on the algorithm for defining CSC for *Report View*.

5.5 MVC-MC Generator for Update operations

An update operation requires client and server-side components. This section provides answers to RQ2.1 and RQ2.3 for the auto-generation of client and server-side components from an update operation perspective. Update operations are always carried on user selected stored data-sets from either *Search Result Containers* or *Report View Containers*. These two cases are considered here while discussing the MVC-MC generator for update.

First consider the case of update of a selected data-set from a *Search Result Container*. As discussed in Section 4.2.5, a selection of data sets for update is triggered when a user clicks a button which is the source of a “**select for update**” annotated navigation widget in a *Search Result Container*. Considering the “Travel Deal” example in Figure 12, when “**select for update**” is triggered two actions occur, firstly the data corresponding to the selected data-set is identified in the CSM, secondly the selected data-set is associated with the target of “**select for update**” annotated navigation widget. The target is a page with one or more *Update*

Containers. This suggests that the CSC for update should have knowledge about the selected data-set and the targeted CSV. Figure 40 is the sequence diagram for the update operations and contains these two pieces of information in the form of “CSMUpdate” and “CSVUpdate” respectively. In the diagram “CSCSearch” is the CSC for the search operation and “CSMUpdateSelections” represents the CSM for managing the client-side storage of selected data-set for future update. The actual update is handled by the CSC for update “CSCUpdate” by passing messages to corresponding CSMs and CSVs for update, namely to “CSMUpdate” and “CSVUpdate”.

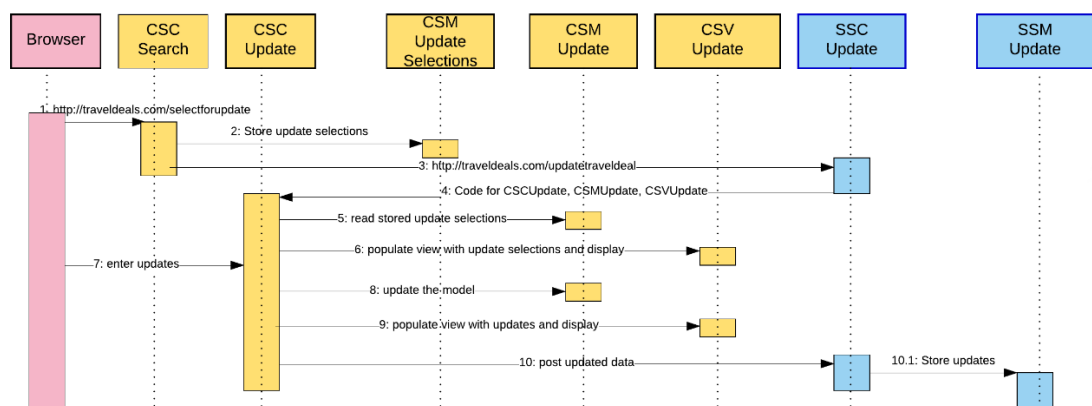


Figure 40: Sequence diagram highlighting component interactions in an update operation

In the sequence diagram when message 1 is passed, selected data-set from the displayed page in the *Search Result Container* is stored on the client-side. It is known from the discussion in Section 5.2.1.1 that this data-set is available in attribute “pageData2” of “CSMSearchResult” (in Figure 30). Figure 41 illustrates how update selections are got from “CSMSearchResult” in the auto-generated code. The last line in Figure 41 picks up a sub-set of “pageData2”, where the sub-set index, “actualRowIndexOfItem”, is got from the clicked row in the *Search Result Container*. Message 2 in Figure 40 stores the sub-set on the client-side. Message 3 and 4 are used to load the update page. A segment of the auto-generated code of this page is shown in Figure 42 wherein “listItems1” is the attribute in “CSMUpdate” for managing of the update in the associated CSM. For example, message 4 reads the previously stored values and initializes “listItems1”. Further “listItems1” is data-bound using Knockout.js features to the update CSV as illustrated by the code in the blue box in Figure 42. Hence when message 6 is passed, the update view is populated

with the selected data-set and displayed. Message 7 represents the user making an update and triggering the “update” in the *Update Container*. Any update in the CSV is automatically reflected in “listItems1” (message 8 and 9) due to the data-binding feature of Knockout.js. Finally, the updated values are posted to the server via message 10.

```
var $row= $(this).closest('tr');
var $table = $row.parent();
var rowsInTable = $table.children().length;
var clickedRowIndex = $row.index();
var actualRowIndexOfItem = Math.floor(clickedRowIndex/2);
var vmItem2=viewModel.pageData2()[actualRowIndexOfItem];
```

Figure 41: Segment of auto-generated code for CSCSearch illustrating selection of entities for update from CSMSearchResult in Travel Deal web app

```
<div id="UpdateTravelDeal_1-DIV">
<form id="UpdateTravelDeal_1-FORM" action="" method="post" autocomplete="on">
<div class="content"><B>Travel</B>
  <table data-bind="template: {name: 'resultRowTemplate1', foreach: listItems1}"></table>
</div>
<script type="text/html" id="resultRowTemplate1">
<tr><td>DealDetails</td><td><input type="text" id="Travel_DealDetails" name="Travel_DealDetails" data-bind="value: Travel_DealDetails" pattern="[a-zA-Z0-9_-]*[\w]{1}" title="The pattern rule (length>0) is not matched." /></td></tr>
<tr><td>Image</td>
<td><a data-bind="attr: { href: Travel_Image, title: Travel_Image }">Click to view</a></td>
<td>Click to change </td><td><input type="file" size="100" name="Travel_Image" id="Travel_Image" data-bind="text: Travel_Image"></td></tr>
<tr><td>ExpiryDate</td><td><input type="date" id="Travel_ExpiryDate" name="Travel_ExpiryDate" data-bind="value: Travel_ExpiryDate"/></td></tr>
<tr><td>Price</td><td><input type="text" id="Travel_Price" name="Travel_Price" data-bind="value: Travel_Price" pattern="^[0-9]*[0-9]*([\.[0-9]+]?|0+\. [0-9]*[1-9][0-9]*$" title="The pattern rule (price > 0) is not matched." /></td></tr>
<tr><td>People</td><td><input type="text" id="Travel_People" name="Travel_People" data-bind="value: Travel_People" /></td></tr>
<tr>
<td><button id="Cancel1" name="button" type="submit" value="Cancel">Cancel</button></td>
<td><button id="Ok1" name="button" type="submit" value="Ok">Ok</button></td>
</tr>
</script>
</form>
</div>
```

Figure 42: Segment of auto-generated Client-Side View code for update in Travel Deal web app

The auto-generator or update management does not restrict the update to a single entity type. Though the above illustrations dealt with update of only Travel deal entity the mock-up of the update page may involve more than one type of entities. This is illustrated in Figure 43 where the “Update Page” updates “Travel” and “Customer” entities in a single update operation. The auto-generation process discussed above is still valid since the update CSM is dependent on the stored data. The stored data in Figure 43 contains both “Travel” and “Customer” (and “Address”) details because data-set in the *Search Result Container* has a combination of these types of entities as discussed previously in Section 4.2.5 . When “select for update” is triggered the selected “Travel” and “Customer” entities will be stored on the client-side to

populate the widgets in the “Update Page” and since the source button of the “**update**” annotated navigation widget is enclosed within the *Update Container* consisting of all data to be updated, the resultant data will be updated on the client-side as well as on the server-side.

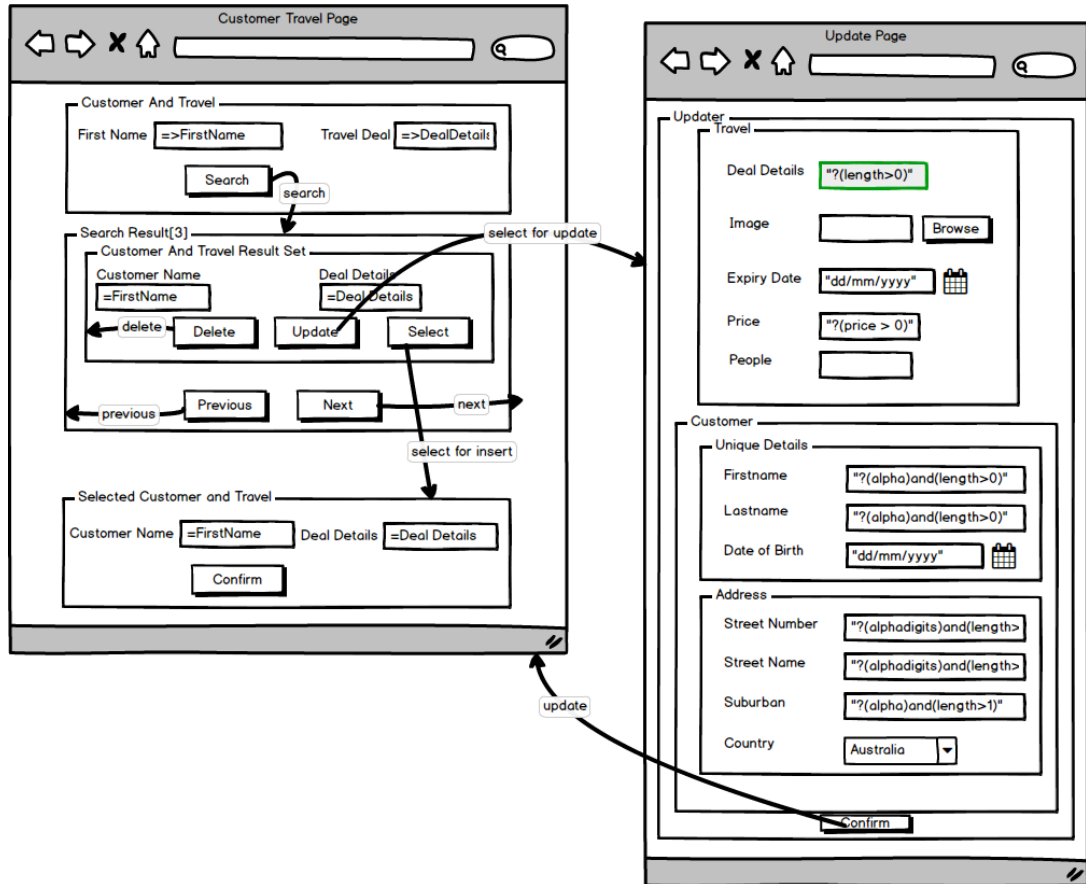


Figure 43:Mock-up segment for update of multiple entity types in a single page

Similar reasoning applies to the other case of update of the data-set in the *Report View Container*. For example, the right column in Figure 19 in Section 4.2.5 illustrates how the source button of the “**select for update**” annotated navigation widget can be used to trigger updates from a *Report View Container*. If any label widget in the *Report View Container* containing text notation of the form “=*reference widget*” where the referenced widget refers to a container that has data of multiple entity types (such as “Customer” entity having a nested “Address” entity) and needs to be updated, then the related data-set is stored on the client-side for update.

In summary, a general algorithm for update is:

Read the stored details required for update from the client-side. Populate the update page with the details read from the client side. Finally send the updated details to the server-side where all the entity types involved are operated in transaction mode to ensure all or none of the entities are updated. Finer details of the algorithm are discussed in the appendix in Appendix 5. In particular, Appendix 5.1 deals with the algorithm for the defining CSC for update, Appendix 5.2 focuses on the algorithm for defining CSV for update; Appendix 5.3 on defining the attributes of the CSM for update and Appendix 5.4 on initializing the attributes of CSM for update.

The next section discusses answers to RQ2.2 and RQ2.3 from a delete operation perspective.

5.6 MVC-MC Generator for Delete operations

The auto-generator manages delete operation very similarly to an update operation except that when the “delete” action is triggered, data in the CSM is deleted and since the CSM is data-bound to the CSV, the corresponding view is also deleted and finally the server-side data is deleted in the form a soft update. That is each delete operation on the server-side is simply marked as deleted instead of a physical delete operation. Furthermore, delete operations like update operations are always carried on selected data-sets in a *Search Result Container*. The deletion of the selected data-set on the client-side requires two actions to be involved: i) identification (and subsequent deletion) of selected data-set in the CSM and ii) an AJAX call to delete the same on the server-side. Since this is analogous to update operations no additional details are provided here.

5.7 Management of the auto generated application

The auto-generated application should be easy to manage. This section discusses how it is easy to manage the auto-generated application due to the simplistic structure of communication between client-server components. From the discussion

on the sequence diagrams in Figure 28, Figure 33, Figure 36 and Figure 40 it can be observed that five sets of components namely CSCs, CSVs, CSMs, SSCs and SSMs, are sufficient for all RIA operations. In addition, typically, users perform five types of operations in any web application regarding one or more entities: inserting, searching, deleting, updating and reporting. Hence, an auto-generation of a RIA can be considered by the auto-generation of the five types of components for each of the five types of behavioural operations. Further we can generalize that a RIA based on MVC-MC architecture has the following communication pattern for any type of server-side data request, once the initial page is loaded:

(1) CSC to SSC to SSM, (2) CSC to CSM, and (3) CSC to CSV.

Moreover, if the client does not require server-side data, only communication patterns in (2) and (3) are relevant. Such a clear communication pattern for every type of request also reduces the design complexity of the auto-generator. For example, since the pages are always generated on the client-side with no distribution of View generation responsibilities between the client and the sever-side, the need for a server-side View component is eliminated. Secondly, the generated application will have good response times since client-side components are not monolithic in nature. That is, the auto-generator generates distinct client side MVC components for each type of operation and the initial load times of these MVC components from the server-side will be low. The response times for subsequent client-side only service requests will be better than RIAs that are serviced by distributed View architectures since no server-side-trips are necessary for View requests. Thirdly, the richness of the UIs will be good because of the usage of JavaScript libraries such as JQuery and Knockout.js. Fourthly, since the generated application does not use any proprietary software such as Flash, such applications may be economically feasible to many small business enterprises.

This concludes the discussion on the algorithms for the auto-generation of the database schema and the MVC-MC components for the anticipated set of behaviours in SME applications. The next section provides a summary of the components required for all pages in the Travel Deal case study.

5.8 MVC-MC components of the case study application

Several MVC-MC components are generated for each page in the web application. Table 7 lists the various components generated for the case study in Figure 12.

Table 7: MVC-MC Components for the example case study

Web Page Name	Client-Side Components			Server-Side Components	
	Model	View	Controller	Controller	Model
Travel Deals	Pre-search, Search result, Search selections	Search, Search result	Search	Search, Delete	Travel
Login	Administrator	Login	Login	Login	Administrator
Order Deal	Travel, Customer, Address, Payment	Insert: Travel, Customer, Payment	Insert	Insert	Insert: Travel, Customer, Payment
Add Travel Deal	Travel	Insert: Travel	Insert	Insert	Travel
Add Administrator	Administrator	Administrator	Insert	Insert	Administrator
Booking Confirmation	Travel, Customer, Address, Payment	Travel, Customer, Address, Payment	Report	Report	Travel, Customer, Address, Payment
Update Travel Deal	Update: Travel	Update: Travel	Update	Update	Travel
Update Customer Details	Customer	Customer, Address	Update	Update	Customer, Address
Update Payment Details	Payment	Payment	Update	Update	Payment
Deal Management	-	Deal Management	Navigate	-	-

It may be observed that each page on the client-side is designed to have a single CSC and one or more CSVs and CSMs whereas the server-side may include many SSCs and SSMs for the same page. A single CSC is required for each page because, as discussed

in Section 5.2.1.4, once a page is loaded on the browser the CSC has the responsibility of registering the event-handlers for all user-initiated and automatically initiated events in the page. These event handlers are identified by scanning for annotated navigation widgets on the mock-up of the page. However, a SSC is required for each type of behavioural request from the page. Each URL requested from a client-side page contains information such as the type of behaviour, entity types and the relevant attributes of the entities. SSCs are defined for each type of behaviour and since each page can have more than one type of behaviour correspondingly a page can contain more than one SSCs.

The methods within each component are not shown in the above table but the important ones are identifiable from the annotations of the navigation widgets in Figure 12. For example, the “Travel Deals” page needs a search method for performing a search. Moreover, flavours of the search method are required in the various MVC-MC components dealing with the “Travel Deals” page. For example, the CSM will require a search method to filter already loaded server-side response data-sets when the user clicks the “Previous” and “Next” button in “Available Deals [3]” *Search Result Container* in the “Travel Deals” page. Correspondingly, the CSC is required to invoke a search method in the CSM (see message 8.1 in Figure 28). On the other hand, when the user does a new search, the CSC invokes a search method in the SSC (see message 5.1 in the figure). The SSC will then invoke a search method (message 5.1.1) on the SSM component to retrieve the data from the database.

Furthermore, as discussed at the beginning of this section, an event handler definition is required for each event type triggered by buttons associated with annotated navigation widgets. Hence, with respect to the “Travel Deals” page, a “Search” button event handler will be defined. Similarly, a “Previous” button event handler and a “Next” button event handler will be defined to manage the search results. Each event handler is defined as part of the CSV definition for the container hosting the source button of the annotated navigation widget in the mock-up and its logic is derived from the annotation type along with the mock-up of the source and target containers (or page). In addition, any multi-valued widget such as a combo-box widget will result in the requirement of a method to pre-populate itself with unique

values from the database fields associated with its *“look-up widget”*. Such a method will exhibit search behaviour since it looks-up data based on a criterion. For example, in Figure 28, “CSM PreSearch” will require a method to pre-populate the values in the “Price” drop down widget in the “Travel Deals” page. Here the search criterion is to search for unique prices. The runtime search page generated for the “Search Deals” *Search Container* in the “Travel Deals” page is shown in Figure 44. This page is generated by a CSV (for example, see message 4.1 in Figure 28) when the URL of the search page is typed in the browser (message 1 in Figure 28). However, before the page is generated, the services of the SSC and the SSM are used to send a file containing the desired client-side MVC components of the search page. As implied from the above discussion, the file containing the desired information is auto-created from the visual model of the search page. Once the file is loaded in the browser, the CSC for search takes control of the operation to invoke the services of the CSV to render the “Search Deals” *Search Container* in the “Travel Deals” page. At this stage, the user interacts with the page (shown by message 5 in Figure 28) to enter the search details. Now on all interactions with the user are managed by the CSC of this page. An example of the resultant “Travel Deals” page is shown in Figure 45. The result section in this page is rendered by another CSV component, which is loaded during the first URL request to the page. Any new search requests require client-server communication to retrieve new data from the server.

Once the response to a search has been loaded from the server, only part of data will be rendered based on the user’s choice. The CSC uses the services of the CSM and CSV to render the page with the appropriate data. This is shown by messages 8, 8.1, and 8.2 in Figure 28. Similarly, other pages of the running example are auto-generated by MVC-MC components but have not been discussed here since the principles are similar. Once the system is found to be functionally viable from the researcher’s perspective, it was trialled by a BA, the details of which are provided in the next section.



Figure 44: An instance of the auto-generated Travel Deals page before the search operation

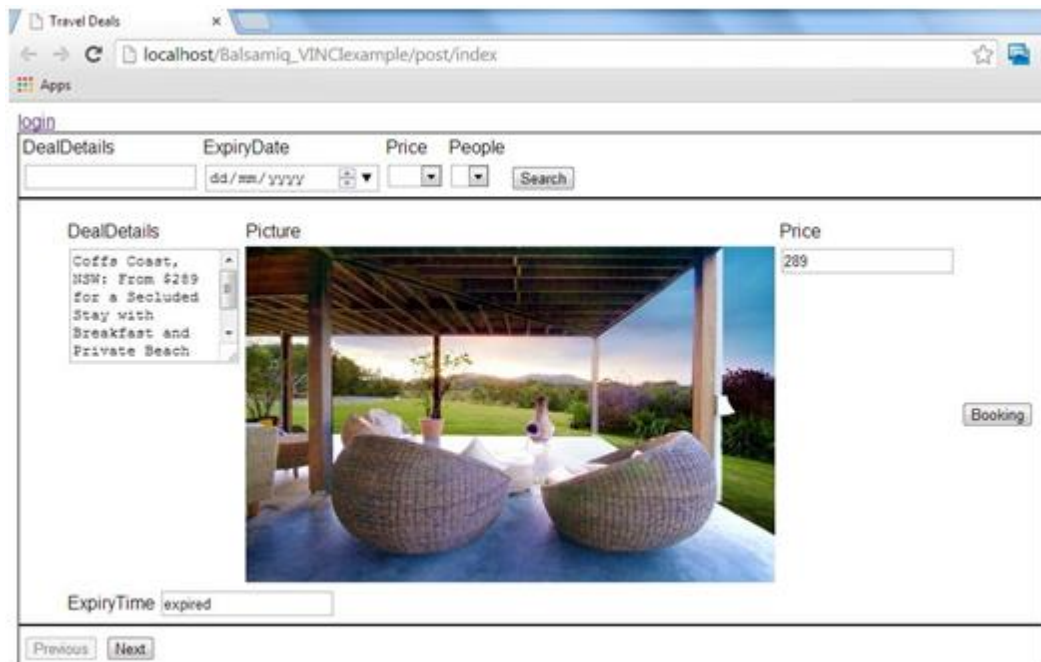


Figure 45: An instance of the auto-generated Travel Deals page after the search operation

5.9 Trial evaluation of the design

A trial evaluation of the tool was conducted as a part of the design process to ascertain whether it is ready for actual field testing by a group of BAs. Functional (black box) and structural (white box) testing evaluation methods were used during the trial test run of the completed system by a usability tester. The usability tester conducting the trial run was a professional Business Analyst with a master's degree in Computer Science. During the trial, the tester designed a mock-up for a SME test case written by him. However, the auto-generator was not able to produce the expected outcomes because it was found to be inflexibly designed to satisfy the type of behaviours not envisaged in the Travel Deals example. The inflexibility was because the auto-generator attempted to address all types of behaviour together in a single

pass of the mock-up model. Consequently, managing unforeseen scenarios in the mock-up design was a challenging task. For example, an activity in the Travel Deal case study is for an existing entity (such as a Travel Deal) to be linked to a newly created entity such as Customer; however, the trial case study required only pre-existing entity types to be linked. The old design of the auto-generator was not capable of doing this without making major changes to the existing code. An important lesson learnt from the usability test of the trial case study was that the design should be flexible to manage future changes seamlessly. This resulted in a major redesign and re-development of the auto-generator in a three-month effort. The re-designed artifact was able to adapt to new business activities because it considers each type of behaviour in a new pass of the mock-up model during the auto-generation of the web application from the mock-up. This means existing algorithms of the auto-generator does not get affected if the auto-generator needs to address new features in the future.

6 VALIDATION

The mock-up language and the auto-generating tool should be useful to Business Analysts as software developers of SME applications. So, evaluation of the design is necessary to ensure that it's functional and usability considerations are met. Hevner, March and Ram (2004) have suggested several evaluation methods of the design artifact in DSR in IS. Table 8 highlights the various methods that can be used for the evaluations.

Table 8: DSR evaluation methods (Hevner, March & Ram 2004, p.86)

Design Evaluation Method	Description
1. Observational	<p><i>Case Study</i>: Study artifact in depth in business environment</p> <p><i>Field Study</i>: Monitor use of artifact in multiple projects</p>
2. Analytical	<p><i>Static Analysis</i>: Examine structure of artifact for static qualities (e.g., complexity)</p> <p><i>Architecture Analysis</i>: Study fit of artifact into technical IS architecture</p> <p><i>Optimization</i>: Demonstrate inherent optimal properties of artifact or provide optimality bounds on artifact behaviour</p> <p><i>Dynamic Analysis</i>: Study artifact in use for dynamic qualities (e.g., performance)</p>
3. Experimental	<p><i>Controlled Experiment</i>: Study artifact in controlled environment for qualities (e.g., usability)</p> <p><i>Simulation</i>: Execute artifact with artificial data</p>
4. Testing	<p><i>Functional (Black Box) Testing</i>: Execute artifact interfaces to discover failures and identify defects</p> <p><i>Structural (White Box) Testing</i>: Perform coverage testing of some metric (e.g., execution paths) in the artifact implementation</p>
5. Descriptive	<p><i>Informed Argument</i>: Use information from the knowledge base (e.g., relevant research) to build a convincing argument for the artifact's utility</p> <p><i>Scenarios</i>: Construct detailed scenarios around the artifact to demonstrate its utility</p>

This research will use a combination of some of the methods highlighted in Table 8. Specifically, the methods used are: experimental, testing and descriptive. Within the experimental methods, controlled experiments are performed by employing usability inspectors to ascertain the usability of the mock-up language and the generated application. During the controlled experiments, functional and structural testing methods are used to routinely debug and fix errors. The descriptive evaluation method is used to make informed arguments regarding the acceptance of usability test results. This chapter discusses how some of these methods are employed to ensure the usefulness of the language and the tool is validated from a BA perspective. That is, it provides answer to the third research question, “How is the auto-generating tool validated”? Chapter 3 considered this question by separating it into three sub-questions: “How is usability of the mock-up language validated? How is usability of the auto-generated RIA validated? How is the usability of the tool validated?” These questions are answered as outputs of activities in block 1d, block 1e and block 1f in the research plan shown in Figure 11.

Usability testing of the mock-up language and the auto-generated application is considered based on ISO standards for usability testing for software solutions. Specifically, ISO/IEC 9126 standard identifies usability testing as validating the *external-quality* and *quality-in-use* (Casteleyn et al. 2009; Molina & Olsina 2008). Becker and Olsina (2010) and Lew et al., (2012), provide in-depth details of usability measurement and evaluation (M&E) processes using a framework called C-INCAMI. C-INCAMI provides an integrated approach for M&E based on ISO 15939 standards for software measurement process and the ISO14598-5 standard regarding the process for evaluators. In addition, the auto-generated application needs to be tested for functional correctness as well. Hence this chapter is organized as follows: Section 6.1 introduces the usability validation principles and the C-INCAMI framework for usability testing. Section 6.2 discusses a validation plan for employing the C-INCAMI framework. Section 6.3 discusses usability testing of the mock-up language using the C-INCAMI framework. Section 6.4 discusses usability testing of the auto-generated applications. Section 6.5 discusses testing functional correctness of the auto-

generated applications. Finally, Section 6.6 provides an analytical discussion on the usability validation of the auto-generating tool as an integrated system.

6.1 Usability validation concepts and the C-INCAMI framework

Software applications are considered to be usable by validating their *external quality* and *quality-in-use* metrics (Casteleyn et al. 2009; Molina & Olsina 2008). This section introduces the principles of usability validation and how the C-INCAMI framework can be used to follow a systematic usability validation process. *External quality* reflects a black-box testing model mainly focussing on product performance and reliability in an environment simulating the actual environment as far as possible. *Quality-in-use* reflects the perception of quality with respect to the *effectiveness*, *efficiency*, *learnability* and *satisfaction* that “actual” users gain while interacting with the application in a real user environment (Casteleyn et al. 2009; Lew et al. 2012). *Effectiveness* refers to how many errors were made by the users because of inadequate or difficult to use usability features of the system. *Efficiency* refers to the resources (or time) required to complete the task using the tool, *learnability* refers to the degree to which the users can learn efficiently and effectively to achieve a task and *satisfaction* is how they felt about the task, that is how happy or how frustrated they are. Five evaluators are found to be sufficient to elicit more than 75% of usability problems (Casteleyn et al. 2009; ‘Foundations of UX: Usability Testing’ 2015; Nielsen & Levy 1994). This is shown by the graph in Figure 46.



Figure 46: Usability problems found versus number of testers

Figure 46 illustrates that adding more than 5 usability testers does not significantly increase the benefits of finding more validation errors. That is usability testing is not performed to find statistical significance where large numbers of participants are necessary to yield better confidence intervals to predict the actual number of users who'd have that problem in real life. Rather usability testing is an attempt to identify and fix ease of use barriers to people using the tool. In usability testing the aim is not to find how many people have the problem, rather it is to find if a problem exists and what triggers it so that the developer can fix it.

Becker and Olsina (2010) and Lew et al., (2012), provide in-depth details of a usability measurement and evaluation (M&E) processes using a framework called C-INCAMI. C-INCAMI provides an integrated approach for M&E based on ISO 15939 standards for software measurement process and the ISO14598-5 standard regarding the process for evaluators. Further details of the C-INCAMI process are presented in the following sub-section.

6.1.1 C-INCAMI framework for usability testing

The C-INCAMI process identifies five activities to carry out usability testing: These include, defining the testing requirements, designing the measurement metrics, designing the evaluation indicators, implementing the measurement and finally analysing and reporting the evaluation. These activities are discussed in Section 6.1.1.1 through to Section 6.1.1.5.

6.1.1.1 Defining the testing requirements

In this activity, the terms and the context in which the testing is performed are defined. The definition of the testing requirements activity includes four sub-activities: *establishing the information needed for the testing, specifying the testing context, designing the testing tasks and selecting a concept model* for the test.

Establishing the information needed for testing includes defining the *purpose*, defining user *viewpoints*, establishing the *object* and defining the *focus concept* to be assessed.

Testing context specifies the environment in which the test is conducted for potential future comparisons. This may include specifying the computing devices in which the test is conducted, back-ground information about the testers and or descriptions of user-oriented application tasks.

Testing tasks are selected and specified during the “*Design the testing tasks*” activity. Here commonly performed tasks are selected for testing so that there is most to gain when changes for improvements are recommended. Furthermore, uncommon tasks are avoided because it is difficult to collect enough data repeatedly and consistently (Lew et al. 2012).

Concept model identifies the high level calculable concepts such as *effectiveness in use, efficiency in use, learnability in use and satisfaction in use* which are defined in ISO 9126-1 and ISO 25010 standards (Lew et al. 2012; Covella & Olsina 2006).

These definitions are represented below:

Effectiveness in use: “The degree to which specified users can achieve specified goals with accuracy and completeness in a specified context of use” (Lew et al. 2012, p.313).

Efficiency in use: “The degree to which specified users expend appropriate amounts of resources [such as time, cost] in relation to the effectiveness achieved in a specified context of use” (Lew et al. 2012, p.313).

Learnability in use: “The degree to which specified users can learn efficiently and effectively while achieving specified goals in a specified context of use” (Lew et al. 2012, p.305).

Satisfaction in use: The degree to which specified users are satisfied in a specified context of use (Molina & Olsina 2008)

In this thesis, *learnability in use* will not be validated since its measurement would necessitate many iterations of usability testing which is not possible due to time constraints. Hence further discussions are provided on *learnability in use*.

Once the important user tasks and the concept model are defined a *quality in use* requirements tree is instantiated. The *quality in use* requirements tree contains leaf nodes and branches. The leaves indicate elementary sub-tasks and branches represent group level task values to be measured.

The next step in the C-INCAMI process is to design the measurement.

6.1.1.2 Designing the measurement metrics

This activity in the Measurement and Evaluation (M&E) process identifies suitable metrics regarding the testing requirements. A metric is the defined measurement or calculation method and the measurement scale (Molina & Olsina 2008). The metric may be designed by experts in the field if it is not feasible to get it from the organization in which the artifact will be used. For example, the attribute “add new UI elements to an existing page” may be defined as the “the ability to add new UI elements to an existing page” and a corresponding metric may be “Degree of the ability to add a new UI element”. The measurement scale for this metric may be specified as three categories in an ordinal (or any appropriate) scale, namely: category 0 for no ability or rare ability of adding a new UI element by the user, category 1 for partial ability to indicate better ability than category 0, representing cases where addition is possible almost every time but not always and category 3 for complete ability. Similarly, other metric specifications need to be documented for each attribute to be measured for evaluations. Then the measures are mapped to

indicator values to assess satisfaction level achieved for each attribute. Hence indicators are required to be designed to interpret the metric's value of an attribute. This is discussed in the next section.

6.1.1.3 Designing the evaluation indicators

Evaluation indicators help in interpretations of usability testing results and consequent decision making. An evaluation indicator is defined as the calculation method and the scale to provide an estimate or evaluation of a calculable concept with respect to defined information needs (Covella & Olsina 2006). Designing the evaluation indicators activity in the M&E process entails defining *elementary* and *partial* and *global indicators*, optionally defining a calculation method and finally identifying a scale. *Partial indicators* represent the sub-characteristics in a *concept model*. An *elementary indicator* does not depend on other indicators to evaluate or estimate a calculable concept while a *global indicator* is derived from other indicators to estimate the calculable concept identified in step one of the C-INCAMI process. A *calculation method* is required in the case of *global indicators* or *partial indicators* and depends on the relationships among the group of related attributes and the relative importance of each attribute in the group. Relationship may be termed as mandatory (AND relationship), alternative (OR relationship) or neutral. Also note that a *scale* needs to be defined where a *scale* can be categorical or numerical and represents a set of values using a scale type such as ratio, ordinal, internal etc.

6.1.1.4 Implementing the measurement

The fourth activity of the C-INCAMI process is to implement the measurement. During this activity data collection is performed for direct metrics (*elementary indicators*). In addition, the computation of the values for indirect metrics (*partial and global indicators*) is performed. Finally, the results are documented for analysis.

6.1.1.5 Analyse and report the evaluation

The last activity is to analyse and report the evaluation. In this final activity, the results are analysed to improve those aspects of the artifact which have a low satisfaction rating. All improvements suggested are recorded in the knowledge base. Appropriate design changes are incorporated to the artifact as a part of the Design Cycle of DSR. The artifact may then be evaluated again as a part of a quality improvement process. This completes the general discussion on the C-INCAMI process for usability testing.

6.2 Usability validation plan

Usability testers should know how to perform the validation. The usability validation plan introduces the overall approach taken in this research to perform the validation of the research artifacts using the C-INCAMI framework. Finding bona fide business users to spare time to test an academic research project is a challenge due to financial and time constraints both for the users and the researchers. Hence *external-quality* and *quality-in-use* evaluation of the tool can be conducted by trained *usability inspectors* (Casteleyn et al. 2009; Molina & Olsina 2008). Inspector evaluations examine usability-related aspects of an application to detect violations of established usability principles and then provide feedback to designers about possible design improvements. The training of the inspectors ensures that they have a good understanding of the usage principles and on how they apply to the application being analysed to uncover critical situations where principle violations occur.

This section discusses three issues. Firstly, it introduces the usability testers in Section 6.2.1. Secondly a discussion on how the testers were trained to perform the validations is provided in Section 6.2.2. Thirdly, it introduces three BA specified SME application requirements in Section 6.2.3 for usability testing.

6.2.1 Business Analysts as usability testers

A tool that is meant for Business Analysts would preferably be accepted if it is tested by the BA community. So, BAs are employed as suitable candidates for usability testers in this research. Specifically, services of nine Business Analysts are employed to perform the validations. All the nine testers had qualifications in Information Systems or in Information Technology or in Computer Science. Moreover, six out of the nine testers had between 1.5 years to 30 years of experience working in various capacities as Business Analysts and or System Analysts and or Developers in research and in professional environments. The other three have zero to 1.5 years' experience. Hence most of the testers can be considered to have average to good BA skills and are suitable candidates for usability testing the tool, provided they are trained to use it. The biographical details of the usability testers are available in Appendix 6.1. The next section discusses how the usability testers are trained to use the tool before the validation.

6.2.2 Training usability testers

BAs require to be trained to use the mock-up language and the tool before the testing. In real world situations BAs, will be familiar with some tool to specify the requirements and may not require specialized training regarding its usability. It is known from the discussion in Section 4.5 that Balsamiq tool is used to create the visual model. This section discusses the process of training the inspectors since none of the nine testers were familiar with either Balsamiq tool as a mock-up editor or with the visual mock-up language used in this research. Specifically, Section 6.2.2.1 discusses how the testers were trained on the usage of the Balsamiq tool and Section 6.2.2.2 discusses how they were then trained to use the mock-up language.

6.2.2.1 Training to use the Balsamiq tool

Training helps users to be familiar with new tools and or activities. Similarly, usability testers are required to be trained to use the tool based on which the testing would

be conducted. The testers were trained for an hour on the usage of the Balsamiq tool. The training included a directed task to familiarize with the elements of the Balsamiq tool before conducting usability tests of the mock-up language. The aim of the task was to familiarise with the terminology and usage of the common elements of Balsamiq that are generally required for producing mock-ups of web applications. The training involved creation of a pre-designed mock-up using widgets identified in the meta-model in Figure 21. That is the *usability testers* were not required to know the mock-up language specifications for this activity. Specifically, the focus was on the process of using the features of the Balsamiq tool discussed in Section 4.5 for creating mock-ups. Creating the mock-up can be a tedious process especially when two or more elements are required to be placed on a layout container in a layered manner. Hence a bottom up approach was used where innermost elements of each container are assembled first. Once the *usability testers* were familiarized with the tool, they were then trained to use the features of the mock-up language to build a mock-up in a directed activity, which is discussed in the next section.

6.2.2.2 Training to use the visual mock-up language with Balsamiq

Directed activities are used to familiarize with common tasks associated with the directed activity without being drained by the problem-solving process. Here the directed activity was to train the *usability testers* with the elements of the language for expressing the mock-up of a SME application. The “Travel Deals” example was chosen as the case study for the training. The directed activity was to create the mock-up shown in Figure 12 using the Balsamiq editor in five increments where each increment correspond to the discussion in Section 4.2.1 through to Section 4.2.5. All the participants were able to complete this activity in a three-hour session. On completing this activity, a PowerPoint documentation containing examples of the numerous ways of using the language was provided to the participants as ready reckoner for future use in actual usability testing. The application requirements for actual usability testing are introduced in the next section.

6.2.3 SME Application Case Studies for Usability Testing

The tool should be validated against realistic SME applications. Three out of the nine testers three were randomly chosen to propose a case study each for a SME web application in a domain of their choice. Three SME application requirements were chosen instead of one to ensure that the validation results are justifiable for diverse types of web apps. However, the three proposals were moderated to have similar task complexity across the three case studies. Similar level of complexity is necessary since usability testing is only required to be conducted for the common features of SME applications identified in Section 4.1 across the three case studies. Each case study was randomly assigned to three testers resulting in a total of nine test cases for nine testers. Finer details of the three case studies are available in Appendix 6.2.1 through to Appendix 6.2.3. Specifically Appendix 6.2.1 introduces the requirements for a “Question and Answer” SME application, Appendix 6.2.2 deals with a “Teacher Consultation System” and Appendix 6.2.1 outlines the requirements for a “Patient-Dietician Consultation System”. This completes the discussion on the validation plan. The next two sections discuss how the C-INCAMI framework was applied for usability testing of the language and the tool for the three case studies by the testers.

6.3 Usability testing of the mock-up language

The usability of the mock-up language needs to be validated. The C-INCAMI framework discussed in Section 6.1.1 is applied for usability testing of the mock-up language. The five activities in the C-INCAMI process discussed in Section 6.1.1.1 through to Section 6.1.1.5 are applied during the mock-up specification for the three case studies. Specifically, Section 6.3.1 deals with *defining of the testing requirements*, Section 6.3.2 provides finer details of how *effectiveness in use*, *efficiency in use* and *satisfaction in use* are defined, Section 6.3.3 defines the specificities of the tester tasks, Section 6.3.4 details the *design of the measurement metrics activity*, Section 6.3.5 deals with the *design of the evaluation indicators*,

Section 6.3.6 discusses *usability testing implementation details* and Section 6.3.7 follows up with the *analysis of the testing*.

6.3.1 Defining the testing requirements of the mock-up language

The *definition of the testing requirements* activity of the C-INCAMI process in Section 6.1.1.1 specifies four sub-activities: *establishing the information needed for the testing, specifying the testing context, design the testing tasks* and *selecting a concept model* for the test. Furthermore, *establishing the information needed for testing* includes *defining the purpose, defining user viewpoints, establishing the object* and *defining the focus concept* to be assessed.

The *purpose* in this case is “to understand and improve the usability of the visual language” from the *viewpoint* of “business analyst testers” of an *object* in the form of “a SME web application visual mock-up”. The *focus concept* to be assessed is *quality in use* and its sub-characteristics such as *effectiveness in use, efficiency in use, and satisfaction in use*. The testing context is “all nine testers use PCs with same specifications”. A screen video recording software unobtrusively recorded the screen activities during the test.

The next sub-activity within the activity of defining the testing requirements is the specification of the testing tasks. There can be many mock-up language specification tasks to test but from the language summary in Section 4.4 it is known that the most frequently used tasks are broadly to evaluate the specification of container structure and the specification of behaviour. The mock-up specification requirements have been refined below in the form of the container structure and behaviour for a SME web application.

Specification of the container (layout) structure in the visual mock-up: This activity has the following six sub-tasks, each of which has further sub-tasks which are discussed later in Table 9 through to Table 13 in Section 6.3.3

- Specification of *DFY Container* structure for creation of business entities
- Specification of *Search Container* structure for searching business entities

- Specification of *Search Result Container* structure for traversing through search results
- Specification of *Data View Container* structure to display business entities
- Specification of *Update Container* structure to update business entities
- Specification of *Navigation Only Container* to navigate among sections of the application

It may be observed that Report View Container specification is not listed above because from the discussion on Report View Containers in Section 4.2.4 it is known that *Report View Containers* are special type of *Data View Containers*.

Specification of behaviour of the web application in the visual mock-up: This activity has the following sub-tasks which are discussed later in Table 15.

- Specification of mock-up sub-sections for “creation” operations
- Specification of mock-up sub-sections for “search” operations
- Specification of mock-up sub-sections for “data view” operations
- Specification of mock-up sub-sections for “update” operations
- Specification of mock-up sub-sections for “delete” operations

Once the tasks and sub-tasks are known next an instance of the *quality in use* requirements tree was defined. An instance of the tree for validating the usability of the visual language is discussed next.

6.3.2 Usability requirements tree of the mock-up language

This section further elucidates the definitions and terms involved in usability testing. That is, it splits up *effectiveness in use* in terms of *sub-task correctness effectiveness*, *sub-task completeness effectiveness* and *task successfulness effectiveness*. Similarly, *efficiency in use* can be sub-divided in terms of *sub-task correctness efficiency*, *sub-task completeness efficiency* and *task successfulness efficiency*. This helps in the definition and measurement of *effectiveness in use* and *efficiency in use* in terms of tasks and sub-tasks along with further information such as the degree to which each task was accomplished *correctly* or *completely* or *successfully*. Similarly, it also provides the further details of how *satisfaction in use* can be measured. The definition of these terms helps in identifying sub-sections of the system that need further

improvement from a usability perspective. Section 6.3.2.1 discusses the details of *effectiveness in use* while Section 6.3.2.2 details *efficiency in use* and Section 6.3.2.3 discusses *satisfaction in use*.

6.3.2.1 Effectiveness in use

This section provides definitions of *sub-task correctness effectiveness*, *sub-task completeness effectiveness* and *task successfulness effectiveness*.

6.3.2.1.1 Sub-task correctness effectiveness

This is defined as “the degree to which specified users *correctly* execute sub-tasks of a task without regard to *completeness*” (Lew et al. 2012, p.313).

Such a sub-task may either be: a) complete and correct b) incomplete but correct

A complete and correct sub-task is a sub-task that been done completely with no errors. An incomplete but correct sub-task is a sub-task that has been correct but is not complete. Incompleteness may be caused due to lack of time or due to lack of perceived information.

6.3.2.1.2 Sub-task completeness effectiveness

This is defined as “the degree to which specified users completely execute sub-tasks of a task without regard to correctness” (Lew et al. 2012, p.313).

Such a sub-task can be complete but incorrect or complete and correct. A complete but incorrect sub-task is a sub-task that has been completed but has one or more errors in it.

6.3.2.1.3 Task successfulness effectiveness

This is defined as “the degree to which specified users correctly complete an entire task. That is, no errors for any sub-task, with all sub-tasks completed”(Lew et al. 2012, p.313).

6.3.2.2 Efficiency in use

The proportion the time for effective accomplishment of a task is used to measure efficiency. The time is found by analysing the screen cast of each user’s actions while developing the mock-up or while using the auto-generated application. This section defines the constituent parts of *efficiency in use* in terms of *sub-task correctness efficiency*, *sub-task completeness efficiency* and *task successfulness efficiency*.

6.3.2.2.1 Sub-task correctness efficiency

This is defined as the proportion of time required for sub-task correctness effectiveness.

6.3.2.2.2 Sub-task completeness efficiency

This is defined as the proportion of time required for sub-task completeness effectiveness.

6.3.2.2.3 Task successfulness efficiency

This is defined as the proportion of time required for task successfulness effectiveness.

6.3.2.3 Satisfaction in use

Satisfaction in use is a subjective opinion of each tester regarding the usability of the system in consideration. This is measured by employing a popular ten-item questionnaire for satisfaction measurement, called, System Usability Scale (SUS). SUS was developed as part of the usability engineering programme at Digital Equipment Co Ltd., Reading, United Kingdom and has been used for a variety of research projects and industrial evaluations (Brooke 2001). The advantage of this instrument is that it has been used for evaluation of many software applications and is easy to fill and calculate (Covella & Olsina 2006). Brooke (2001) recommends SUS to be used after the testers have completed the tasks. Further the tester should mark the centre point of the scale if they cannot respond to any item. SUS yields a score in the range of 0 to 100 with 0 for least satisfaction and 100 for full satisfaction.

The next sub-activity within the activity of definition of testing requirement in the C-INCAMI process is to define the specificities of the test tasks for validating the usability of the language. This is discussed in Section 6.3.3 which provides examples of how usability tasks can be broken down in terms of tasks and sub-tasks.

6.3.3 Usability testing tasks for validating the mock-up language

Usability test tasks define the specific tasks to be performed by the testers during usability testing. From Section 6.3.1 it is known this involves specifying: the layout structure of various mock-up sections within the application, navigation among mock-up sections and the behaviour of the system. This section elucidates these tasks in detail. From the discussion on the mock-up language specifications in Chapter 4, primarily six types of containers were identified, namely *Database Field Yielding Container*, *Search Container*, *Search Result Container*, *Data View Container (Report Container is considered as a special type of Data View Container)*, *Update Container* and *Navigation Only Container*. Correspondingly associated visual mock-up tasks are specified for each of the above-mentioned containers. The sub-tasks associated with

each of these tasks are highlighted in Table 9 through to Table 13 in this section, using examples from the “Travel Deals” case study.

The tasks and sub-tasks defined in Table 9 through to Table 13 also contain specification of the source widget for the navigation widgets that are used to trigger behaviour (operation) such as “insertion”, “search”, “deletion”, “update” and “display” of business entities. The mock-up specification also shows the target container once the operation is completed. Hence behavioural specification is primarily discussed in terms of: (i) specification of a widget that triggers the behaviour (ii) specification of an annotated navigation widget where the annotation represents the type of behaviour (iii) specification of a target for the annotated navigation widget. Moreover, certain behavioural tasks may require navigations (and hence sub-tasks) over many layout containers. For example, with respect to the “Travel Deals” case study in Figure 12, the specification of the *insert* operation of an “Order Deal” starts with the specification of the widgets associated with **“select for insert”** annotated navigation widget in “Available Deals[3]” *Search Result Container* in the “Travel Deals” page. This is followed by the specification of the widgets associated with the **“temporarily store for insert”** annotated navigation widget in the “Selected Order Details” *Data View Container*, followed by another **“temporarily store for insert”** annotated navigation widget in the “Customer” *Database Field Yielding Container* and finally ending with the specification of the **“commit inserts”** annotated navigation widget within the “Payment” *Database Field Yielding Container* in the “Order Deal” page. Similarly, the specification of “update” behaviour includes sub-tasks for specification of the widgets associated with the **“select for update”** annotated navigation widgets as well as for **“update”** annotated navigation widgets. Likewise, the specification of “data view” behaviour may require specification of the widgets associated with the **“previous”** and **“next”** annotations in a *Search Result Container* or within a *Report View Container*. Table 15 and Table 16 highlight the tasks and sub-tasks for the specification of behaviour with respect to the mock-up of the “Travel Deals” case study. This concludes all the sub-activities with the first activity of “defining the testing requirements” within the C-INCAMI process. The next activity is to design the measurement metrics which is discussed in Section 6.3.4.

Table 9: Sub-tasks in Database Field Yielding Container mock-up specification task

Task	Task name and description	Example from Travel Deals Case Study
1	Specify <i>DFY Container</i> for data entry for storage of new-information: In this step the user assembles labels, <i>data input widgets</i> and buttons for new data entry in one or more containers. The user may also enter data validation strings in the <i>data input widget</i> . In addition, the user may also use the "Unique Details" container and nesting of containers.	Example containers are: "Travel" in the "Add Travel Deal" page, "Administrator" in the "Add Administrator" page and "Customer", "Payment", "Unique Details", "Address" in the "Order Deal" page.
1.a	Assemble a label and a text input widget for a data input operation	"Deal Details" Label and the Text Input containing the "?(length>0)" text in the "Travel Deals" container in the "Add Travel Deal" page
1.b	Assemble a label and a date chooser widget for a data input operation	"Expiry Date" Label and the Date Chooser Input containing the "dd/mm/yyyy" text in the "Travel Deals" container in the "Add Travel Deal" page
1.c	Assemble a label, a text input widget and a button for a file upload operation	"Image" Label, empty Text Input and the button with text "Browse" in the "Travel" container in the "Add Travel Deal" page
1.d	Assemble a label and radio group for a data input operation	"Card Type" Label and the radio buttons in the "Payment" container in the "Order Deal" page
1.e	Assemble a label and a combo box for a data input operation	"Country" Label and the combo box in the "Address" container in the "Order Deal" page
1.e	Specify validation in <i>data input widgets</i>	Text Input containing the "?(length>0)" text in the "Travel" container in the "Add Travel Deal" page
1.f	Assemble one or more buttons to signify completion or cancellation of data input operation	"Add Deal" button in the "Travel" container in the "Add Travel Deal" page
1.g	Assemble a container widget to contain one or more of the widgets mentioned in 1.a to 1.f	"Travel Deals" container in the "Add Travel Deal" page
1.h	Specify a data input container widget to uniquely identify an entity.	"Unique Details" container in "Customer" container in the "Order Deal" page
1.i	Assemble nested container widgets to indicate has-a kind of relationship between data input widgets in the outer container to that in the inner container widgets	"Address" container within Customer" container in the "Order Deal" page

Table 10: Sub-tasks in Search Container specification task

Task	Task name and description	Example from Travel Deals Case Study
2	Specify container for data entry for searching existing information: In this step the user assembles labels, <i>data input widgets</i> and buttons to enter data for searching against previously entered data using the “=>Looked-up Widget” notation	“Search Deals” container in the “Travel Deals” page and “Administrator Sign-in” container in the “Login” page
2.a	Assemble a label and a text input widget containing the “=>Looked-up Widget” notation to specify a search criterion	“Deal Details” Label and the Text Input containing the “=>Deal Details” text in the “Search Deals” container in the “Travel Deals” page
2.b	Assemble a label and a date chooser <i>data input widget</i> containing the “=>Looked-up Widget” notation to specify a search criterion	“Expiry Date” Label and the Text Input containing the “=>Expiry Date” text in the “Search Deals” container in the “Travel Deals” page
2.c	Assemble a label and a combo box <i>data input widget</i> containing the “=>Looked-up Widget” notation to pre-populate a combo box for a “search criterion”	“Price” Label and the Combo box input containing the “=>Price” text in the “Search Deals” container in the “Travel Deals” page
2.d	Assemble one or more buttons to signify completion or cancellation of search operation	“Search” button in the “Search Deals” container in the “Travel Deals” page
2.e	Assemble a container to contain one or more widgets specified in 2.a to 2.d	“Search Deals” container in the “Travel Deals” page

Table 11: Sub-tasks in Search Result Container specification task

Task	Task name and description	Example from Travel Deals Case Study
3	Specify container for displaying search results: In this step the user assembles labels, <i>data view widgets</i> and buttons in one or more containers to model the display and traversal of search result sets	“Available Deals [3]” and “Deal” containers in the “Travel Deals” page
3.a	Assemble a label and a text input or text area widget containing the “= <i>reference widget</i> ” notation to specify a search result field	“Deal Details” Label and the Text Area “=Deal Details” text in the “Deal” container in the “Travel Deals” page
3.b	Assemble a Label, an Image data view widget and another Label with text containing the “= <i>reference widget</i> ” notation to reference an image for display	“Picture” Label and the Image with the =Image Label in the “Deal” container in the “Travel Deals” page
3.c	Assemble a container to contain one or more widgets specified in 3.a to 3.c	“Deal” container in the “Travel Deals” page
3.d	Assemble a container whose name contains text ending with the “[an integer]” notation to contain one or more buttons and a nested container to enable traversal of search result data-sets in a paginated form where each page contains “an integer” number of data-sets.	“Available Deals [3]” container in the “Travel Deals” page to display 3 search result data-sets
3.e	Assemble a container to contain the container specified in 3.e and one or more buttons for update, delete or other purposes.	“Search Results” container in the “Travel Deals” page

Table 12: Sub-tasks in Data View Container specification task

Task	Task name and description	Example from Travel Deals Case Study
4	Specify container for displaying selected data-sets from previous operations: In this task, the user assembles labels and or <i>data view widgets</i> and buttons in one or more containers to display data on the client side.	Example containers are “Selected Order Details” in the “Order Deal” page and “Order Deals” in the “Booking Confirmation” page.
4.a	Assemble a label and a text input or text area or image widget containing the “= <i>reference widget</i> ” notation for display of a selected client-side data set	“Deal Details” Label and the Text Area “=Deal Details” text in the “Selected Order Details” container in the “Order Deal” page
4.b	Assemble a label containing the “= <i>reference widget</i> ” notation for data set display from a previously selected <i>Data View Container</i>	Label containing “=Selected Order Details” text in the “Order Details” container in the “Booking Confirmation” page
4.c	Assemble a label containing the “= <i>reference widget</i> ” notation for data-set display from a <i>DFY Container</i>	Label containing “=Customer” text in the “Order Details” container in the “Booking Confirmation” page
4.d	Assemble a container to contain one or more widgets specified in 4.a to 4.c	“Selected Order Details” container in the “Order Deal” page and “Order Details” container in the Booking Confirmation” page

Table 13: Sub-tasks in Navigation Only Container specification task

Task	Task name and description	Example from Travel Deals Case Study
6	Specify container for containing navigation only widgets. In this step, the mock-up designer assembles buttons or link widgets with arrows emanating from them to targets. The arrows represent navigations. The target can be page or container.	The “Admin Operations” container in the “Deal Management” page.
6.1	Assemble a navigation widget with no data transfer or service request while navigating between a button or a link widget and target page or container	The arrows emanating from the “Admin Operations” container.

Table 14: Sub-tasks in Update Container specification task

Task	Task name and description	Example from Travel Deals Case Study
5	Specify a container for update of a selected data-set from search results: In this step the user assembles labels, <i>data input widgets</i> and buttons in one or more containers to update a selected data-set	Example containers are “Selected Deals” and “Travel” in the “Update Travel Deal” page, “Payment” in the “Update Payment Details” page and “Customer”, “Unique Details” and “Address” containers in the “Update Customer Details” page
5.a	Assemble a label and a text input widget for an update operation	“Deal Details” Label and the Text Input containing the “?(length>0)” text in the “Travel” container in the “Update Travel Deal” page
5.b	Assemble a label and a date chooser widget for an update operation	“Expiry Date” Label and the Date Chooser Input widget containing the “dd/mm/yyyy” text in the “Travel” container in the “Update Travel Deal” page
5.c	Assemble a label, a text input widget and a button for a file upload operation	“Image” Label, empty Text Input and the button with text “Browse” in the “Travel” container in the “Update Travel Deal” page
5.d	Assemble a label and radio group for an update operation	“Card Type” Label and the radio buttons in the “Payment” container in the “Update Payment Details” page
5.e	Assemble a label and a combo box for an update operation	“Country” Label and the combo box in the “Address” container in the “Update Customer Details” page
5.f	Specify validation in data update widgets	Text Input containing the “?(length>0)” text in the “Travel” container in the “Update Travel Deal” page
5.g	Assemble one or more buttons to signify completion or cancellation of update operation	“OK” button in the “Selected Deals” container in the “Update Travel Deal” page
5.k	Specify a data input container widget to uniquely identify an entity.	“Unique Details” container in “Customer” in the “Update Customer Details” page
5.l	Assemble nested container widgets to indicate has-a kind of relationship between data input widgets in the outer container to that in the inner container widgets	“Address” container in Customer” container in the “Update Customer Details” page

Table 15: Sub-tasks in behavioural tasks' specifications

Task	Task name and description	Example from Travel Deals Case Study
7.a	Specification of the insert behaviour: This behaviour is associated with mock-up segments in rows 7.a.i or 7.a.ii or 7.a.iii	
7.a.i	Specify a “select for insert” annotation on a navigation widget to indicate the selection of a business entity from a search result container for temporary storage on the client-side.	The “select for insert” annotation between the source “Search Results” container in the “Travel Deals” page and the target “Order Deal” page
7.a.ii	Specify a “temporarily store for insert” annotation on a navigation widget in a <i>data view container</i> or a <i>DFY container</i> to indicate temporary storage of a selected data entity associated with the source of the navigation widget and its linkage with an entity associated with the target of the navigation widget in a series of physical transactions during an insert operation	The “temporarily store for insert” annotation between source “Selected Order Details” container and the target “Customer” container in the “Order Deal” page. The “temporarily store for insert” annotation between source “Customer” container and the target “Payment” container in the “Order Deal” page.
7.a.iii	Specify a “commit inserts” annotation on navigation widget to indicate completion of an insert operation, resulting on one or more entities to be committed to the database.	The “commit inserts” annotation between source “Travel” container in “Add travel deal” page and target in “Deal Management” page
7.b	Specification of the “search” behaviour: Occurs during annotation on a navigation widget to indicate search behaviour based on the search criteria in the data input widgets in a <i>Search Container</i>	The “search” annotation while navigating from source “Search Deals” container in “Travel Deals” page. The “search” annotation while navigating from source “Administrator Sign-in” container in “Login” page
7.c	Specification of the update behaviour: This behaviour is associated with mock-up segments in rows 7.c.i and 7.c.ii	
7.c.i	Specify a “select for update” annotation on navigation widget to specify selection of data sets for future update operations in the database	The “select for update” annotation between “Search Results” container in the “Travel Deals” page and “update travel deals” page
7.c.ii	Specify an “update” annotation on navigation widget to specify update of data in an update container and in the database.	The “update” annotation between “Update Travel Deal” and “Travel Deal” page

Table 16: Sub-tasks in behavioural tasks' specifications (continuation)

Task	Task name and description	Example from Travel Deals Case Study
7.d	Specification of the data view behaviour: This behaviour is associated with mock-up segments in rows 7.d.i and 7.d.ii	
7.d.i	Specify a “ previous ” and “ next ” annotation on two navigation widgets to specify management of sets data in a <i>Data View Container</i>	The “ previous ” and “ next ” annotations in the “Available Deals[3]” container
7.d.ii	Optionally specify a <i>report view container</i> as the target of a “ commit insert ” navigation widget.	The “order” container in the “Booking Confirmation” page.
7.e	Specification of the “delete” behaviour: Occurs during the “ delete ” annotation on navigation widget to specify deletion of selected data sets in a <i>Data View container</i> and in the database.	The “ delete ” annotation in the “Travel Deals” page

It may be observed that the description of the tasks in Table 9 through to Table 15 are not related to the three case studies used in the validation of the language. The testing task details for the three case studies are discussed in the appendix. Specifically, Appendix 6.2.4 contains the visual modelling tasks required for the “Question and Answer” case study. Similarly, Appendix 6.2.5 and Appendix 6.2.6 respectively deal with the visual modelling tasks for “Teacher Consultation System” and “Patient-Dietician Consultation System”.

6.3.4 Designing the measurement metrics for the usability of the language

In this activity, the metrics are specified for each attribute of the requirements tree specified in Section 6.3.2. In addition, the measurement metrics for *satisfaction in use* is also discussed. That is with respect to *effectiveness in use* the discussion involves specification of metrics for *sub-task correctness effectiveness*, *sub-task completeness effectiveness* and *task successfulness effectiveness*. Similarly, metrics for attributes of *efficiency in use* are also defined. This section uses Lew et al., (2012)

definitions for the metrics. Section 6.3.4.1 through to Section 6.3.4.3 discuss the metrics for the attributes of *effectiveness in use* while Section 6.3.4.4 through to Section 6.3.4.6 discusses the same with respect to *efficiency in use*. Finally, Section 6.3.4.7 discusses the metrics for the measurement of *satisfaction in use*.

6.3.4.1 Designing the measurement of sub-task correctness effectiveness

Attribute: Sub-task correctness (coded in Section 6.3.2.1.1)

Indirect metric: This is the average ratio of sub-tasks that are correct whether incomplete or complete (defined as AvgRatioSubTasksCorrect)

Interpretation: $0 \leq \text{AvgRatioSubTasksCorrect} \leq 1$, higher is better

Objective: Calculate the overall proportion of the sub-tasks of a task, performed by all testers that are correct, irrespective of whether completed or not.

Computational method (Formula):

$$\text{AvgRatioSubTasksCorrect} = \frac{\sum(\text{SubTasksCorrect})_{j=1\dots n}}{\sum(\text{TotalSubTasks})_{j=1\dots n}}$$

for testers' $j = 1\dots n$, where n is the number of testers, "SubTasksCorrect" is the total of correct sub-tasks in a task performed by a tester and "TotalSubTask" is the total sub-tasks in the task expected to be performed by each tester.

Scale: Numeric Scale **Type:** Ratio

Unit (type, description): Percent (%), percentage of sub-tasks performed correctly without regard to whether complete or incomplete.

Each tester's visual mock-up is analysed to find the direct metric of whether each sub-task is correct or not.

6.3.4.2 Designing the measurement of sub-task completeness effectiveness

Attribute: Sub-task completeness (coded in Section 6.3.2.1.2)

Indirect metric: Average ratio of sub-tasks that are complete whether correct or incorrect (defined as AvgRatioSubTasksComplete)

Interpretation: $0 \leq \text{AvgRatioSubTasksComplete} \leq 1$, higher is better

Objective: Calculate the overall proportion of the sub-tasks performed by all testers that are complete, irrespective of whether correct or not.

Computational method (Formula):

$$\text{AvgRatioSubTasksComplete} = \frac{\sum(\text{SubTasksComplete})_{j=1\dots n}}{\sum(\text{TotalSubTasks})_{j=1\dots n}}$$

for testers' $j = 1\dots n$, where n is the number of testers, "SubTasksComplete" is the total of complete sub-tasks in a task performed by a tester and "TotalSubTasks" is the total sub-tasks in the task expected to be performed by each tester.

Scale: Numeric Scale **Type:** Ratio

Unit (type, description): Percent (%), percentage of sub-tasks performed completely without regard to whether correctly or not.

As before each tester's visual mock-up is analysed used to find the direct metric of whether each sub-task is complete or not.

6.3.4.3 Designing the measurement of task successfulness effectiveness

Attribute: Task successfulness (coded in Section 6.3.2.1.3)

Indirect metric: Average ratio of tasks that are complete and correct (defined as AvgRatioTasksSuccessful)

Interpretation: $0 \leq \text{AvgRatioTasksSuccessful} \leq 1$, higher is better

Objective: Calculate the overall proportion of a task performed by all users that are complete and correct.

Computational method (Formula):

$$\text{AvgRatioTasksSuccessful} = \sum(\text{TaskSuccessful})_{j=1\dots n} / \sum(\text{ATask})_{j=1\dots n}$$

for testers' $j = 1\dots n$, where n is the number of testers, $\sum(\text{TaskSuccessful})_{j=1\dots n}$ is the number of successful tasks of a kind performed by all testers and $\sum(\text{ATask})_{j=1\dots n}$ is the total number tasks of the same kind, expected to be performed by all testers.

Scale: Numeric Scale **Type:** Ratio

Unit (type, description): Percent (%), percentage of tasks that are performed completely and correctly.

6.3.4.4 Designing the measurement of sub-task correctness efficiency

The measurement sub-task correctness efficiency is got by finding the time expended for each correct sub-task or task. The time for each sub-task or task is calculated by analysing the video screen casts of each tester's activity.

Attribute: Sub-task correctness efficiency (coded in Section 6.3.2.2.1)

Indirect metric: This is the average ratio of time for sub-tasks that are correct whether incomplete or complete (AvgRatioTimeSubTasksCorrect)

Interpretation: $0 \leq \text{AvgRatioTimeSubTasksCorrect} \leq 1$, higher is better

Objective: Calculate the overall proportion time for sub-tasks of a task, performed by all users that are correct, irrespective of whether completed or not.

Computational method (Formula):

$$\text{AvgRatioTimeSubTasksCorrect} = \sum(\text{SubTasksTimeCorrect})_{j=1\dots n} / \sum(\text{TotalTasksTime})_{j=1\dots n}$$

for testers' $j = 1\dots n$, where n is the number of testers, SubTasksTimeCorrect is the total time for correct sub-tasks in a task performed by a tester and TotalTasksTime is the total time for all sub-tasks in a task expected to be performed by each tester.

Scale: Numeric Scale **Type:** Ratio

Unit (type, description): Percent (%), percentage of time for sub-tasks performed correctly without regard to whether complete or incomplete

6.3.4.5 Designing the measurement of sub-task completeness efficiency

The measurement sub-task completeness efficiency is got by finding the time expended for each completed sub-task or task. The time for each sub-task or task is calculated by observing the video screen casts of each tester's activity.

Attribute: Sub-task completeness efficiency (coded in Section 6.3.2.2.2)

Indirect metric: Average ratio of time for sub-tasks that are complete whether correct or incorrect (AvgRatioTimeSubTasksComplete)

Interpretation: $0 \leq \text{AvgRatioTimeSubTasksComplete} \leq 1$, higher is better

Objective: Calculate the overall proportion time for sub-tasks performed by all users that are complete, irrespective of whether correct or not.

Computational method (Formula):

AvgRatioTimeSubTasksComplete =

$$\frac{\sum(\text{SubTasksTimeComplete})_{j=1\dots n}}{\sum(\text{TotalTasksTime})_{j=1\dots n}}$$

for testers' $j = 1\dots n$, where n is the number of testers, SubTasksTimeComplete is the total time for completed sub-tasks performed by each tester and TotalTasksTime is the total time for all sub-tasks in a task expected to be performed by each tester.

Scale: Numeric Scale **Type:** Ratio

Unit (type, description): Percent (%), percentage of time for sub-tasks performed completely without regard to whether correct or not.

6.3.4.6 Designing the measurement of task successfulness efficiency

Attribute: Task successfulness efficiency (coded in Section 6.3.2.2.3)

Indirect metric: Average ratio of time for tasks that are complete and correct (AvgRatioTimeTasksSuccessful)

Interpretation: $0 \leq \text{AvgRatioTimeTasksSuccessful} \leq 1$, higher is better

Objective: Calculate the overall proportion of time for a task performed by all users that are complete and correct.

Computational method (Formula):

$$\text{AvgRatioTimeTasksSuccessful} = \frac{\sum(\text{TaskSuccessfulTime})_{j=1\dots n}}{\sum(\text{ATaskTime})_{j=1\dots n}}$$

for testers' $j = 1\dots n$, where n is the number of testers, $\sum(\text{TaskSuccessfulTime})_{j=1\dots n}$ is the total time for successful tasks of a kind performed by all testers and $\sum(\text{ATaskTime})_{j=1\dots n}$ is the total time for same kind of tasks, expected to be successfully completed by all testers.

Scale: Numeric Scale **Type:** Ratio

Unit (type, description): Percent (%), percentage of time for tasks performed completely and correctly.

Following the design of the measurement step in the C-INCAMI process is the design of the evaluation indicators which is discussed next.

6.3.4.7 Designing the measurement metrics for satisfaction in use of the mock-up language

Satisfaction in use is subject to how fulfilled the testers feel on using the system. Since feelings cannot be measured easily, as discussed at the beginning of this chapter, System Usability Scale (SUS), a popular industrial strength instrument is used for the measurement of *satisfaction in use* of the mock-up language. SUS is a 10-question survey in which common responses to odd numbered items refers to strong agreement, and to even number items refers to strong disagreement. This prevents response biases caused by respondents not having to think about each statement and by alternating positive and negative items, the respondent has to make an effort

to think whether they agree or disagree with it (Brooke 2001). Figure 47 highlights the 10 items in the SUS. SUS yields a single value score between 0 and 100 representing the overall value for satisfaction of the system being tested. The SUS score is computed by adding the score contributions from each item which is a value between 0 and 4. For odd numbered items the score contribution is scale position minus 1 and for even number items the contribution is 5 minus the scale position. The sum of the score is multiplied by 2.5 to obtain the overall SUS score.

This concludes the discussion on the design of the measurement metrics for *effectiveness in use*, *efficiency in use* and *satisfaction in use*. The next activity in the C-INCAMI process is to design evaluation indicators, which is discussed in Section 6.3.5.

	Strongly disagree				Strongly agree
1. I think that I would like to use this system frequently	1	2	3	4	5
2. I found the system unnecessarily complex	1	2	3	4	5
3. I thought the system was easy to use	1	2	3	4	5
4. I think that I would need the support of a technical person to be able to use this system	1	2	3	4	5
5. I found the various functions in this system were well integrated	1	2	3	4	5
6. I thought there was too much inconsistency in this system	1	2	3	4	5
7. I would imagine that most people would learn to use this system very quickly	1	2	3	4	5
8. I found the system very cumbersome to use	1	2	3	4	5
9. I felt very confident using the system	1	2	3	4	5
10. I needed to learn a lot of things before I could get going with this system	1	2	3	4	5

Figure 47: System Usability Scale. Copyright Digital Equipment Corporation, 1986

6.3.5 Specifying acceptable threshold levels for evaluation indicators

Evaluation indicators help in interpreting the values of measurements. “Indicators are ultimately the foundations for interpretation of information needs and decision-making”(Covella & Olsina 2006, p.5). Acceptable threshold values can be set for partial and global indicators for *sub-task correctness effectiveness, sub-task completeness effectiveness, task successfulness effectiveness, sub-task correctness efficiency, sub-task completeness efficiency, task successfulness efficiency* and *satisfaction in use*.

Table 17 highlights the elementary and global indicators along with threshold values for acceptance of the measure values. The ranges for threshold values were set by reviewing existing literature on usability testing of software systems (Covella & Olsina 2006; Lew et al. 2012; Becker & Olsina 2010). In Table 17 colour codes are used to interpret the level of satisfaction, namely, green for no improvement required, orange for some improvement required and red for must have improvement. During the calculation of the global indicators weighting may be assigned for sub-tasks however since this thesis is exploratory in nature equal weights are assigned to each sub-task. Furthermore, all sub-tasks are assumed to be mandatory for any given task. That is, a global (linear additive) aggregation model is used to calculate the values for each attribute in requirement tree, with equal weights for their elements. However, a real-world scenario may demand extra weighting for specific tasks requiring changes to the computation method for global indicators. In this thesis, all attributes are assumed to be of equal importance and all attributes are deemed to be mandatory during the calculation of the global indicators.

Once the evaluation indicators for interpreting the results are designed, the next activity in the C-INCAMI process is to implementation the measurement of the usability testing values of the mock-up language which is discussed in the next section.

Table 17: Threshold values for Quality in Use indicator levels

Global Indicator (GI) Name	Elementary Indicator (EI) Name	% Range	Interpretation
Quality in use is dependent on GI1, GI2, GI3		>= 80 >=70>80 <70	Satisfactory: needs no improvement Marginal: needs some improvement Unsatisfactory: needs improvement
GI1: Task effectiveness in use is dependent on EI1, EI2, EI3		Same as above	Same as above
	EI1: Task effectiveness correctness in use	Same as above	Same as above
	EI2: Task effectiveness completeness in use	Same as above	Same as above
	EI3: Task effectiveness successfulness in use	Same as above	Same as above
GI2: Task efficiency in use is dependent on EI4, EI5, EI6		Same as above	Same as above
	EI4: Task efficiency correctness in use	Same as above	Same as above
	EI5: Task efficiency completeness in use	Same as above	Same as above
	EI6: Task efficiency successfulness in use	Same as above	Same as above
GI3: Overall satisfaction in use		Same as above	Same as above

6.3.6 Implementing the measurement - Mock-up Language Usability values

Implementing the measurement defines how data is collected for the direct metrics and how data is computed for indirect metrics discussed in Section 6.1.1.4. Data is collected for direct metrics from elementary tasks like those defined in Table 9 through to Table 15 but with respect to the three case studies specified in Section 6.2.3. Section 6.3.6.1 discusses how the data is collected for direct metrics. This is followed by the discussion on how the computation of the values for indirect metrics is done. This is discussed in Section 6.3.6.2.

6.3.6.1 Data collection for direct metrics

The data collection from testing tasks related to direct metrics needs to be done in a systematic manner. This section discusses how the data is collected when the testers performed tasks for three case studies specified in Section 6.2.3.

Each case study was randomly assigned to three testers resulting in a total of nine test cases for nine testers. Each tester was requested to develop the mock-up of the assigned case study in a two-step approach, paper and pencil followed by editing using the Balsamiq tool. Within a week of the training session discussed in Section 6.2.2, the testers were allocated a day to produce the mock-up on paper for an assigned case study. The creation of the mock-up using paper and pencil rather than using the tool directly ensures that the testers are not burdened by the tool. The following day the testers were requested to use the Balsamiq tool to produce a softcopy of the paper-based mock-up. During this process a screen recording software was used to unobtrusively record their actions for measurement of efficiency using time as a factor. The rest of this section discusses how data is collected, from the structural and behavioural specifications of the visual mock-up and from the SUS instrument.

It may be recalled from Section 6.3.1, usability testing of the visual mock-up essentially includes two aspects, one, testing of the container structure and widgets within the containers and, two, testing of the behavioural elements in the mock-up. Correspondingly data can be collected while specifying mock-up tasks for the specification of: *DFY Containers*, *Search Containers*, *Search Result Containers*, *Data View Containers*, *Update Containers* and *Navigation Only Containers*. Similarly, the data can be collected during the specification of the behavioural elements for “insert”, “update” “delete”, “search” and “data view” operations. Table 9 through to Table 15 in Section 6.3.3 provides examples of how these tasks are performed with respect to the “Travel Deals” case study while Appendix 6.2.4 to Appendix 6.2.6 discusses the same with respect to each of the three case studies used for testing.

The mock-ups produced by randomly chosen three testers for the three case studies are provided as an example each in Appendix 6.2.4 through to Appendix 6.2.6.

Data collection for elementary indicators in a task or sub-task was performed by identifying whether each action in a sub-task was correct or incorrect and complete or incomplete by comparing it with the expected actions for a task or a sub-task. This is based on the definitions for sub-task correctness effectiveness, sub-task completeness effectiveness and task successfulness effectiveness from Section 6.3.2. In addition, the screen recording was used to find the time required for each task or sub-task for the measurement of corresponding sub-task correctness efficiency, sub-task completeness efficiency and task successfulness efficiency. Figure 48 illustrates an example of how the data is tabulated for each action in a sub-task for the creation of a Question with respect to the “Question and Answer” case study. It is clear from this figure that each action within a sub-task or a task can be objectively marked as correct or incorrect and as complete or incomplete. Data is similarly collected for all tasks and sub-tasks specified in Appendix 6.2.4, Appendix 6.2.5 and Appendix 6.2.6 for the three case studies.

		1: Correct 0: Incorrect	1: Complete 0: Incomplete	Units of time
Sub-Task: Mock-up for creating a Question				202
Actions in the sub-task	Mock-up uses Container for creation of Question	1	1	
	In mock-up Question container exists in a page	1	1	
	In mock-up Question container is DFY Container	1	1	
	The DFY Container has a unique name	1	1	
	In Container a unique lable exists for question	1	1	
	In Container a unique label exists for answer	0	0	
	In Container a unique label exists for status	0	0	
	In Container a Data Input Widget exists for question text	1	1	
	In Container a Data Input Widget exists for answering	0	0	
	In Container a Data Input Widget exists for status	0	0	
	In Container a button exists to trigger an insert operation	1	1	
In Container a navigation widget with "commit inserts" exists with above button as source	1	1		

Figure 48: Illustrating tabulation of data collection

Data for behavioural tasks were similarly collected for business operations by considering them either as a single physical transaction operation or as a series of physical transactions where each physical transaction is represented by a service widget (which is normally a button) that triggers the event and an appropriately annotated navigation widget. Specifically, the expected actions for an insert behaviour involving a single physical transaction is considered by associating the

tester's mock-up actions for specification of a button and a **"commit inserts"** annotated navigation widget within a DFY Container. On the other hand, the expectation for an insert behaviour involving multiple physical transactions is considered by first specification of a button and a **"select for insert"** annotated navigation widget, followed by a series of zero or more pairs of a button and a **"temporarily store for insert"** annotated navigation widgets and finally another pair of a button and a **"commit inserts"** annotated navigation widget as discussed in Section 5.3. Similarly, the expected task specification for "search" behaviour is a representation of a pair of button and a **"search"** annotated navigation widget in a *Search Container*. The "update" behaviour specification is made of two physical transactions, one for selection of entities for update and another for update of the selected entities. Correspondingly the expected task specifications for update is made of a pair of button and a **"select for update"** annotated navigation widget and a pair of button and a **"update"** annotated navigation widget. The expected specification for delete behaviour is a pair of a button and a **"delete"** annotated navigation button in a *Search Result Container*. In addition, the tasks for specifying the **"previous"** and the **"next"** annotated navigation widgets along with their associated buttons were grouped together with tasks for report generation while collecting data for representing data view behaviour.

Finally, data collection for satisfaction in use was done by requesting the testers to answer the SUS questionnaire provided in Figure 47 following the creation of the mock-up using Balsamiq.

6.3.6.2 Computing indirect metrics

Indirect metrics represent the overall usability values for each task of interest. From the discussion in the previous sub-section it is known that the tasks are to create mock-ups for the three SME application requirements using appropriate types of containers and annotations on navigational widgets. Specifically, usability values for indirect metrics for common tasks such as "insert", "update", "delete", "search" and

“data view” operations are computed from the nine mock-up specifications for the three case studies.

The following two paragraphs discuss the implementation details of the computation method defined in Section 6.3.4 with respect to *effectiveness in use* and *efficiency in use* by nine testers during the specification of *DFY Containers* of the web applications for the three case studies defined in Appendix 6.2. The discussion is not repeated for other containers since the implementation details of the computation method is same.

As discussed earlier *sub-task correctness effectiveness* is the ratio of “the sub-tasks that are correct without regards to completeness” to “the total sub-tasks expected in the task”. The “task” in consideration is “the mock-up specification of the *DFY Container*” and “sub-tasks” are “the actions within the task to be performed by the nine testers for the three case studies”. The expected total sub-tasks for the specification of *DFY Containers* by the nine testers were 231 and the total of the correct sub-tasks were 217, thereby yielding a value of 94% for “*DFY Container* sub-task mock-up *correctness effectiveness*”. Similarly, *sub-task completeness effectiveness* is the ratio of “the sub-tasks that are complete without regards to correctness” to “the total sub-tasks expected in the task”. The total of the completed sub-tasks was 218, yielding a value of 94% again for “*DFY Container* sub-task mock-up *completeness effectiveness*”. Finally, *task successfulness effectiveness* is defined as the ratio of “similar tasks that are correct and complete” to “the total number of tasks” of the same kind. The total number of *DFY Container* specification tasks was 21 for the nine testers and total number of successfully completed tasks was 15, yielding a *task successfulness effectiveness* percentage value of 71.

Sub-task correctness efficiency is the ratio of “time for sub-tasks in a task that are correct without regards to completeness” to “the expected time for all sub-tasks in the task”. The time for total correct sub-tasks was found by analysing video records captured during the specification of the mock-ups. On analysis of the video records the time for the specification of *DFY Containers* by the nine testers was found to be 4786 units. However, this also included the time for incorrect actions. Hence “the

time for correct specification of *DFY Containers*” was computed by multiplying 4786 by the ratio of sub-task correctness effectiveness, yielding a value of 4499 units of time. Similarly, “the expected time for total sub-tasks in the task” was computed by multiplying 4786 by the inverse ratio of sub-task correctness effectiveness, yielding a value of 5092. Hence the sub-task correctness efficiency for *DFY Container* specification is 4499/5092 or 88%. Similar computational methods were used to find the sub-task completeness efficiency and task-successfulness efficiency during *DFY Container* specifications.

Table 18 through to Table 23 indicate the percentage usability values of the elementary indicators and global indicators for the specification tasks of the containers. Furthermore, Table 24 to Table 25 contain similar values for the behaviours using the usability metrics defined in Section 6.3.4. Finally, the SUS instrument was implemented to find the *satisfaction in use* value of 73%. The analysis of these values is discussed in Section 6.3.7.

Table 18: Effectiveness in Use and Efficiency in Use values for DFYC mock-up task

Characteristics and attributes	Percentage Values: Elementary Indicator (EI) and Global Indicator (GI). Average for 9 testers	
	EI Value	GI Value
1.1 Effectiveness in use		86
1.1.1 Sub-task correctness effectiveness	94	
1.1.2 Sub-task completeness effectiveness	94	
1.1.3 Task successfulness effectiveness	71	
1.2 Efficiency in use		75
1.2.1 Sub-task correctness efficiency	88	
1.2.2 Sub-task completeness efficiency	89	
1.2.3 Task successfulness efficiency	48	

Table 19: Effectiveness in Use and Efficiency in Use values for Search Container mock-up task

Characteristics and attributes	Percentage Values: Elementary Indicator (EI) and Global Indicator (GI). Average for 9 testers.	
	EI Value	GI Value
1.1 Effectiveness in use		92
1.1.1 Sub-task correctness effectiveness	93	
1.1.2 Sub-task completeness effectiveness	94	
1.1.3 Task successfulness effectiveness	88	
1.2 Efficiency in use		88
1.2.1 Sub-task correctness efficiency	93	
1.2.2 Sub-task completeness efficiency	89	
1.2.3 Task successfulness efficiency	83	

Table 20: Effectiveness in Use and Efficiency in Use values for Search Result Container mock-up task

Characteristics and attributes	Percentage Values: Elementary Indicator (EI) and Global Indicator (GI). Average for 9 testers.	
	EI Value	GI Value
1.1 Effectiveness in use		95
1.1.1 Sub-task correctness effectiveness	100	
1.1.2 Sub-task completeness effectiveness	95	
1.1.3 Task successfulness effectiveness	90	
1.2 Efficiency in use		95
1.2.1 Sub-task correctness efficiency	100	
1.2.2 Sub-task completeness efficiency	95	
1.2.3 Task successfulness efficiency	90	

Table 21: Effectiveness in Use and Efficiency in Use values for Data View Container mock-up task

Characteristics and attributes	Percentage Values: Elementary Indicator (EI) and Global Indicator (GI). Average for 9 testers.	
	EI Value	GI Value
1.1 Effectiveness in use		96
1.1.1 Sub-task correctness effectiveness	98	
1.1.2 Sub-task completeness effectiveness	98	
1.1.3 Task successfulness effectiveness	92	
1.2 Efficiency in use		95
1.2.1 Sub-task correctness efficiency	97	
1.2.2 Sub-task completeness efficiency	97	
1.2.3 Task successfulness efficiency	90	

Table 22: Effectiveness in Use and Efficiency in Use values for Update Container mock-up task

Characteristics and attributes	Percentage Values: Elementary Indicator (EI) and Global Indicator (GI)	
	Average for 9 testers	
	EI Value	GI Value
1.1 Effectiveness in use		83
1.1.1 Sub-task correctness effectiveness	83	
1.1.2 Sub-task completeness effectiveness	83	
1.1.3 Task successfulness effectiveness	83	
1.2 Efficiency in use		69
1.2.1 Sub-task correctness efficiency	69	
1.2.2 Sub-task completeness efficiency	69	
1.2.3 Task successfulness efficiency	69	

Table 23: Effectiveness in Use and Efficiency in Use values for Navigation Only Container mock-up task

Characteristics and attributes	Percentage Values: Elementary Indicator (EI) and Global Indicator (GI). Average for 9 testers	
	EI Value	GI Value
1.1 Effectiveness in use		95
1.1.1 Sub-task correctness effectiveness	98	
1.1.2 Sub-task completeness effectiveness	98	
1.1.3 Task successfulness effectiveness	89	
1.2 Efficiency in use		93
1.2.1 Sub-task correctness efficiency	96	
1.2.2 Sub-task completeness efficiency	96	
1.2.3 Task successfulness efficiency	87	

Table 24: Effectiveness in Use values for behavioural tasks in the mock-up

Characteristics and attributes	Percentage Values: Elementary Indicator (EI) and Global Indicator (GI). Average for 9 testers	
	EI Value	GI Value
1.1 Effectiveness in use		94
1.1.1 Sub-task correctness effectiveness		95
1.1.1.a Sub-task correctness effectiveness (search result, data view)	92	
1.1.1.b Sub-task correctness effectiveness (search)	97	
1.1.1.c Sub-task correctness effectiveness (insertion operation)	92	
1.1.1.d Sub-task correctness effectiveness (update operation)	92	
1.1.1.e Sub-task correctness effectiveness (delete operation)	100	
1.1.2. Sub-task completeness effectiveness		95
1.1.2.a Sub-task completeness effectiveness (search result, data view)	92	
1.1.2.b Sub-task completeness effectiveness (search)	100	
1.1.2.c Sub-task completeness effectiveness (insertion operation)	92	
1.1.2.d Sub-task completeness effectiveness (update operation)	92	
1.1.2.e Sub-task completeness effectiveness (delete operation)	100	
1.1.3 Task successfulness effectiveness		92
1.1.3.a Sub-task successfulness effectiveness (search result, data view)	91	
1.1.3.b Sub-task successfulness effectiveness (search)	86	
1.1.3.c Sub-task successfulness effectiveness (insertion operation)	89	
1.1.3.d Sub-task successfulness effectiveness (update operation)	92	
1.1.3.e Sub-task successfulness effectiveness (delete operation)	100	

Table 25: Efficiency in Use values for behavioural tasks in the mock-up

Characteristics and attributes	Percentage Values: Elementary Indicator (EI) and Global Indicator (GI). Average for 9 testers	
	Average for 9 testers	
	EI Value	GI Value
1.1 Efficiency in use		88
1.1.1 Sub-task correctness efficiency		89
1.1.1.a Sub-task correctness efficiency (search result, data view)	84	
1.1.1.b Sub-task correctness efficiency (search)	94	
1.1.1.c Sub-task correctness efficiency (insertion operation)	85	
1.1.1.d Sub-task correctness efficiency (update operation)	84	
1.1.1.e Sub-task correctness efficiency (delete operation)	100	
1.1.2. Sub-task completeness efficiency		91
1.1.2.a Sub-task completeness efficiency (search result, data view)	84	
1.1.2.b Sub-task completeness efficiency (search)	100	
1.1.2.c Sub-task completeness efficiency (insertion operation)	85	
1.1.2.d Sub-task completeness efficiency (update operation)	84	
1.1.2.e Sub-task completeness efficiency (delete operation)	100	
1.1.3 Task successfulness efficiency		84
1.1.3.a Sub-task successfulness efficiency (search result, data view)	82	
1.1.3.b Sub-task successfulness efficiency (search)	74	
1.1.3.c Sub-task successfulness efficiency (insertion operation)	80	
1.1.3.d Sub-task successfulness efficiency (update operation)	84	
1.1.3.e Sub-task successfulness efficiency (delete operation)	100	

6.3.7 Analysis and reporting of the evaluation of mock-up language

The last step in the C-INCAMI framework is to analyse and report the evaluation, in this case as an answer to research question 3.1 regarding the validation of the usability of the mock-up language. The results are analysed to improve those aspects of the mock-up language which have a low usability rating. The analysis of the usability testing results is performed by comparing them with the threshold values set in Section 6.3.5. It may be observed that those values above 80% are considered as acceptable, while those between 70 and 79% are considered to need some improvement and those below 70% are deemed to be unsatisfactory. This section first discusses the analysis of the result provided in the previous section, with respect to *effectiveness in use* and *efficiency in use* during the specification of the layout containers and the behaviours. This is followed by the discussion on *satisfaction in use*.

Figure 49 represents the percentage plots of the effectiveness in use and efficiency in use for the mock-up specifications of containers and Figure 50 represents the same for all behavioural operations. It may be observed that the values are above the 80% threshold for all specifications except *DFY Container* (75%) and *Update Container* (69%) efficiency in use. This indicates marginal improvement is required for the specification of these two types of containers. However, the average of all containers is above 80%, indicating that overall answer to research question 3.1 regarding the validation of the usability of the mock-up language is satisfactorily answered.

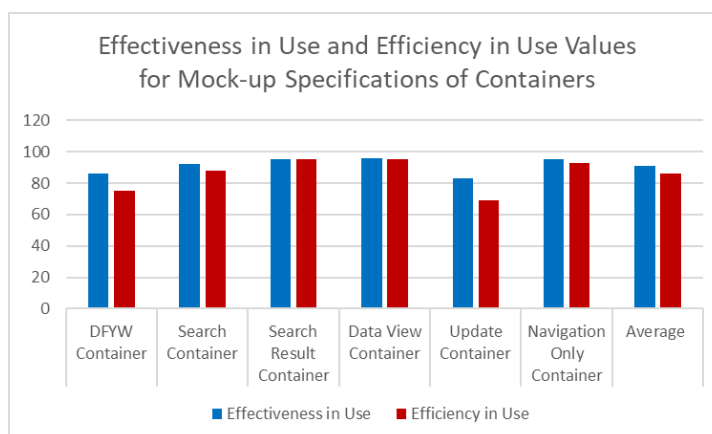


Figure 49: Effectiveness in Use and Efficiency in Use for mock-up specifications of Containers

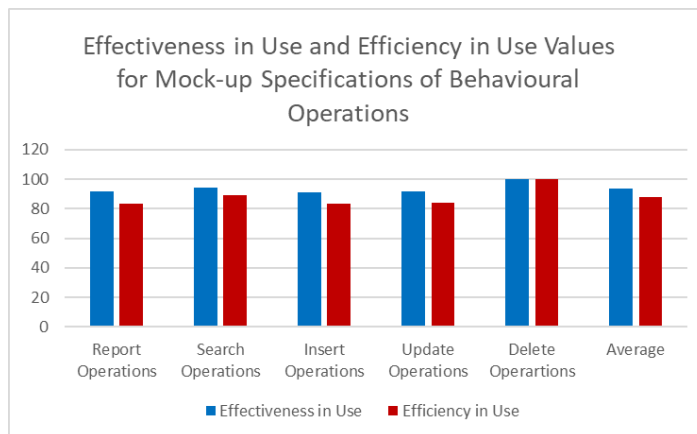


Figure 50: Effectiveness in Use and Efficiency in Use for mock-up specifications of behavioural operations

Interpreting the usability of the language for creating mock-up of *DFY Containers*, the global indicator value of 86% for *effectiveness in use* and 75% for *efficiency in use* indicates that 86 out of 100 times the nine testers could achieve the goals of specifying the diverse types of layout containers with accuracy and completeness and in the process 75 out of every 100 units of time is efficiently used for specifying the layout with accuracy and completeness. The *efficiency in use* value of 75 which is below the 80% is due to low values for *task successfulness in use* despite having high values of close to 90% or more for sub-task correctness and completeness. This indicates that despite most testers are doing most actions correctly and completely they are missing or doing some actions incorrectly. This could be due the fact that *DFY Containers* specifications involve many minor details to be fulfilled causing an elevated level of cognitive load. This could be rectified in the future by providing a self-check cheat sheet to mock-up designers to verify whether all the expected actions are accomplished.

The results for the mock-up of *Search Containers* are satisfactory with 92% and 88% values for the global indicators of *effectiveness in use* and *efficiency in use*. This indicates that the testers were confident of using mock-up notations for specifying search criteria. Similarly *Search Result Containers* and *Data View Container* specifications yielded above 95% value for the *effectiveness in use* and *efficiency in use* signifying confidence in the usage of mock-up notations for viewing search result data along with other notations for traversal through search results.

The results of the mock-up for *Update Container* yielded a satisfactory value of 83% for *effectiveness in use* but with a below satisfactory value of 69% for *efficiency in use*. Interestingly each of the attributes of the *effectiveness in use* and *efficiency in use* has equal values. This indicates whenever (83% of the total) the specification of *Update Container* task is accomplished, it is done correctly and completely but about 17% of the expected *Update Container* tasks are not attempted, which happens to be because of one tester forgot to provide specifications for the *Update Container*. It should be noted that the "Patient Dietician" case study had no update requirements; consequently, the *Update Container* specifications were analysed with respect to six mock-ups. In other words, one tester out of the remaining six missed the update container, resulting in a lower value for *efficiency in use*. As explained earlier a cross check-sheet may overcome such problems in the future.

The results of the usability test of Navigation Only Containers yields above 90% values for *effectiveness in use* and *efficiency in use*, indicating high-level of confidence in using the notations.

Similarly, the overall results of the usability test for the specification of behavioural operations such as for "search", "data view", "insert", "update" and "delete" yielded values of 94% and 84% respectively for *effectiveness in use* and *efficiency in use*, indicating that users understand the meaning of the various annotations used for behavioural specifications and are successfully able to apply them to the various case studies.

Finally, the SUS instrument discussed in Section 6.3.2.3 and in Section 6.3.4.7 yielded a *satisfaction in use* value of 73% for the mock-up language. Though this value is based on testers' perception of the usage of the language rather than an objective estimation method like that used for *effectiveness in use* and *efficiency in use*, it indicates that the testers are fairly satisfied of the features of tool. Figure 51 highlights the testers' responses on a scale of one to five to the various questions in the SUS instrument. It is worth observing that the average value of responses to positive questions is generally higher than four, indicating good satisfaction, while the average value of responses to negative questions is about two which is closer to

the lower limit of one, again indicating that testers were decisive in their appreciation of the mock-up language. This completes the analysis of the usability testing of the mock-up language. The next section deals with the usability testing of the generated application.

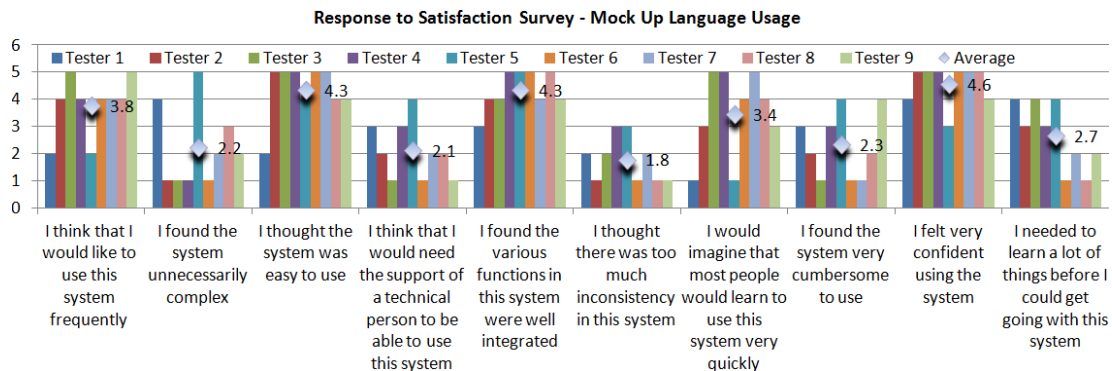


Figure 51: Plot of testers' responses to SUS questions on mock-up language usage

6.4 Usability testing of the auto-generated applications

Usability testing of the auto-generated applications can be used to validate the auto-generating tool. This section discusses how the C-INCAMI process was used again to perform the usability testing of the auto-generated application to successfully answer research question 3.2. The mock-ups developed by the nine testers for the three case studies were used to auto-generate the applications. For successful auto-generation, mock-ups should comply with the mock-up language specifications discussed in Section 4. From the analysis of the usability testing of the mock-ups in Section 6.3, it is seen that not all tasks achieved 100% rating for task *successfulness in use*, indicating that some sections of the mock-ups need to be redesigned before they could be input to the auto-generator. This was done by having a follow-up training session with each tester and addressing those elements that needed attention. Most of these elements were due to minor omissions rather than errors in usage of the language. Hence each such session was completed in a few minutes and the testers could produce correct mock-ups that could be used to auto-generate the applications.

This section contains detailed answers to research question 3.2 regarding the validation of the auto-generated applications. The C-INCAMI method of usability testing was used again for this purpose since it yields objective values of usage for *effectiveness in use* and *efficiency in use* of the auto-generated applications. In addition, the SUS instrument is also used to find a subjective value for the *satisfaction in use* of the auto-generated applications. It may be recalled from the discussion in Section 6.2 the C-INCAMI process identifies five activities to carry out usability testing: defining the testing requirements, designing the measurement metrics, designing the evaluation indicators, implement the measurements and analysing and reporting the evaluation. The application of this process for the usability testing of the nine auto-generated applications from the mock-ups of the nine testers is discussed in the following sections. Specifically Section 6.4.1 deals with designing of the testing requirements, Section 6.4.2 details the designs of the measurement metrics activity, Section 6.4.3 deals with the design of the evaluation indicators, Section 6.4.4 discusses usability testing implementation details and Section 6.4.5 follows up with the analysis and reporting of the testing.

6.4.1 Defining the testing requirements of the generated applications

As mentioned earlier in Section 6.1.1.1 the definition of the testing requirements phase of the C-INCAMI process includes four sub-activities: establishing the information needed for the testing, specifying the testing context, designing the testing tasks and selecting a concept model for the test. Furthermore, it may be recalled that the first sub-activity of establishing the information needed for testing includes additional details such as defining the **purpose**, defining user **viewpoints**, establishing the **object** and defining the focus **concept** to be assessed.

The **purpose** is “to understand and improve the auto generated application” from the **viewpoint** of “business analyst testers” of an **object** in the form of “a SME web application”. The focus concept to be assessed is Quality in Use and its sub-characteristics such as *effectiveness in use*, *efficiency in use* and *satisfaction in use* of

the generated application. The testing context is same as in the usability testing of the mock-up language, so its details are not repeated.

The next sub-activity in the definition of the testing requirements is the specification of the testing tasks. There can be many operations of the generated applications but the most frequently used tasks in SME applications discussed in Section 4.2 are for insert, search, delete, update and display operations. From a testing perspective these tasks can be viewed as the usability of the corresponding types of layout containers. Hence the usability testing of the generated applications is same as the usability testing of the layout containers for:

- insert operations
- search operations
- data view operations
- update operations
- delete operations

Figure A-2 to Figure A-4 represent use cases related to the above type of operations and Appendix 6.2.7 through to Appendix 6.2.9 discuss their finer details of user tasks with respect to the three case studies for the auto-generated applications.

The next sub-activity in the definition of the testing requirements is the specification of the *quality in use* requirements tree for usability testing of the generated application. Since the *quality in use* requirements remain the same as that discussed in Section 6.3.2 it has not been repeated here. However, in this context it is important to note that correctness does not refer to functional correctness of the generated application rather it refers to the correct usage of the generated application. Usability testing for functional correctness of the generated application is discussed separately in Section 6.5. The next step in the C-INCAMI process is to design the measurement which is discussed in the next section.

6.4.2 Designing the measurement for the usability of the generated applications

In this activity, the metrics are specified for each attribute of the requirements tree specified in Section 6.3.2. Since the details are same as that discussed in Section 6.3.4.1 to Section 6.3.4.7, further details are not repeated here.

6.4.3 Designing the usability evaluation indicators of the generated applications

In this next step of the C-INCAMI process the design of the evaluation indicators for the usability of the generated application are specified. Once again for reasons mentioned earlier the thresholds for acceptable values of measurements are same as those discussed in Section 6.3.5 so further details are avoided here.

6.4.4 Implementing usability testing measurement of the generated applications

Usability testing measurement of the auto-generated applications is implemented by analysing the screen records of the nine testers' actions for the three case study user tasks defined in Appendix 6.2.7 to Appendix 6.2.9. All the testers performed the user tasks correctly and completely during the implementation of the measures. That is the implementation of the usability test of the auto-generated applications resulted a value of 100% for *effectiveness in use* as well as for *efficiency in use* for all operations identified in Section 6.4.1. The 100% ratings for *effectiveness in use* and *efficiency in use* can be attributed to the fact that the testers had themselves designed the mock-up of the applications and thus were confident of using the auto-generated tool. However, the *satisfaction in use* rating from the SUS (John Brooke 2001) survey yielded a value of 78%. Hence the overall usability rating for the generated application was found to be 93%.

6.4.5 Analysis and reporting of the evaluation of the generated applications

This is the last activity in the C-INCAMI process for usability testing of the auto-generated applications. On comparing the *effectiveness in use*, *efficiency in use* and *satisfaction in use* values of the auto-generated application with the threshold set in Section 6.3.5, it is evident that the auto-generated applications portray good usability ratings. Thus, it satisfactorily answers research question 3.2, regarding the validation

of the auto-generated application. In addition, the SUS survey responses indicate that the testers perception regarding the usage of developed applications is positive. Specifically, Figure 52 highlights the testers' responses on a scale of one to five to the various questions in the SUS instrument. As in the case of the mock-up language it is worth observing that the average value of responses to positive questions is generally higher than four, indicating good satisfaction, while the average value of responses to negative questions is about two which is closer to the lower limit of one, again indicating that testers were decisive in their appreciation of the auto-generated applications.

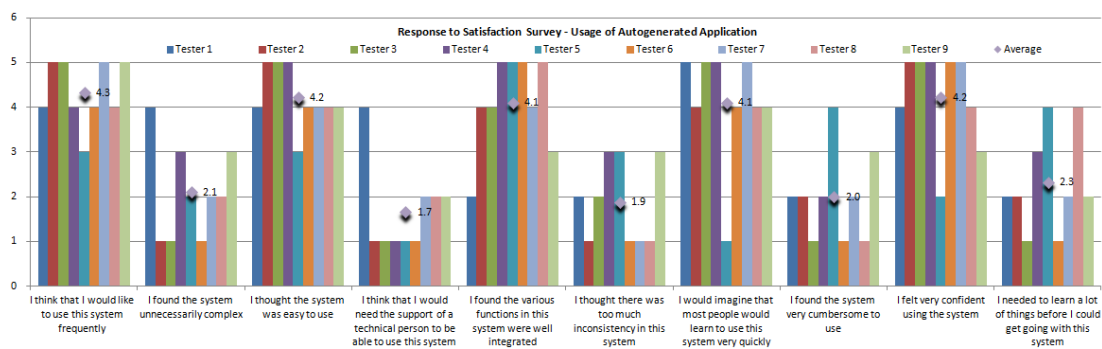


Figure 52: Plot of testers' response to SUS questions on usage of auto-generated applications

6.5 Testing functional correctness of the generated applications

Functional testing ensures that the applications are performing the operations correctly as expected. Functional correctness is tested for the primary operations of the generated application. These include the functional correctness of: insert, search, update, delete and report operations. The functional correctness for each type of operation is found by using a formula:

$$\text{Percentage functional correctness} = (\text{Sum of times the operation was found to be correct} / \text{Total number of times the operation was performed}) * 100.$$

Here correctness is defined by the appropriateness of the status of an entity or a set of entities immediately following an operation. For example, a successful insert operation should result in the correct values of the entity being stored in a database.

The status of the entities in the database were observed using a third-party tool such as phpMyAdmin. Thus, the number of times the insertion of entities was correct to the total number of such operations was found to compute the functional correctness of insert operations. Similarly, the formula was used to compute the functional correctness of the other operations.

The nine testers verified the functional correctness of the above-mentioned operations for each of the test cases specified in Appendix 6.2. All the operations were found to be functionally correct for each test use case, yielding a value of 100% for functional correctness of the four types of operations. Screenshots of auto-generated instances of web applications corresponding to the three test cases are specified in Figure 53, Figure 54 and Figure 55.

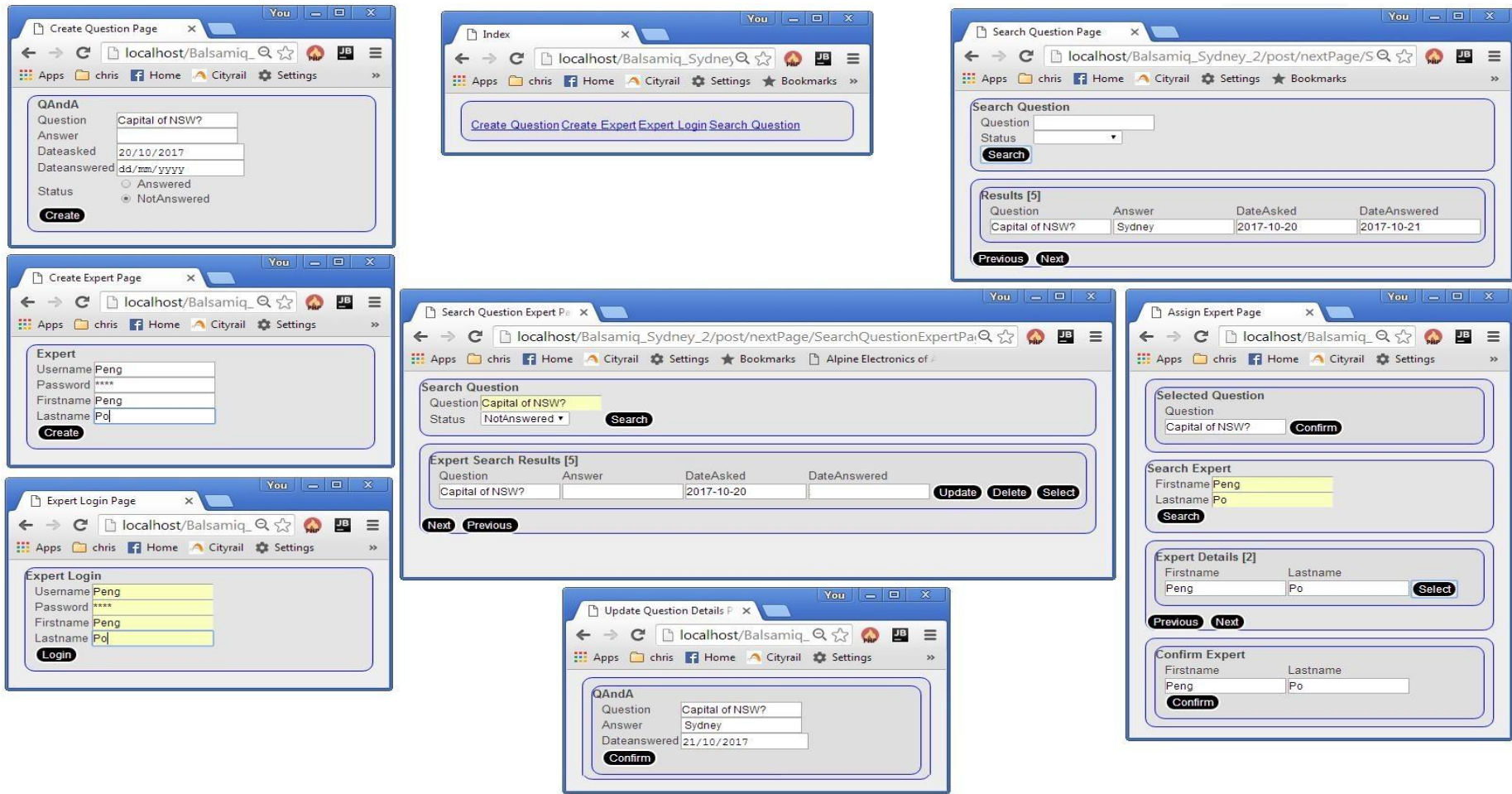


Figure 53: Screenshots of the auto-generated Question -Answer system

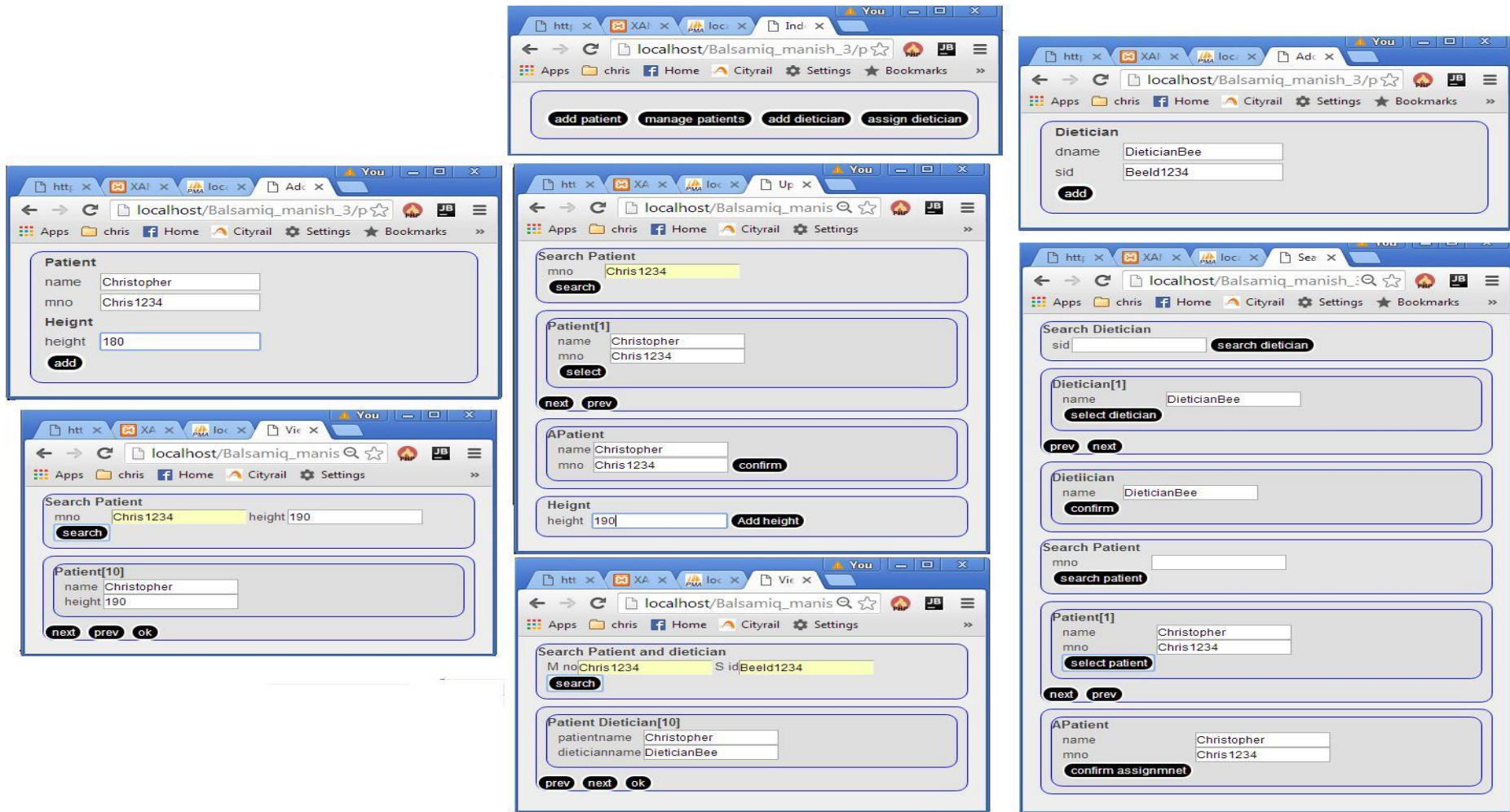


Figure 54: Screenshots of the auto-generated Patient-Dietician system

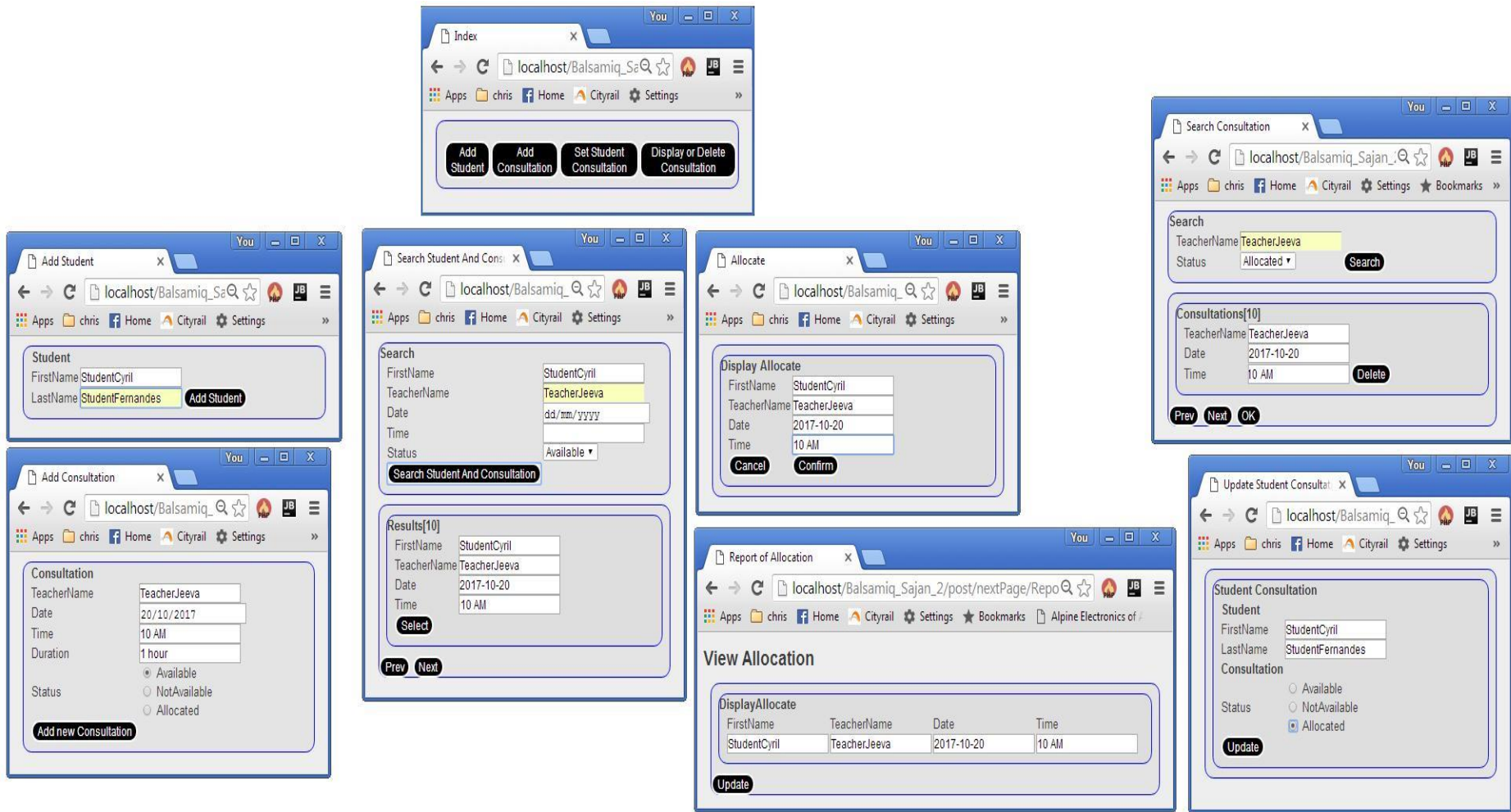


Figure 55: Screenshots of the auto-generated Teacher-Student Consultation system

6.6 Validation of the auto-generating tool as an integrated system

This section provides analytical reasoning as an answer to research question 3.3 regarding the validation of the tool for creating SME applications from mock-ups. That is, the validation of the auto-generating tool encompasses the validation of the usability of the language, validation of the usability of the mock-up tool for the creation of mock-ups and the usability of the auto-generated applications. Moreover, the usability of the auto-generated applications involves the validation of the usability of the applications for non-functional requirements as well as for functional requirements. The usability of the mock-up language and the mock-up tool was successfully validated by following the C-INCAMI process for usability testing in Section 6.3. It may be noted that the mock-up tool mainly includes the Balsamiq mock-up editor. Though a specific survey was not conducted on the usability of the Balsamiq editor, the SUS survey for the mock-up specification implicitly involved questions about the usage of the mock-up system. Since the survey yielded a satisfaction rating of 73, it can be argued that the editing tool is validated as well. The usability of the auto-generated applications was successfully validated in Section 6.4 while the functional testing was validated in Section 6.5. Since the percentage values for: usability of the mock-up language, satisfaction in use of the auto-generating tool, usability of the auto-generated applications, and functional correctness of the auto-generated applications were individually well above the threshold value of 80%, it can be argued that the whole auto-generating tool has been successfully validated.

7 GENERAL DISCUSSIONS, LIMITATIONS, FUTURE DIRECTIONS AND CONCLUSIONS

The seeds of this thesis were laid with the aim to help SMEs develop web applications by following innovative developmental approaches. Chapter 1 provided the relevance of this research indicating that current software development methodologies for SME applications have several limitations. Some of the limitations are: superfluous focus on developmental processes rather than on solutions to business problems, segregation of analysis, design and development roles or unnecessarily distributing the job to teams working in tandem or untoward focus on documentation or on needless flexible designs, leading to time lags or unfulfilled requirements and increased costs. The chapter recommended using the services of BAs as developers of SME applications. Chapter 2 provided background information on the various approaches to modelling web applications with a view to find the most suitable approach for BA driven software development. Consequently, a mock-up driven auto-generating process driven by BA developers was found suitable to address the concerns raised above. This requires the design and development of an auto-generating tool to help BAs in the development process. Hence in Chapter 3 the main research question was identified - how to design a tool to help BAs to automatically develop applications for SMEs using a mock-up language. Chapter 3 also discussed DSR in IS as a suitable research method due to the innovative nature of the IT artifact expected from the research. In response to research question, MockApp, a mock-up-based tool for the auto-generation of SME applications was designed and developed using DSR in IS. MockApp required the design of a simple mock-up language for specifying the mock-up of SME applications. This was discussed in Chapter 4. MockApp also required the design of algorithms for the auto-generation of a SME application from mock-ups. Specifically, Chapter 5 provided details of how to auto-generate the database and the application logic from the mock-up. Finally,

Chapter 6 provided details of how MockApp was validated to satisfy usability concerns.

The aim of this chapter is three-fold: Firstly, to provide general discussions on how the research was conducted by using the DSR in IS checklist provided by Hevner and Chatterjee (2010b). This is done in Section 7.1. Secondly, to discuss the limitations and future directions of this research which is presented in Section 7.2. The last aim is to provide concluding remarks. Section 7.3 discusses the conclusions.

7.1 General discussions based on DSR in IS checklist questions

Checklists are frequently used to ensure that all required activities are carried out as desired. An eight-point checklist was introduced in Chapter 3 with an intention of using it on completion of the design and validation activities of this research, to ensure that all the activities in DSR in IS are indeed followed. The eight-point checklist is discussed below from a post-design perspective to provide a summary of how the various activities were conducted.

What is the research question (design requirements)?

This checkpoint question verifies whether the research question was indeed relevant to be solved by DSR in IS. It can be recalled that DSR in IS is relevant where creativity and innovations are necessary to improve the effectiveness and utility of IT artifacts in solving business problems (Baskerville, Pries-Heje & Venable 2009). In such cases DSR in IS guides the researcher to capture the knowledge created during the design process (Hevner & Chatterjee 2010a). The main research question was “How to design a tool to help BAs to automatically develop a fully functional Rich Internet Application for small to medium enterprises, holistically from visual UI requirement specifications using a visual mock-up language?”

As discussed in Chapter 1, this research was born from a need to support BAs in the development of RIAs for SMEs since modern development approaches were found to be not suitable for SME applications. Chapter 2 provided arguments suggesting that existing techniques, methods and tools are inadequate to support BAs in

developmental activities of SME applications. Specifically, from the discussions on RIA Development Using Technological tools (Section 2.4.2) and Visual Mock-up Approaches to Software development (Section 2.4.5) it was evident that the existing visual approaches have several limitations. Some of them are listed below:

- They only create prototypes that are non-functional (Hartson & Smith 1991; Panach et al. 2008)
- They require two or more models to be manually integrated (Rivero et al. 2014)
- They require high level knowledge such as graph structures or state transition diagrams to specify the requirements (Bouchrika et al. 2013; Störrle 2010)
- Most do not automatically derive the database structure and the database logic
- Designers are required to specify complete structure of business entities (Gonda & Jodal 2007; Liang, Marmaridis & Ginige 2007)
- They use non-intuitive approaches (Ramdoyal, Cleve & Hainaut 2010)

The above list of limitations meant that BAs who are traditionally considered to have weak developmental skills could not adopt existing tools for development. Hence the research aimed to provide a new tool to support BAs using a mock-up-based approach to auto-development of SME application. Auto-development of web applications solely from mock-ups is innovative in nature. Hence DSR in IS was used as a research method to guide the research.

What is the artifact? How is the artifact represented?

This checkpoint enables the researcher to verify whether the artifact is correctly identified and represented. From the discussion in Chapter 3 it is known that the artifacts can be in the form of a *construct*, a *model*, a *method*, or an *instantiation* or a combination of these.

Mock-up language *constructs* were defined in Chapter 4 and used in this research to specify the structure and behaviour of SME applications using a WYSIWYG approach. The mock-up language constructs were created keeping in mind managerial or

business stakeholders' (that is BA) perspectives. This was done to ensure BAs could create a holistic visual model of the SME application without being concerned about complex technical issues such as the internal architecture, database design and application design of client and server-side components.

Algorithms (*methods*) were also produced as artifacts in this research. Specifically, in Chapter 5 two types of algorithms were discussed, namely algorithms for the auto-generation of database structure from a mock-up and the algorithms for the auto-generation of client-server application logic and database logic. The auto-generating nature of these artifacts ensured that BAs do not need to have complex technical (developmental) skills normally associated with designers, to develop SME applications. The final and possibly the most important form of artifact produced is the fully functional and usable auto-generated SME application (*instance*) from the mock-ups.

What design processes (search heuristics) were used to build the artifact?

Hevner, March and Ram (2004) note that the search for an effective artifact in DSR requires exploitation of available “means” to reach expected outcomes (“ends”) while complying with the design requirements. A summary of the requirements used to design the artifacts are listed below:

- (i) The tool should be suitable for SME applications. This led to the identification of SME application requirements. The essential features of web applications for SMEs were found by researching examples used in existing literature on modelling of business web applications. The essential features were discussed in Section 4.1 in Chapter 4.
- (ii) The visual mock-up language should be cognitively easy to use by BAs who possess rich business requirement gathering skills but low developmental skills. This requirement helped in identifying the features of the visual mock-up language. Further details of the expected features were provided in Section 4.2 to Section 4.4 in Chapter 4.
- (iii) The tool should have rich features to easily create and edit visual mock-ups of SME applications. This was found by studying popular commercially available

mock-up editors and identifying common features of such tool. Further details of such tools were discussed in Section 4.5 in Chapter 4.

- (iv) The tool should be capable of auto-generating fully functional SME applications holistically from the visual mock-up. Further details of how auto-generation of database schema, client-server application logic and database logic were discussed in Chapter 5.
- (v) The tool and the auto-generated application should be usable. The usability study of the tool and the auto-generated application was performed by searching for standard approaches to field-testing. Chapter 6 contains details of how this was successfully performed.

How are the artifact and the design processes grounded by the knowledge base?

This checkpoint enables a design science researcher to verify that the design of the artifacts is based on existing *constructs, models, methods or instantiations*. This research is grounded on three broad areas of existing knowledge, namely, mock-up language models, design & development of the tool and validations. Under mock-up language and models a meta-model for UIs modelling of web applications is considered while design and development of the tool considered architectural models, development approaches and model driven engineering approaches for the grounding of the design process. Finally, the validation of the design tool is grounded on sound ISO principles for usability testing of web applications. Figure 56 provides a summary of the knowledge base on which this research was conducted during the relevance cycle and design cycle activities of DSR. In Figure 56 the relevance cycle activities are shown in orange text and the design cycle activities are shown in blue text. In addition, the arrows in the figure indicate the knowledge required to answer the various research sub-questions. Some of the important sources of knowledge, instruments, tools, methods or constructs required for this research is summarized below.

Valverde and Pastor's (2009) RIA UI Meta-Model discussed in Section 2.4.3 is the foundation on which the mock-up language was built. Specificities of how the RIA UI meta-model was used along with Balsamiq mock-up editor is discussed in Chapter 4.

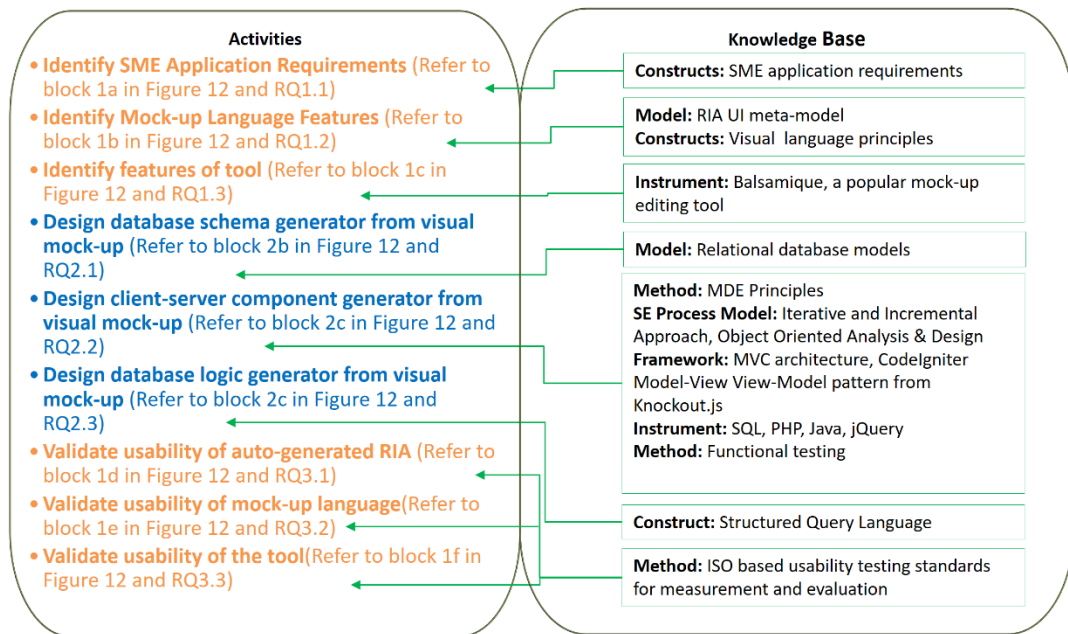


Figure 56: Knowledge base of this research

Model Driven Engineering (MDE) principles discussed in Section 2.4.4 are used for the auto-generation of the RIA from the visual mock-up. Specifically, the visual mock-up model created by BAs was used both as a Computational Independent Model as well as Platform Independent Model during the MDE process of the RIA.

The design and development of the tool was carried out by following the Iterative and Incremental development process. In addition the Object Oriented Analysis and Design method (Jacobson, Booch & Rumbaugh 1999) was used to auto-generate interacting classes and objects required for the client-server components referred to in Figure 56. The Model Driven Engineering instruments used for the generation of the Platform Specific Models of the Data Model Generation component include Structured Query Language as a Relational Database Management System language using MySQL as a database management system for the auto-generated RIA. Similarly, the instruments used for the RIA MVC-MC Generation component include JQuery and Knockout.js for client-side code of auto-generated RIA, Java™ for the coding the auto-generating tool, PHP and CodeIgniter for server-side code of the

auto-generated RIA and html. Specific details of how these instruments were used is discussed in Chapter 5.

Finally, the validation of the auto-generated application and the visual modelling language and consequently the tool was guided by ISO/IEC evaluation standards to ensure that the researcher has followed standard principles and methods for acceptance of the auto-generating tool.

What evaluations are performed during the Internal Design Cycles? What design improvements are identified during each design cycle?

This checkpoint ensures that evaluation of the design is carried on an on-going basis to provide design improvements as the design cycle activities are iterated. Table 8 in Chapter 6 highlighted some of the DSR in IS strategies for evaluations. This research used *analytical*, *experimental*, *testing* and *descriptive* strategies during the internal design cycle.

Static analysis, architecture analysis and dynamic analysis was used as a part of the *analytical* strategy to settle on the MVC-MC architecture for the generated application. Please refer to Sections 2.2.1 and 2.2.2 to review why MVC-MC architecture was considered as an optimal architecture for modern web applications following an analysis of several web application architectures.

Descriptive scenarios of SME applications' usage were utilized to appraise the utility of the mock-up language features in supporting the common features of SME applications, during the design process. Section 4.2 in Chapter 4 discusses how the scenarios were used.

Within the *experimental* strategy, controlled experiments were performed by employing usability inspectors to ascertain the usability of the mock-up language and the generated application. The *testing* strategy was employed in the form of functional and structural testing to discover functional failures and non-coverage of all execution paths during the implementation of each increment in the Iterative and Incremental Development process. In addition, a trial usability test run of the completed system was performed by a usability tester. The trial test run was

performed by a professional Business Analyst with a master's degree in Computer Science. During the trial, the tester designed a mock-up for a SME test case written by him. However, the auto-generator was not able to produce the expected outcomes because the auto-generating components were found to be inflexible to satisfy behaviours not envisaged in the Travel Deals example. The inflexibility was because the auto-generator attempted to address all types of behaviour together in a single pass of the mock-up model. This meant that managing unforeseen scenarios in the mock-up model was a challenging task. For example, an activity in the Travel Deal case study is for an existing entity (such as a Travel Deal) to be linked to a newly created entity such as Customer; however, the trial case study required two pre-existing entity types to be linked. The old design of the auto-generator was not capable of doing this without making major changes to the auto-generator. An important lesson learnt from the usability test of the trial case study was that the design should be flexible to manage future changes seamlessly. This resulted in a major redesign and re-development of the auto-generator in a three-month effort. The re-designed artifact could adapt to new business activities because it considers each type of behaviour in a new pass over the mock-up model while auto-generating the web application. The new design ensures that existing algorithms are not affected by new behaviours in the future.

Feedback from international conference presentations and from journal articles was mainly used to identify the field-testing mechanism for the tool. The initial approach was to compare the tool with other similar tools by conducting surveys following usability testing by testers. However, this approach was rejected because it is solely based on testers' perceptions which are subjective to the level of experience of the testers. So, an objective approach was chosen instead where usability testers' actual actions were analysed to produce a true indication of the tool's usage. This was followed by a survey to understand the satisfaction rating of the tool.

Controlled experiments were carried on as part of the *experimental* strategy for evaluating the usability of the mock-up language and the generated application on completion of the re-design following the trial run of the artifact. Nine usability

inspectors were employed to perform field-testing by BAs. The summary of how the artifact was field-tested are discussed in the next sub-section.

How was the artifact introduced into the application environment and how was it field-tested? What metrics were used to demonstrate artifact utility and improvement over previous artifacts?

This checkpoint of DSR in IS addresses how the researcher has ensured that the auto-generating tool is useful to the world. In other words, it attempts to answer research question 3: How is the auto-generating tool validated? Validation is considered from the point of view of the usability of the mock-up language, usability of the auto-generated application and consequently the whole tool, as perceived by BAs. ISO standards were used to measure usability. The ISO/IEC 9126 standard identifies usability testing as validating the quality-in-use (Casteleyn et al. 2009; Molina & Olsina 2008). Quality-in-use was measured with respect to the effectiveness, efficiency, and satisfaction that users gain while interacting with the application in a real user environment (Casteleyn et al. 2009; Lew et al. 2012). Quality-in-use was conducted by nine BAs trained as usability inspectors, since it was a challenge to find bona fide business users to field-test this academic research project due to financial and time constraints. Moreover, using trained usability testers is widely accepted for field-testing (Casteleyn et al. 2009; Molina & Olsina 2008).

Following the usability testing of the mock-up language, each tester completed a ten-question survey on satisfaction in use of the mock-up language, by employing a popular System Usability Scale (SUS). SUS was chosen because it is widely used for evaluation of many software and is easy to fill and calculate (Covella & Olsina 2006).

The validation measurements yield percentage values between 0 and 100 for effectiveness in use, efficiency in use and satisfaction in use, where higher values are better indicators of acceptability of the system. These values were then compared with acceptable threshold values to ascertain whether the tool is valid or not for use by BAs. The results of the analysis of the usability testing values indicated that BAs were successful in accomplishing the expected tasks for the creation of mock-ups to satisfy the requirements of SME applications. Similarly, the auto-generated

applications were found to be functionally correct and usable by BAs. Finer details of the implementation of the measurement of the usability of the mock-up language and the auto-generated applications are available in the Chapter 6.

What new knowledge was added to the knowledge base and in what form?

The knowledge gained from this research has been regularly disseminated to the wider world by presenting the research at various international conferences and by publishing in reputed journals. A paper on an optimal RIA architecture containing MVC components on the client side and MC components on the server side was presented at the 2010 International Conference on Computer and Software Modelling(D'Souza & Ginige 2010). This was followed by a paper on user-friendly UI modelling language for RIA development, presented at the 7th International Conference on Software Paradigm Trends in 2012(D'Souza, Ginige & Liang 2012). In 2013 a paper on visually modelling data intensive web applications was published by ACM in the proceedings of the 6th International Symposium on Visual Information Communication and Interaction(Deufemia, D'Souza & Ginige 2013).

Two journal papers were also published based on the findings from this research. The first one was on managing user access control through mock-up. This was published in the International Journal of Software Engineering and Knowledge Engineering in 2015 (Caruccio et al. 2015). The second journal article was on enabling the generation web applications from mock-ups and published in the Journal of Software and Practise (D'Souza et al. 2018).

Has the research question been satisfactorily addressed?

The research question has been answered by answering the nine sub-questions introduced in Chapter 3. Specifically, Section 4.1 provided answers to RQ1.1 regarding the generic requirements of SME applications. Section 4.2 through to Section 4.4 provided answer to RQ1.2 regarding the features of the mock-up language to express the requirements. Section 4.5 answered RQ1.3 regarding how the features of the tool were used for integration with the features of the language. Section 5.1 answered RQ2.1 regarding the derivation of the database structure from

a visual mock-up. Section 5.2 to Section 5.6 provided answers to RQ2.2 and RQ2.3 in terms of algorithms for the auto-generation of MVC-MC components and the associated CRUD operations on the database, to support the common features of SME applications. The issue of “satisfaction” while addressing the research question is covered in RQ3.1 to RQ3.3. Specifically, Section 6.3 discussed how usability of the mock-up language was validated, as an answer to RQ3.1. Section 6.4 discussed how usability of the auto-generated application was validated, as an answer to RQ3.2. Finally, Section 6.6 provided an analytical discussion based on results of the usability test of the generated application and usability test of the mock-up language, as an answer to RQ3.3. Table 26 through to

Table 28 provides a summary of how research sub-questions 1, 2 and 3 have been answered by providing links to the relevant discussions topics.

Table 26: Cross checking research question 1 with findings

RQ1: What is a suitable visual mock up language to fully capture the SME application requirements?	
Sub Questions	Findings and Discussion Links
RQ1.1: What are the generic requirements of SME applications?	Section 4.1 contain details on how the following operations of SME applications were identified for: creating business entities, searching, search result management, updating, deleting and report generation.
RQ1.2: What are the features of a visual mock-up language to fully express the requirements?	Section 4.2 through to Section 4.4 contain details of mock-up language features to express the mock-up in terms of appropriate: <ul style="list-style-type: none"> • container structure types for the various user interface sections of the application • behaviour associated with navigations of the application
RQ1.3: How are the features integrated into a tool?	Section 4.5 contains a discussion on features of popular commercial mock-up editing tools for easy integration with the features of the language. Some of the desirable features of the tool for editing of the mock-up were identified as, ability to: <ul style="list-style-type: none"> • drag and drop of widgets • group and ungroup widgets • place a group of widgets in a container • copy, paste, cut, undo actions • annotate navigation widgets to express application behaviour • produce and export mock-up in XML or JSON form

Table 27: Cross checking research question 2 with findings

RQ2: How is a RIA for a SME auto generated from a visual mock-up?	
Sub Questions	Findings and Discussion Links
RQ2.1: How is the database structure of the RIA auto-generated from a visual mock-up?	Section 5.1 discusses algorithms for the auto-generation of: database tables, fields within each table, field data-type, relationships among tables and multiplicities in the relationships.
RQ2.2: How are the client side and server-side components of the RIA auto-generated from a visual mock-up?	Section 5.2 to Section 5.6 discuss algorithms for the auto-generation of MVC-MC components to support the common features of SME applications identified by RQ1.1.
RQ2.3: How is the database logic for Creation, Update, Delete and Retrieve operations auto-generated from a visual mock-up?	Section 5.2 to Section 5.6 also discuss algorithms for the derivation of Create, Retrieve, Update and Delete operations on the database.

Table 28: Cross checking research question 3 with findings

RQ3: How is the auto-generation tool validated?	
Sub Questions	Findings and Discussion Links
RQ3.1: How is the usability the mock-up language validated?	Section 6.3 discusses how usability of the mock-up language was validated by BAs while performing usability testing using C-INCAMI framework, for the common features SME applications, for the common features of SME applications. Satisfactory results were obtained for usability of the mock-up language.
RQ3.2: How is usability of the auto-generated RIA validated?	Section 6.4 discusses how usability of the auto-generated application was validated by BAs while performing usability testing using C-INCAMI framework which is based on ISO standards. Satisfactory results were obtained for usability of the auto-generated RIAs.
RQ3.3: How is the usability of the tool validated?	Section 6.6 provides an analytical discussion based on results of the usability test of the generated application and usability test of the mock-up language.

7.2 Limitations and future directions of the research

Any research has its own sets of limitations which provides opportunities for future directions for the research. Similarly, though this research has led to the successful

development of a mock-up language and the design and development of an auto-generator to help BAs to upskill to develop SME applications, there are several limitations of this research. These can broadly be related to: mock-up, database, technology, validation and evolution.

Mock-up related limitations are: (i) The researcher tacitly assumes that BAs can apply their requirements skills to create mock-up designs though this may be cognitively challenging at times. (ii) The features of the mock-up language are minimalistic in nature and may need to be extended to incorporate complex business operations. (iii) Currently multiple search result sets cannot be selected for subsequent processing rather it only allows a single search result set to be selected at time. (iii) user access control has not been considered in the mock-up design though this has been explored in a journal paper co-authored with researchers from University of Salerno, Italy.

Database related limitations are: (i) The data types of database fields are assumed from the names of the widgets which may potentially cause computational problems when complex mathematical operations are necessary. (ii) All database table relationships are assumed to be many-many which may lead to an inefficient database design.

Technology related limitations are: (i) The auto-generated application is not mobile friendly. (ii) It is not completely web service compliant. (iii) The auto-generated applications do not use Object Relational Mapping frameworks for easy integration with several types of database engines. (iv) Limited styling is applied to the auto-generated application.

Validation related limitations primarily deal with: (i) The environmental settings for conducting the usability testing of the language and the auto-generated system. (ii) The validation was not performed for real-world SME organizations due to the limitations discussed in Section 6.

In addition, the support for the automatic evolution of the generated application is limited. This may impact the acceptance of the tool in a real-world setting.

These limitations are discussed in further details along with the opportunities they provide for future research directions in the following sub sections.

7.2.1 Mock-up limitations and opportunities for future research

As stated earlier BAs have excellent requirement skills but it could be wrong to assume that they can apply those skills to create mock-up designs since designing mock-ups can be a cognitively challenging task. For example though Bruner, Goodnow, and Austin (cited in Bell 2008) in their book, *A Study of Thinking*, suggest that the most effective mechanism in visualising concepts into distinguishable entities is the act of invention and human creativity, it is ironic since visualizing intangible entities requires the manipulation of concepts that are not concrete and hard to understand at times. Invention and human creativity is a learning activity and may require embracing of methodological approaches to identify business entities, relationships among business entities and relating them to operational requirements of SMEs. Hence this limitation provides opportunities to explore ways in which BAs may need to be trained in various approaches to manage the mock-up of web applications that require highly interactive behaviour. However, it would be fair to assume that BAs could create mock-ups of most SME applications, since the entities (business concepts) closely resemble their corresponding real-world objects such as invoices, addresses etc and if the BAs can think abstractly in terms of higher order notations followed in modelling notations such as UML, the mock-up notation would be easier to visualize since it follows the What You See Is What You Get approach.

The other limitation regarding the minimalistic features of the language hindering creation of mock-ups for complex operations can also be regarded as its strength since minimalism enables BAs to quickly learn the features. However, a future version of the language could be considered in terms of must-have and nice-to-have features. For example mock-up feature for user-access control is a must-have feature that is not defined in this thesis, though it is considered in the journal paper written collaboratively with Caruccio et al.(2015). Similarly, currently the mock-up notation does not consider selection of multiple search result entity sets for subsequent processing, rather it only considers one result set to be processed at a time. This is

another must-have feature, since it can be tedious to perform the same type of operations many times if multiple result sets are required to be processed. However, a feature such as specification of client-side pagination and or server-side pagination could be considered as a nice to have feature. Currently all search results are downloaded to the browser and pagination is only performed on the client side. This may have a detrimental effect on page loading efficiencies if considerable number of search result sets are generated. A future version of the mock-up system could handle pagination automatically depending on the size and number of result sets generated. The mock-up notation may not change but the auto-generating system could have code to manage the pagination partly on the server side and partly on the client-side. Server-side pagination implementation is desirable when large data sets are involved to limit the resources required by RIA clients. The number of records to display at a time in a page can be set automatically by the server side if not specified during the design of the page. Then the server-side code can calculate the number of pages required and transfer a pool of XML or JSON resources to be displayed in a page during each AJAX call from the client side. For example, if the page size is five and if a total of 500 records are generated by the server-side code, then during the first AJAX call assume a first block of 100 records are transferred though only 5 records will be displayed at a time. The client can keep track of the current page number and could have an algorithm to automatically trigger a new AJAX call when it senses the user is nearing the limit of the block of records in its storage. Each AJAX call could explicitly specify the next block number of records to be transferred. Finally, when the client moves away from the page, an AJAX call could be made to remove the result set from the server side. However, if the data set on the server side is not large, then the pagination could be solely handled on the client side. This means that the server code should have the capability to inform the client when server-side pagination is necessary and client side should have code for managing requests for either client-side pagination or server-side pagination.

Another limitation of the current approach is that the implementation of the “**WSRequest**” notation discussed in Section 4.3 is not complete. Hence the current version of the system does not support web service invocations.

7.2.2 Database limitations and opportunities for future research

It may be recalled from the discussion in Section 5.1 that the database tables of the auto-generated system are derived from the mock-up of the *Database Field Yielding Containers* and the relationships among the tables are derived from annotated navigational widgets or from nested *Database Filed Yielding Containers*. Also note the mock-up does not include data type information in *Database Field Yielding Widgets* and the data type of the fields of the database tables are assumed from the names of the widgets. This may potentially cause computational problems when complex mathematical operations are necessary. Thus, for example if the label of a *Database Field Yielding Widget* is “date of birth”, then a “date” data type is assumed, whereas if it is “customer name” then a “string” type is assumed. Similarly, other types are identified. However, if the system is not able to identify the datatype from the label, it defaults to “string” type. Consider an example of a “Dating” enterprise, that enables people to find partners. The mock-up of a *Database Field Yielding Container* for such a system may have labels such as “date name”, “date time”, “date location”. Since all the labels have “date” in them, the data type of the database fields for all three labels could be “date”. Clearly this will result in wrong values to be stored in the database tables and can lead to run time exceptions if computations are carried on using the assumed types. This limitation provides opportunities to optionally provide data type information in data input widgets within *Database Field Yielding Containers* during the design of the mock-up.

Another limitation concerning the derivation of database table relationships is that all relationships are assumed to be many-many which may lead to inefficient databases. For example, if one “Customer” can make many “Payments” but if each “Payment” can be made by one “Customer”, then a many-one relationship exists. However, since relationship multiplicities are automatically derived and not manually specified in the mock-up, the system assumes all relationships are many-many which may lead to unnecessary junction tables, making the database design in-efficient. This limitation can be overcome by extending the mock-up language to optionally allow

specification of the multiplicities on navigation widgets linking appropriate *Database Field Yielding Containers* or between nested *Database Field Yielding Containers*.

7.2.3 Technological limitations and opportunities for future research

Most of the modern enterprise applications are mobile friendly. That is the View layer of the application should render the display gracefully whether on a desktop or a mobile device. This is called Responsive Web Design (Marcotte 2010). W3Schools, a popular site for web application learning, states “Responsive Web Design is about using CSS and HTML to resize, hide, shrink, enlarge, or move the content to make it look good on any screen” (*HTML Responsive Web Design* 2017 para 1). The auto-generated application uses a rudimentary CSS that does not incorporate all the features required for responsiveness. Hence the look and feel of the auto-generated application is not mobile friendly. This provides another avenue for future research by incorporating styling information in mock-up design.

Another feature of modern applications is to use of web services. Research conducted by business applications specialist Compuware reveals 53% of SMEs attribute increased revenues as a key business benefit of the technology compared with 40% of large organizations (Beckett 2002). Interoperability, usability, reusability and deployability are considered as some of the advantages of web services. Web services typically use standards-based communications methods for better interoperability with web applications, consequently leading to longer life-spans and offering better return on investment (Johnko 2007). Usability and reusability are improved because web services allow exposure of many types of business logic over the web. Hence web applications using them do not need to re-invent the wheel. The mock-up language for example can use the “**WSRequest**” keyword to specify communication with a web service at runtime. This feature can further reduce the time lag for development of SMEs applications since some of the services are provided by interoperable and reusable web services deployed over standard internet technologies. However, this feature has not been implemented. Hence this research can be extended by making the auto-generated application web service compliant.

Furthermore, the auto-generated applications do not use modern technologies such as Object Relational Mapping (ORM) frameworks for easy integration with several types of database engines. ORM is a powerful method for designing and querying relational database models at a conceptual level, where the application is described in terms easily understood by non-technical users. A large part of the auto-generated code on the server side of most enterprise applications deal with transferring business objects in and out of a relational database. Modern applications exploit the services of ORM framework, such as Doctrine for PHP¹³, to declaratively define the mapping between the server-side object model and relational database schema and express database-access operations in terms of objects (Richardson 2009). This high-level approach not only reduces the amount of database access code that needs to be auto-generated but also makes the system more robust in supporting several types of database engines without any extra effort. This limitation provides another avenue for future research.

7.2.4 Testing limitations and opportunities for future research

Due to the challenges of conducting usability testing based on real-world settings, discussed at the beginning of Section 6, the usability of the mock-up language and the auto-generated applications were tested by Business Analysts with low experience in SME set-up though they had rich experience in research environments. Secondly, the testing was not conducted for real world enterprises. Thirdly the testing was conducted by usability inspectors in the university's laboratory rather than by actual users in a SME organization. In future, this research will be strengthened by employing real BAs for real-world SME requirements for validating the tool.

7.2.5 Adaptability limitations and opportunities for future research

Finally, another limitation is the support for evolution of the generated applications for future requirements. There are two aspects of the support for evolution, automatic evolution and manual evolution.

¹³ <http://www.doctrine-project.org/projects/orm.html>

Support for automatic evolution refers to the built-in features of the tool for automatically changing the features of the auto-generated application. Consider the scenario where a SME needs new features for an existing application. This may result in altering client side MVC components, server-side MC components as well as database schema and logic. Alterations can be accomplished by automatically transforming the Computationally Independent Models, Platform Independent Models and the Platform Specific Models (discussed in Section 2.4.4) when any of them is updated manually. However, the change should not affect legacy code and data. Thus, for example if the mock-up design is changed, the new database schema should yet work with the old data. In future, this could be done by migrating and transforming legacy data to support the new database schema. This may require certain policy restrictions on what forms of evolutions are possible. For example, a deletion of an existing *Database Field Yielding Container* specification may not be allowed unless there is no data stored relating to that container. On the other hand, adding new input widgets in an existing DFY Container could be allowed since it does not corrupt existing data. These changes may be incorporated by using inheritance principles in the auto-generated code. However, the current version of the application does not support automatic evolution.


Support for manual evolution refers to adaptive maintenance of the code. The auto-generated code may need to be adapted to deal with new features not originally included in the visual specifications. The maintenance of the auto-generated code requires the knowledge of the dependent technologies such as Knockout.js, JQuery, JavaScript, CodeIgniter, PHP, SQL and MySQL. In addition, knowledge of the MVC-MC architecture is also important. The base class of the Server-Side Controller code has simple public API's for creation, search, update and delete operations. These public API's return JSON encoded strings. The public functions identify the type of Models required for processing client queries from the table name in the requested data. Refer to Section 5.2 through to 5.6 for details about how Server-Side Models and their behaviour are derived. The public API's in turn invoke appropriate Model class methods. However, the Model classes do not have auto-generated code to manage complex functionality. For example, if a business requirement is that a 10% discount

is to be applied for travel deal orders worth exceeding \$10000, then this would require manually extending the code of the appropriate Server-side Models. Thus, a "Payment Details" class for example would need a method to manage the discount. Future versions of the visual mock-up language will require a provision to auto-generate stubs for such functionality for maintenance by a software engineer. Correspondingly the mock-up would require a keyword such as "function" to trigger the creation of the stub. Figure 57 illustrates how this could be done. It uses a =function(applyDiscount) notation in the mock-up of the "Payment Details" container to trigger the creation of an applyDiscount stub in the "Payment Details" Server-Side Model.

Payment Details

Card Number

CV_CVV

Card Expiry Date 

Card Type

Visa

MasterCard

American Express

=function(applyDiscount)

Figure 57: Using potential "function" keyword in future versions

7.3 Conclusions

This research has demonstrated that BA expertise can be utilized not only for the activities of requirements gathering and requirements specification but also for design and development activities of SME application. To demonstrate this a mock-up language and an auto-generating tool was designed and developed since no help was available to BAs to seamlessly harness their rich requirements gathering skills for developmental activities. The process of designing the mock-up language has led to

abstractions of the desired features of SME applications in the form of visual specifications. It also led to innovative ways of viewing mock-ups as a source for identifying database schema of the system. That is the graphical user interface views of the system can be used as a source to derive the models of the systems instead of the traditional way of designing the domain models independently of the UI design. Similarly, this research also demonstrated that mock-ups need not be used purely as a passive artifact for wireframing the UI design but can be used as an active artifact of the system in a Model Driven Engineering approach to automatically develop fully functional applications. This is made possible partly by embedding annotations in the mock-up. In addition to the mock-up language features, the algorithms used in the auto-generator for deriving database and the application logic form a rich set of foundational knowledge to pursue this research for modern areas such as mobile application development, web service development and for evolutionary systems.

Looking at the bigger picture, this research recommends a fresh approach in to web engineering, purely in terms of a holistic annotated mock-up model that is easy to comprehend by business stakeholders. A holistic WYSIWYG mock-up model can sufficiently capture the requirements - business entities, the relationships among them as well as behavioural logic, requiring no separate models for database design, presentation design, navigation design and application logic design. The mock-up language and the algorithms in the auto-generating tool produced in this research can be extended to support capturing the requirements and software engineering of large enterprise applications.

8 REFERENCES

- Abrahamsson, P., Salo, O., Ronkainen, J. & Warsta, J. 2002, *Agile Software Development Methods: Review and Analysis*, VTT, Espoo.
- Antonelli, C. & Scellato, G. 2015, 'Firms size and directed technological change', *Small Business Economics*, vol. 44, no. 1, pp. 207–18.
- Architecture Board ORMSC 2001, *Model Driven Architecture (MDA)*. Document number *ormsc/2001-07-01*, viewed 8 December 2015, <<http://www.omg.org/cgi-bin/doc?ormsc/2001-07-01>>.
- Baskerville, R., Pries-Heje, J. & Venable, J. 2009, 'Soft design science methodology', *proceedings of the 4th international conference on design science research in information systems and technology*, ACM, p. 9, viewed 8 February 2016, <<http://dl.acm.org/citation.cfm?id=1555631>>.
- Becker, P. & Olsina, L. 2010, *Towards support processes for web projects*, Springer, viewed 19 May 2016, <http://link.springer.com/chapter/10.1007/978-3-642-16985-4_10>.
- Beckett, H. 2002, 'The business benefits of Web services', *ComputerWeekly*, May, viewed 25 July 2017, <<http://www.computerweekly.com/feature/The-business-benefits-of-Web-services>>.
- Bell, M. 2008, *Service-Oriented Modeling (SOA) Service Analysis, Design, and Architecture*, Wiley, Chichester, viewed 24 July 2017, <http://west-sydney-primo.hosted.exlibrisgroup.com/primo_library/libweb/action/dlDisplay.do?vid=UWS-ALMA&docId=UWS-ALMA51203935620001571>.
- Bernroider, E. & Koch, S. 2001, 'ERP selection process in midsize and large organizations', *Business Process Management Journal*, vol. 7, no. 3, pp. 251–7.
- Birley, S. & Norburn, D. 1985, 'Small vs. Large Companies: The Entrepreneurial Conundrum', *Journal of Business Strategy*, vol. 6, no. 1, pp. 81–7.
- Boehm, B.W. 1988, 'A spiral model of software development and enhancement', *COMPUTER*, vol. 21, no. 5, pp. 61–72.
- Bouchrika, I., Ait-Oubelli, L., Rabir, A. & Harrathi, N. 2013, 'Mockup-based navigational diagram for the development of interactive web applications', *Proceedings of the 2013 International Conference on Information Systems and Design of Communication*, ACM, pp. 27–32, viewed 14 January 2016, <<http://dl.acm.org/citation.cfm?id=2503864>>.
- Bozzon, A., Comai, S., Fraternali, P. & Carughi, G.T. 2006a, 'Capturing RIA Concepts in a Web Modeling Language', *Proceedings of the 15th international conference on World*

- Wide Web*, ACM, pp. 907–908, viewed 18 December 2015, <<http://dl.acm.org/citation.cfm?id=1135938>>.
- Bozzon, A., Comai, S., Fraternali, P. & Carughi, G.T. 2006b, 'Conceptual Modeling and Code Generation for Rich Internet Applications', *Proceedings of the 6th international conference on Web engineering*, ACM, pp. 353–360, viewed 18 December 2015, <<http://dl.acm.org/citation.cfm?id=1145649>>.
- Brambilla, M. & Fraternali, P. 2014, 'Large-scale Model-Driven Engineering of web user interaction: The WebML and WebRatio experience', *Science of Computer Programming*, vol. 89, pp. 71–87.
- Brogneaux, A.F., Ramdoyal, R., Vilz, J., Hainaut, J.-L. & Grandgagnage, R. 2005, 'Deriving User-Requirements from Human-Computer Interfaces.', *Databases and Applications*, pp. 77–82, viewed 14 January 2016, <https://pure.fundp.ac.be/portal/files/226150/IASTED_DBA_05.pdf>.
- Brooke, J. 2001, *SUS - A quick and dirty usability scale*, Digital Equipment Corporation, Reading, UK, viewed 27 June 2016, <<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.393.3882&rep=rep1&type=pdf>>.
- Busch, M. & Koch, N. 2009, *Rich internet applications: State-of-the-Art*, Technical Report, Institute for Informatics, Ludwig-Maximilians-Universität, München, Germany, viewed 18 December 2015, <http://uwe.pst.ifi.lmu.de/publications/maewa_rias_report.pdf>.
- Caruccio, L., Deufemia, V., D'Souza, C., Ginige, A. & Polese, G. 2015, 'A Tool Supporting End-User Development of Access Control in Web Applications', *International Journal of Software Engineering and Knowledge Engineering*, vol. 25, no. 02, pp. 307–31.
- Casteleyn, S., Daniel, F., Dolog, P. & Matera, M. 2009, *Engineering Web Applications*, Springer Berlin Heidelberg, Berlin, Heidelberg, viewed 19 May 2016, <<http://link.springer.com/10.1007/978-3-540-92201-8>>.
- Ceri, S., Fraternali, P. & Bongio, A. 2000, 'Web Modeling Language (WebML): a modeling language for designing Web sites', *Computer Networks*, vol. 33, no. 1, pp. 137–157.
- Chaffey, D. 2011, *E-business & E-commerce Management: Strategies, Implementation and Practice*, 5th edn, Pearson/Financial Times Prentice Hall, viewed 11 December 2015, <https://books.google.com.au/books/about/E_business_E_commerce_Management.html?id=E6NktwAACAAJ>.
- Chen, J.Q. & Heath, R.D. 2005, 'Web Application Development Methodologies', *Web Engineering: Principles and Techniques*, IGI Global, Hershey PA and London, pp. 76–96.
- Cohn, M. 2004, *User stories applied: For agile software development*, Addison-Wesley Professional.

- Cormode, G. & Krishnamurthy, B. 2008, 'Key differences between Web 1.0 and Web 2.0', *First Monday*, vol. 13, no. 6, viewed 14 December 2015, <<http://firstmonday.org/ojs/index.php/fm/article/view/2125>>.
- Covella, G.J. & Olsina, L.A. 2006, 'Assessing Quality in Use in a Consistent Way', *Proceedings of the 6th international conference on Web engineering*, ACM, pp. 1–8, viewed 19 May 2016, <<http://dl.acm.org/citation.cfm?id=1145583>>.
- Coyette, A. & Vanderdonckt, J. 2005, 'A Sketching Tool for Designing Anyuser, Anyplatform, Anywhere User Interfaces', *Human-Computer Interaction-INTERACT 2005*, Springer, pp. 550–564, viewed 14 January 2016, <http://link.springer.com/chapter/10.1007/11555261_45>.
- Curtis, B., Krasner, H. & Iscoe, N. 1988, 'A field study of the software design process for large systems', *Communications of the ACM*, vol. 31, no. 11, pp. 1268–1287.
- Davenport, T.H. 1998, 'Putting the enterprise into the enterprise system', *Harvard business review*, vol. 76, no. 4, viewed 11 March 2016, <https://jps-dir.com/forum/uploads/12967/Davenport_1998.pdf>.
- Deufemia, V., D'Souza, C. & Ginige, A. 2013, *Visually modelling data intensive web applications to assist end-user development*, ACM Press, p. 17, viewed 29 June 2017, <<http://dl.acm.org/citation.cfm?doid=2493102.2493105>>.
- D'Souza, C., Deufemia, V., Ginige, A. & Polese, G. 2018, 'Enabling the Generation of Web Applications from Mockups', *SOFTWARE—PRACTICE AND EXPERIENCE*.
- D'Souza, C. & Ginige, A. 2010, 'MVC-MC: A rich internet application architecture for optimal separation of concerns', *Proceeding of the Int. Conf. Computer and Software Modeling, 2010*, Manilla, pp. 78–82.
- D'Souza, C., Ginige, A. & Liang, X. 2012, 'End-user friendly UI modelling language for creation and supporting evolution of RIA', *Proceedings of the 7th International Conference on Software Paradigm Trend*, SciTePress - Science and Technology Publications, Rome, Italy, pp. 190–8.
- Ebase Technology - Getting Started* 2005, viewed 21 March 2017, <http://www.ebasetech.com/ebase/XI09_GETSTARTED.eb?ebd=1&ebp=10&ebz=1_1490136932508>.
- Exploiting the Software Advantage - Lessons from Digital Disrupters* 2015, Freedom Dynamics Ltd, viewed 26 November 2015, <http://rewrite.ca.com/content/dam/rewrite/files/White-Papers/Exploiting%20the%20Software%20Advantage_final.pdf>.
- Fernández, D.M. & Penzenstadler, B. 2015, 'Artefact-based requirements engineering: the AMDiRE approach', *Requirements Engineering*, vol. 20, no. 4, pp. 405–34.

- Ferreira, J., Noble, J. & Biddle, R. 2007, 'Agile Development Iterations and UI Design', *Agile Conference (AGILE), 2007*, IEEE, pp. 50–58, viewed 14 January 2016, <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4293575>.
- 'Foundations of UX: Usability Testing' 2015, lynda.com, viewed 19 May 2016, <<http://www.lynda.com/User-Experience-tutorials/Foundations-UX-Usability-Testing/421803-2.html>>.
- Fraternali, P., Comai, S., Bozzon, A. & Carughi, G.T. 2010, 'Engineering rich internet applications with a model-driven approach', *ACM Transactions on the Web*, vol. 4, no. 2, pp. 1–47.
- Garrigós, I., Meliá, S. & Casteleyn, S. 2009, 'Adapting the Presentation Layer in Rich Internet Applications', in M. Gaedke, M. Grossniklaus & O. Díaz (eds), *Web Engineering: 9th International Conference, ICWE 2009 San Sebastián, Spain, June 24-26, 2009 Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 292–9.
- GeneXus Overview* n.d., viewed 2 January 2018, <<https://training.genexus.com/overviews/genexus?en>>.
- Ghobadian, A. & Gallear, D. 1997, 'TQM and organization size', *International Journal of Operations & Production Management*, vol. 17, no. 2, pp. 121–63.
- Ginige, A. 2010, 'Meta-Design paradigm based approach for iterative rapid development of enterprise web applications', *Proceedings of the Fifth International Conference on Software and Data Technologies*, SciTe Press, Portugal, pp. 337–43.
- Ginige, A. 2003, 'Re-engineering Software Development Process for eBusiness Application Development.', *SEKE*, Citeseer, pp. 1–8, viewed 10 January 2016, <<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.83.7233&rep=rep1&type=pdf>>.
- Gonda, B. & Jodal, N. 2007, *KNOWLEDGE-BASED DEVELOPMENT - Philosophy and Theoretical Foundations of GeneXus*, viewed 7 December 2015, <<http://www.genexususa.com/files/knowledge-based-development-philosophy-and-theoretical-foundation-of-genexus?en>>.
- Grigera, J., Rivero, J.M., Luna, E.R., Giacosa, F. & Rossi, G. 2012, 'From requirements to web applications in an agile model-driven approach', *Web Engineering*, Springer, pp. 200–214, viewed 13 January 2016, <http://link.springer.com/chapter/10.1007/978-3-642-31753-8_15>.
- Hartson, H.R. & Smith, E.C. 1991, 'Rapid Prototyping in Human-Computer Interface Development', *Interacting with Computers*, vol. 3, no. 1, pp. 51–91.
- Hevner, A. & Chatterjee, S. 2010a, *Design Research in Information Systems*, vol. 22, Springer US, Boston, MA, viewed 7 April 2016, <<http://link.springer.com/10.1007/978-1-4419-5653-8>>.

- Hevner, A. & Chatterjee, S. 2010b, 'Design Science Research in Information Systems', *Design Research in Information Systems*, vol. 22, Springer US, Boston, MA, pp. 9–22, viewed 2 February 2016, <http://link.springer.com/10.1007/978-1-4419-5653-8_2>.
- Hevner, A.R., March, S.T. & Ram, S. 2004, 'Design Science in Information Systems Research', *MIS Quarterly*, vol. 28, no. 1, pp. 75–105.
- Highsmith, J. 2002, 'What is Agile Software Development?', *The Journal of Defense Software Engineering*, vol. 15, no. 10, pp. 4–9.
- Homrighausen, A., Six, H.-W. & Winter, M. 2002, 'Round-trip prototyping based on integrated functional and user interface requirements specifications', *Requirements Engineering*, vol. 7, no. 1, pp. 34–45.
- HTML Responsive Web Design* 2017, viewed 25 July 2017, <https://www.w3schools.com/html/html_responsive.asp>.
- International Institute of Business Analysis 2017, *Becoming a Business Analyst*, viewed 6 October 2017, <<https://www.iiba.org/Careers/Careers/becoming-a-business-analyst.aspx>>.
- Introduction to Ebase Xi* 2017, viewed 18 September 2017, <http://portal.ebasetech.com/cp/doc/Introduction_To_EbaseXi.htm>.
- Jacobson, I., Booch, G. & Rumbaugh, J. 1999, 'The unified process', *The unified software development process*, Addison-Wesley, Reading, Mass.
- JavaFX Developer Home* n.d., viewed 10 January 2016, <<http://www.oracle.com/technetwork/java/javase/overview/javafx-overview-2158620.html>>.
- Johnko 2007, *Advantages & Disadvantages of Webservices*, viewed 25 July 2017, <<https://social.msdn.microsoft.com/Forums/en-US/435f43a9-ee17-4700-8c9d-d9c3ba57b5ef/advantages-disadvantages-of-webservices?forum=asmxandxml>>.
- Kent, S. 2002, 'Model driven engineering', *Integrated formal methods*, Springer, pp. 286–298, viewed 8 December 2015, <http://link.springer.com/chapter/10.1007/3-540-47884-1_16>.
- Ko, A.J., Myers, B., Rosson, M.B., Rothermel, G., Shaw, M., Wiedenbeck, S., Abraham, R., Beckwith, L., Blackwell, A., Burnett, M., Erwig, M., Scaffidi, C., Lawrance, J. & Lieberman, H. 2011, 'The state of the art in end-user software engineering', *ACM Computing Surveys*, vol. 43, no. 3, pp. 1–44.
- Koch, N., Knapp, A., Zhang, G. & Baumeister, H. 2008, 'The Authoring Process of the UML-based web engineering', *Web Engineering: Modelling and Implementing Web Applications*, Springer, pp. 157–191, viewed 15 December 2015, <http://link.springer.com/content/pdf/10.1007/978-1-84628-923-1_7.pdf>.

- Koch, N. & Wirsing, M. 2001, 'Software engineering for adaptive hypermedia applications', *8th International Conference on User Modeling, Sonthofen, Germany*, Citeseer, viewed 15 December 2015, <<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.24.4017&rep=rep1&type=pdf>>.
- Kraut, R.E. & Streeter, L.A. 1995, 'Coordination in software development', *Communications of the ACM*, vol. 38, no. 3, March, pp. 69–81.
- Larman, C., Kruchten, P. & Bittner, K. 2001, 'How to fail with the rational unified process: Seven steps to pain and suffering', *Valtech Technologies & Rational Software*, viewed 2 December 2015, <<http://taika.com/files/fail-with-rational-unified-process.pdf>>.
- Lech, P. 2013, 'Time, Budget, And Functionality?—IT Project Success Criteria Revised', *Information Systems Management*, vol. 30, no. 3, pp. 263–75.
- Lew, P., Olsina, L., Becker, P. & Zhang, L. 2012, 'An integrated strategy to systematically understand and manage quality in use for web applications', *Requirements Engineering*, vol. 17, no. 4, pp. 299–330.
- Liang, X., Marmaridis, I. & Ginige, A. 2007, *Facilitating Agile Model Driven Development and End-User Development for Evolving Web-based Workflow Applications*, IEEE, pp. 231–8, viewed 11 January 2016, <<http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4402096>>.
- March, S.T. & Smith, G.F. 1995, 'Design and natural science research on information technology', *Decision Support Systems*, vol. 15, no. 4, pp. 251–66.
- Marcotte, E. 2010, *Responsive Web Design*, viewed 25 July 2017, <<http://alistapart.com/article/responsive-web-design>>.
- Meliá, S., Gómez, J., Pérez, S. & Díaz, O. 2010, 'Architectural and Technological Variability in Rich Internet Applications', *Internet Computing, IEEE*, vol. 14, no. 3, pp. 24–32.
- Memmel, T. & Reiterer, H. 2008, 'Model-Based and Prototyping-Driven User Interface Specification to Support Collaboration and Creativity.', *J. UCS*, vol. 14, no. 19, pp. 3217–3235.
- Microsoft Silverlight* n.d., viewed 10 January 2016, <<https://www.microsoft.com/silverlight/>>.
- Milosavljevic, G., Filipovic, M., Marsenic, V., Pejakovic, D. & Dejanovic, I. 2013, 'Kroki: A mockup-based tool for participatory development of business applications', *Intelligent Software Methodologies, Tools and Techniques (SoMeT), 2013 IEEE 12th International Conference on*, IEEE, pp. 235–242, viewed 14 January 2016, <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6645668>.

- Molina, H. & Olsina, L. 2008, *Assessing Web Applications Consistently: A Context Information Approach*, IEEE, pp. 224–30, viewed 19 May 2016, <<http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4577886>>.
- Morales-Chaparro, R., Linaje, M., Preciado, J.C. & Sánchez-Figueroa, F. 2007, 'MVC web design patterns and rich internet applications', *Proceedings of the Jornadas de Ingeniería del Software y Bases de Datos*, pp. 39–46.
- Mukasa, K.S. & Kaindl, H. 2008, *An Integration of Requirements and User Interface Specifications*, IEEE, pp. 327–8, viewed 13 January 2016, <<http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4685696>>.
- Myers, B.A. & Rosson, M.B. 1992, 'Survey on user interface programming', *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM, pp. 195–202.
- Nawrocki, J. & Olek, L. 2005, 'UC workbench—a tool for writing use cases and generating mockups', *Extreme Programming and Agile Processes in Software Engineering*, Springer, pp. 230–234, viewed 14 January 2016, <http://link.springer.com/chapter/10.1007/11499053_34>.
- Nielsen, J. & Levy, J. 1994, 'Measuring usability: preference versus performance', *Communications of the ACM*, vol. 37, no. 4, pp. 66–75.
- Noda, T. & Helwig, S. 2005, *Rich Internet Applications -Technical Comparison and Case Studies of AJAX-Flash_Java based RIA*, University of Wisconsin E-Business Consortium, Madison, Wisconsin, viewed 18 December 2015, <<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.119.4482&rep=rep1&type=pdf>>.
- Panach, J.I., España, S., Pederiva, I. & Pastor, Ó. 2008, 'Capturing Interaction Requirements in a Model Transformation Technology Based on MDA.', *J. UCS*, vol. 14, no. 9, pp. 1480–1495.
- Pastor, O., Fons, J. & Pelechano, V. 2003, 'OOWS: A method to develop web applications from web-oriented conceptual models', *International Workshop on Web Oriented Software Technology (IWWOST)*, pp. 65–70, viewed 14 December 2015, <http://ceit.aut.ac.ir/~sa_hashemi/My%20Research/0-Selected%20Papers/2-ECommerce%20Systems/OOWS%20A%20Method%20to%20Develop%20Web%20Applications%20from%20Web-Oriented%20Conceptual%20_%206.pdf>.
- Paul, D., Cadle, J. & Yeates, D. (eds) 2014, *Business Analysis*, 3rd edn, BCS Learning & Development Limited, viewed 26 November 2015, <http://proquestcombo.safaribooksonline.com.ezproxy.uws.edu.au/book/software-engineering-and-development/software-requirements/9781780172774/front-cover/00_frontcover_xhtml>.
- Perrini, F., Russo, A. & Tencati, A. 2007, 'CSR strategies of SMEs and large firms. Evidence from Italy', *Journal of business ethics*, vol. 74, no. 3, pp. 285–300.

- Ramdoyal, R., Cleve, A. & Hainaut, J.-L. 2010, 'Reverse engineering user interfaces for interactive database conceptual analysis', *Advanced Information Systems Engineering*, Springer, pp. 332–347, viewed 14 January 2016, <http://link.springer.com/chapter/10.1007/978-3-642-13094-6_27>.
- Reenskaug, T.M.H. 1979, *The original MVC reports*, viewed 15 December 2015, <<https://www.duo.uio.no/handle/10852/9621>>.
- Richards, D., Marrone, M., Vatanasakdakul, S. & others 2011, 'What does an Information Systems Graduate need to Know? A focus on Business Analysts and their role in sustainability', *22nd Australasian Conference on Information Systems, Sydney, Australia*, viewed 18 November 2015, <<http://aisel.aisnet.org/cgi/viewcontent.cgi?article=1014&context=acis2011>>.
- Richardson, C. 2009, 'ORM in dynamic languages', *Communications of the ACM*, vol. 52, no. 4, pp. 48–55.
- Rivero, J.M., Grigera, J., Rossi, G., Luna, E.R. & Koch, N. 2011, 'Improving Agility in Model-Driven Web Engineering.', *CAiSE Forum*, pp. 163–170, viewed 13 January 2016, <<http://ceur-ws.org/Vol-734/PaperVision05.pdf>>.
- Rivero, J.M., Grigera, J., Rossi, G., Robles Luna, E., Montero, F. & Gaedke, M. 2014, 'Mockup-Driven Development: Providing agile support for Model-Driven Web Engineering', *Information and Software Technology*, vol. 56, no. 6, pp. 670–87.
- Rivero, J.M., Rossi, G., Grigera, J., Burella, J., Luna, E.R. & Gordillo, S. 2010, *From mockups to user interface models: an extensible model driven approach*, Springer, viewed 13 January 2016, <http://link.springer.com/chapter/10.1007/978-3-642-16985-4_2>.
- Rivero, J.M., Rossi, G., Grigera, J., Luna, E.R. & Navarro, A. 2011, 'From interface mockups to web application models', *Web Information System Engineering–WISE 2011*, Springer, pp. 257–264, viewed 13 January 2016, <http://link.springer.com/chapter/10.1007/978-3-642-24434-6_20>.
- Rosson, M.B., Ballin, J. & Rode, J. 2005, 'Who, what, and how: A survey of informal and professional web developers', *2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, IEEE, pp. 199–206, viewed 14 January 2016, <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1509504>.
- Royce, W.W. 1970, 'Managing the development of large software systems', *proceedings of IEEE WESCON*, Los Angeles, pp. 328–388, viewed 27 November 2015, <http://leadinganswers.typepad.com/leading_answers/files/original_waterfall_paper_winston_royce.pdf>.
- Schwabe, D., Rossi, G. & Barbosa, S.D. 1996, 'Systematic hypermedia application design with OOHDM', *Proceedings of the the seventh ACM conference on Hypertext*, ACM, pp. 116–128, viewed 14 December 2015, <<http://dl.acm.org/citation.cfm?id=234840>>.

- Scott, T. 2007, *AJAX Web Model*, viewed 18 December 2015, <<http://derivadow.com/2007/01/05/ajax-what-is-it-its-not-dhtml/>>.
- Shakuntla, R., Sharma, A. & Sarangdevot, S.S. 2013, 'A Study on Modeling Standards for Web Applications and Significance of AspectWebML', *International Journal of Engineering Trends and Technology*, vol. 4, no. 6, pp. 2400–4.
- Sommerville, I. 2007, *Software Engineering*, 8th edn, Addison-Wesley.
- Störrle, H. 2010, *Model Driven Development of User Interface Prototypes: An Integrated Approach*, ACM, Copenhagen, Denmark, pp. 261–8.
- Sumner, M. 1999, 'Critical success factors in enterprise wide information management systems projects', *Proceedings of the 1999 ACM SIGCPR conference on Computer personnel research*, ACM, pp. 297–303, viewed 11 March 2016, <<http://dl.acm.org/citation.cfm?id=299722>>.
- Sweller, J. 1988, 'Cognitive Load During Problem Solving: Effects on Learning', *Cognitive science*, vol. 12, no. 2, pp. 257–285.
- The Standish Group 2009, *Chaos Report*, viewed 18 November 2015, <<http://repository.eafit.edu.co/handle/10784/2771>>.
- Turner, R., Ledwith, A. & Kelly, J. 2010, 'Project management in small to medium-sized enterprises: Matching processes to the nature of the firm', *International Journal of Project Management*, vol. 28, no. 8, pp. 744–55.
- Turner, R., Ledwith, A. & Kelly, J. 2012, 'Project management in small to medium-sized enterprises: Tailoring the practices to the size of company', *Management Decision*, vol. 50, no. 5, pp. 942–57.
- Unified Modeling Language (UML) Resource Page 1997, *Unified Modeling Language (UML) Resource Page*, viewed 12 January 2016, <<http://www.uml.org/>>.
- Urbieta, M., Rossi, G., Ginzburg, J. & Schwabe, D. 2007, 'Designing the interface of rich internet applications', *Web Conference, 2007. LA-WEB 2007. Latin American*, IEEE, pp. 144–153, viewed 8 January 2016, <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4383169>.
- Valderas, P., Pelechano, V. & Pastor, O. 2006, 'A transformational approach to produce web application prototypes from a web requirements model', *International Journal of Web Engineering and Technology*, vol. 3, no. 1, pp. 4–42.
- Valverde, F., Panach, I., Aquino, N. & Pastor, O. 2009, 'Dealing with abstract interaction modeling in an MDE development process: a pattern-based approach', *New Trends on Human–Computer Interaction*, Springer, pp. 119–128, viewed 8 January 2016, <http://link.springer.com/10.1007/978-1-84882-352-5_12>.

- Valverde, F. & Pastor, O. 2009, *Facing the technological challenges of web 2.0: A RIA model-driven engineering approach*, Springer, Berlin / Heidelberg, viewed 11 January 2016, <http://link.springer.com/chapter/10.1007/978-3-642-04409-0_18>.
- Vuorimaa, P., Laine, M., Litvinova, E. & Shestakov, D. 2016, 'Leveraging declarative languages in web application development', *World Wide Web*, vol. 19, no. 4, pp. 519–43.
- Walia, G.S. & Carver, J.C. 2009, 'A systematic literature review to identify and classify software requirement errors', *Information and Software Technology*, vol. 51, no. 7, pp. 1087–109.
- Yang, F., Gupta, N., Botev, C., Churchill, E.F., Levchenko, G. & Shanmugasundaram, J. 2008, 'WYSIWYG development of data driven web applications', *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 163–175.
- Zhang, J. & Chung, J.-Y. 2003, 'Mockup-driven fast-prototyping methodology for Web application development', *Software: Practice and Experience*, vol. 33, no. 13, pp. 1251–1272.

GLOSSARY

"commit inserts" annotation on navigation widget: A mock-up language construct to capture the specification of the end of an insert transaction.

"delete" annotation on navigation widget: A mock-up language construct to capture the specification of a delete operation.

"next" annotation on navigation widget: A mock-up language construct to capture the specification of the display of the next page of data-sets in a search result container.

"previous" annotation on navigation widget: A mock-up language construct to capture the specification of the display of the previous page of data-sets in a search result container.

"search" annotation on navigation widget: A mock-up language construct to capture the specification of a search operation.

"select for insert" annotation on navigation widget: A mock-up language construct to capture the specification of a selection of a data set in a search result container as a part of an insert transaction.

"select for update" annotation on navigation widget: A mock-up language construct to capture the specification of a selection of a data set for an update operation.

"temporarily store for insert" annotation on navigation widget: A mock-up language construct to capture the specification of confirmation an already selected data set to be included as a part of an insert transaction.

"update" annotation on navigation widget: A mock-up language construct to capture the specification of an update operation in an update container.

"WSRequest": A mock-up language construct to capture the specification of a web service invocation.

"=>looked-up widget" notation: A notation used within a look-up widget to specify a search criterion where the looked-up widget refers to a DFYW

"=reference widget" notation: A notation used to specify the source of data to be displayed in a data view widget.

Activity Diagram: An activity diagram refers to an UML construct to model both computational and organizational processes

Agile Software Development Process, Agile Process: "Agile" is an umbrella term for any iterative and incremental software development methodology that uses continuous planning, continuous testing, continuous integration and continuous feedback through the collaborative effort of self-organizing cross-functional teams.

Analyst: A person involved in identifying and specifying the requirements

Asynchronous JavaScript and XML (AJAX): Asynchronous JavaScript and XML (AJAX) is a web application that uses JavaScript as a client-side processing language in the browser and has capabilities to make asynchronous calls to the server to refresh parts of a web page without whole page refresh.

Asynchronous Communication: In RIA context, asynchronous communication means, client-server communication can take place at the same time as the user interacts with a web page

Balsamiq: A popular commercial mock-up editor

Business Analyst (BA): A person involved in the practise of enabling change in an organization by defining the needs and requirements and recommending value-based solutions to stakeholders

Business Transaction: Refers to an interaction between a business and its client and may involve one or more tasks

Calculation Method: A method to calculate values for partial indicators or global indicator

Class diagram: A class diagram is a Unified Modelling Language (UML) construct identifying the structural relationships among important concepts (also known as classes) under consideration in a system

Client: A person or a representative of the person for whom the software needs to be developed

Client-Side Processing: Client-side processing refers to processing on the browser using technologies such as JavaScript

Client-Side Controller (CSC): A Client-side Controller is a client-side control logic unit to manage client-side presentation layer and client-side model layer

Client-Side Model (CSM): A Client-side Model is a logical unit for managing business entities on the client-side.

Client-Side View (CSV): A Client-side View is the logical unit that manages presentation of data and UI on the client-side.

CodeIgniter: CodeIgniter is an open-source software rapid development web framework based on MVC architecture, for use in building dynamic web sites with PHP.

Cognitive load: Cognitive load refers to the extra cognitive effort required for activities not directly related to the problem-solving task

Computation Independent Model (CIM): Computation Independent Model (CIM) depicts the business needs of the application using business user vocabulary and hides the computational details

Concept model with respect to usability measurement: Concept model identifies the high level calculable concepts such as effectiveness in use, efficiency in use, learnability in use and satisfaction in use which are defined in ISO 9126-1 and ISO 25010 standards

Conceptual Model: A conceptual model is a representation of a system in terms of easy to understand concepts and is frequently used to gather and confirm requirements.

Concur Task Tree (CTT): A CTT is a graphical tree structure models that capture end-user interaction tasks as well as the chronological relationships in which user interaction tasks are performed

Container Widget, Layout Widget: A widget that is used to organize and contain other widgets

Contextual-Information Need, Concept model, Attribute, Metric and Indicator (C-INCAMI): C-INCAMI is a framework that defines the concepts and relationships needed to design and implement measurement and evaluation of actual usability of web applications, in a consistent and repeatable way

Create Read Update Delete (CRUD): Create Read Update and Delete are the principal operations in a database system

Data Input Container: A container which can contains data input widgets

Data Model Generator: A component of the auto-generator that helps in finding E-R model of the auto-generated application

Data View Container: A Data View Container is a container that holds one or more Data View Widgets for displaying data where the data to be displayed is available from a search operation or from selected data-set(s)

Data View Selection Container: An abstract data view container with optional actions for “delete” and/or “select for insert” and/or “temporarily store for insert”.

Data View Widget: A widget that is used for displaying structured data

Database Field Yielding Container: A mock-up language term for a data input container that is used to populate a record in a database table.

Database Field Yielding Container (DFYC): A container from which a database table’s schema can be derived

Database Field Yielding Widget (DFYW): A data input widget in a Database Field Yielding Container from which a database table's field can be identified

Database Schema: The organization of data as a blueprint of how the database is constructed

Data-binding: Data-binding is the process that establishes a connection between the application UI (User Interface) and Business logic in the Knockout.js JavaScript library

Design Cycle: The cycle in DSR method during which the design of the artifact is identified along with the heuristics used to build the artifact and the evaluations to be performed for its validation of the artifact

Design Science Research (DSR): A research method where knowledge and understanding of a wicked problem and its solution is gained while designing an artifact and during the application of that artifact

Design Science Research in Information System (DSR in IS): DSR in IS is a DSR method employed to guide IS design and to capture the knowledge created during the design process

Designer: A person involved in specifying platform independent solutions

Dynamic View of UI Model: A dynamic view of UI model abstracts the fundamental behavioural changes to the UI due to user interaction

Effectiveness in use: The degree to which the system is used correctly

Efficiency in use: The degree to which the system is used efficiently

Elementary Indicator: A characteristic that is not dependent on another characteristic to evaluate or estimate a calculable concept

Enterprise Resource Planning (ERP): ERP is a software system that integrates applications that allows an organization to manage the business and automate many back-office functions related to IS

Entity - Relationship (E-R): A relationship among entities typically used in computing for the organization of data within a database or an Information System

Evaluation Indicator: An evaluation indicator is the calculation method and the scale to provide an estimate or evaluation of a calculable concept with respect to defined information needs

Extensible Hyper Text Mark-up Language (XHTML): XHTML stands for Extensible Hyper Text Mark-up Language and is a family of XML mark-up languages that mirror or extend versions of the widely used Hyper Text Mark-up Language (HTML)

Extensible Mark-up Language (XML): XML is a popular textual language for structuring data for communications on the web

For-Each Container: A special type of Data View Container that is used to create a repeated set of data view containers, for pagination effect on the client side.

Functional Testing: Functional testing is a black-box testing process to discover functional failures or defects.

Global Indicator: An attribute whose value is calculated from partial indicator values using a suitable formula

Implementer: A person involved in specifying platform specific solutions

Input Widget: A widget that is used by a user to input data

Insert Business Transaction: A business transaction resulting in creation of one or more database records

Iterative and Incremental Development Process: A software development process where the development is carried out following a prioritized list of increments

Java: A popular cross-platform programming language for building network and web applications

JavaScript: A client-side processing language

JavaScript Object Notation (JSON): JSON is a data format that uses human-readable text to transmit data objects consisting of attribute–value pairs and array data types.

JQuery: JQuery is an open source cross-platform JavaScript library designed to simplify the client-side scripting of HTML

Knockout.js: Knockout.js is a JavaScript library that helps in creating rich, responsive display and editor user interfaces with a clean underlying data model on the client-side

Latency of response: The time taken to service a request

Learnability: Learnability refers to the degree to which the users can learn efficiently and effectively to achieve a task

Look-up widget: A data input widget containing the “=>looked-up widget” notation to specify a search criterion

Measurement metric: A metric is the defined measurement or calculation method and the measurement scale

Meta-Model: A meta-model is a model derived from a category of models by deriving common patterns in them

Model Driven Architecture (MDA): MDA is a software design approach with a set of guidelines for structuring specifications in the form of models and facilitating transformations between different model types using automated tools and services

Model Driven Engineering (MDE): MDE is an engineering approach for the automatic production of software from simplified models rather than detail rich models

Model View Controller - Model Controller (MVC-MC): An architecture with a client side MVC component and a server-side MC component

Model View Controller - Model View Controller (MVC-MVC): An architecture that uses a MVC framework in the browser and a MVC framework on the server-side

Model View Controller (MVC): An architectural pattern for solving a recurring architectural challenge of decoupling the sub-system for user interface components (also known as View) from the sub-system for controlling each business process (also known as Controller) and the sub-system for the business logic component (as known as Model).

Multi-Row Container: Is a special type of Data View Container for presenting repetitive information generally as rows in a HTML table

MVC-MC Component Generator: A component of the auto-generator that helps in deriving the application logic and database logic of the auto-generated application

MySQL: MySQL is a popular open source database management system used by many high-volume, business-critical applications to power large Web sites, enterprise support and packaged software

Navigation Only Container: A Navigation Only Container is a container that does not contain Data View Widgets or Data Input Widgets but contains at least one navigation widget

Navigational Widget: A widget that is used to capture the target from which the UI is to be perceived

Non-functional testing: Non-functional testing refers to testing to evaluate non-functional requirements

Object Oriented Design and Development (OODD): A software development method where a system is designed as components in the form of interacting business objects where an object is an instance of a class and a class is a blueprint that encapsulates the behaviour and properties of a business object

Object Relational Mapping (ORM): ORM is a powerful method for designing and querying relational database models at a conceptual level, where the application is described in terms easily understood by non-technical users.

Partial Indicator: A sub-characteristic value in a concept model

PHP: PHP is a server-side scripting language designed primarily for web development

Physical Transaction, Computational Transaction: A computational task as a part of a business transaction

Platform Independent Model (PIM): PIM is a model that contains no platform specific details to enable its mapping to any desired platform by transformations

Platform Specific Model (PSM): A platform-specific model is a model of a software or business system that is linked to a specific technological platform

Quality-in-use: Quality-in-use refers to the effectiveness, efficiency, and satisfaction that users gain while interacting with the application in a real user environment

Relational Database Management System (RDBMS): RDBMS is a database management system that is based on set theory

Relevance Cycle: The cycle in DSR method during which the design requirements are identified

Report View Container: A special type of data view container optionally used for displaying data in a report format

Rich Internet Application (RIA): A class of web applications that exhibit desktop application like UI and responsiveness

Rigor Cycle: The cycle in DSR method that ensures the design is based on scientific theories and methods to produce a new knowledge base of artifacts is useful to the society

Satisfaction in use: The degree to which users perceive the system as approvable

Search Container: Search Container is a container that contains the widgets required for a search operation

Search Result Container: A Search Result Container is a container that is used to contain the results of a search operation

Server-Side Controller (SSC): Server-side Controller is a part of the application logic to control the operations of the program on the server-side

Server-Side Model (SSM): SSM represents the information (the data) of the application and the business rules required to manipulate the data on the server side

Service Widget: A widget that allows a user to initiate the execution of a domain logic SME is an enterprise that generally has few employees though it could also be high as 250 employees

Small to Medium Enterprise (SME): SME is an enterprise that generally has few employees though it could also be high as 250 employees

Smart Business Object Modelling Language (SBOML): SBOML is a conceptual modelling language that uses succinct, pseudo-English sentences to model business objects and the relations among them

Software Crisis, Software Chaos: A set of problems that highlight the need for changes in existing approaches to software development

Software Requirement Specification (SRS): An SRS is a description of the functional and non-functional requirements of a software system to be developed, where functional requirements specify the behaviour and non-functional requirements specify certain criteria to judge the behaviour

Spiral Software Development Methodology, Spiral model: The spiral model is a risk-driven process model for software development

Static View of UI Model: A static view of UI model abstracts fundamental UI element types among the multitude of UI elements in a web application

Sub-task completeness effectiveness: The degree to which specified users completely execute sub-tasks of a task without regard to correctness

Sub-task completeness efficiency: The proportion of time required for sub-task completeness effectiveness

Sub-task correctness effectiveness: The degree to which specified users correctly execute sub-tasks of a task without regard to completeness

Sub-task correctness efficiency: The proportion of time required for sub-task correctness effectiveness

System Analyst, system analysis: System Analyst is a person who analyses and specifies the IS requirements for further software development

System Usability Scale (SUS): A popular ten-item questionnaire developed by Digital Equipment Co for measuring system usability through a survey

Task successfulness effectiveness: The degree to which specified users correctly complete an entire task. That is, no errors for any sub-task, with all sub-tasks completed

Task successfulness efficiency: The proportion of time required for task successfulness effectiveness

Tester: A person involved in testing the system for functional and non-functional requirements

Transactional Web Application: A transactional web application is one which manages each business process in a series of one or more physical (computational) transactions, in which if one physical transaction fails, the entire process is considered to have failed.

UI Mock-up Tree Data Structure Generator: A component of the auto-generator that helps in performing operations, such as searching a widget, identifying a group of widgets, checking uniqueness of a group of widgets, and finding source and destinations of navigation widgets, during the auto-generation process

UI Mock-up, Wireframe: An UI mock-up is a model of a software that looks like the real UI but is designed to gather information to build the real system

Unified Modelling Language (UML): UML is a general purpose graphical modelling language predominantly used in the field of software engineering for visualizing the requirements and design of a system

Update Container: An Update Container is a container that contains data input widgets which are initialized by a previously selected data-set

Usability Inspector, Usability Tester: A trained tester often used for usability testing instead of real users of the system

Usability of the Auto-generated Application: A measure of the functional correctness, effectiveness in use, efficiency in use and satisfaction in use of the auto-generated application

Usability of the Mock-up Language: A measure of the effectiveness, efficiency and satisfaction of users in using the mock-up language for the creation of a mock-up

Usability Testing: Testing the usability of a system

Use Case: A use case is a list of actions defining the interactions typically between a user and a system to achieve a goal

User: A person who uses the system or software

Validating usability: Validating usability implies validating functional correctness and non-functional features such as effectiveness, efficiency and satisfaction in use

Waterfall Software Development Methodology, Waterfall model: Waterfall model of software development is a document centric process of software development with distinct activity phases such as conception, initiation, analysis, design, construction, testing, deployment and maintenance

Web 1.0, traditional web application: Web applications in which the client-side is mostly powered by HTML code whose principal function is to present the UI for user interaction

Web 2.0: Web applications having client-side processing resulting in better responsiveness and UI features than traditional web applications

Web Application Architecture: A conceptual structure and logical organization of a web application.

Web Application Server: A web application server contains the business logic, front end logic as well as the database logic of the application

Web Service: A web service extends the web infrastructure (such as HTTP, XML, JSON, SOAP/REST) so that one software can utilize the services of another software application without worrying about how the invoked web service is implemented

What You See Is What You Get (WYSIWYG): A modelling approach where stakeholders get to see what the end-result will look like while the interface or document is created

APPENDIX 1 ALGORITHMS FOR GENERATING DATA MODEL FROM MOCK-UP

This section contains the detailed versions of the algorithms in Section 5.1 for generating the data model.

Appendix 1.1 Algorithm to identify a DFYW

This algorithm is the detailed version of the algorithm in Section 5.1.1

```
Boolean isDFYW (widget)
PRE-CONDITION:
widget is NOT NULL and represents either a data input or data view
or navigation widget
POST-CONDITIONS:
The widget is marked as DFYW if it is a DFYW
BEGIN
  IF widget is data input widget type AND
    widget is NOT look-up type AND
    widget is NOT update type
  THEN
    Mark widget as DFYW
    Return true
  ENDIF
  Return false
END
```

Appendix 1.2 Algorithm to identify a DFY Container

The algorithms in this section through to Appendix 1.2.3 represent the detailed version of the algorithm in Section 5.1.2

```
Boolean isDFYContainer(container)
PRE-CONDITION:
container is NOT NULL and may contain a single widget and or other
nested containers.
POST-CONDITIONS:
If the container has at least one DFYW it is marked as
DFY Container. However, if container name starts with "unique"
then it is not marked as DFYContainer
BEGIN
  IF isInferringKeys (container) THEN Return false ENDIF (see
Appendix 1.2.1)
  linkFound = false
  dfywFound = false
  FOR each widget in container
```

```

IF widget is Navigation type AND
  Navigation annotation is "temporarily store for insert" OR
  Navigation annotation is "commit inserts"
THEN
  linkFound = true
ELSE IF isDFYW(widget) is true
THEN
  dfywFound = true
ENDIF
IF dfywFound AND linkFound
THEN
  Mark container as DFY Container
  Return true
ENDIF
ENDFOR
Return false
END

```

Appendix 1.2.1 Algorithm to identify whether a container is a "unique" container

This algorithm identifies whether a container is a "unique" container.

Boolean isInferringKeys(**container**)

```

BEGIN
  IF container name starts with "unique"
  THEN
    Return true
  ENDIF
  Return false
END

```

Appendix 1.2.2 Algorithm to store references to all inner DFY containers in each container

This algorithm finds all inner DFY containers in a given container.

listDFYContainer (**container**, **listOfDFYContainers**)

PRE-CONDITION:

container is NOT NULL and contains at least one widget and or other containers.

listOfDFYContainers is a list of containers which may or may not be empty.

POST-CONDITIONS:

If **container** has a first level "unique" container then the **container** will also have links to DFYWs of the unique container and the unique container is marked void to indicate that it is should not be treated as DFY Container.

If **container** has at least one DFYW it is marked as DFY Container. Any widget in **container** which satisfies the DFYW criteria is also marked as DFYW.

If any nested container has at least one DFYW it is also marked as DFYW Container.

listOfDFYContainers will store a reference to *container* if *container* is a DFYW container.
listOfDFYContainers will also store references to all DFY containers found within *container*.

```

BEGIN
  IF isInferringKeys (container) (see SectionAppendix1.2.1)
  THEN
    Get parent of container
    FOR each widget in container
      Add a link from parent to widget
    ENDFOR
    Mark unique container as void
  ENDIF
  IF isDFYWContainer(container) (see Appendix1.2)
  THEN
    Mark container as DFYW container
    Add container reference to listOfDFYContainers
  ENDIF
  FOR each element in container
    IF element is container
    THEN
      listDFYContainer (element, listOfDFYContainers) (see Appendix1.2.2)
    ENDIF
  ENDFOR
END

```

Appendix 1.2.3 Algorithm to store references to all DFY Containers

This algorithm finds all DFY Containers in the mock-up of a web app.

```

populateDFYContainers (listOfDFYContainers, webApplication)
PRE-CONDITIONS:
  listOfDFYContainers is an empty list of containers
  webApplication is a tree data structure of a web application's
  visual model.
POST-CONDITIONS:
  listOfDFYWContainers will contain reference to all DFY Containers
  in webApplication.
  All the DFYW containers in webApplication are identified.
  All DFYWs in webApplication are also identified.
BEGIN
  FOR each page in webApplication
    listDFYContainer (page, listOfDFYContainers) (see Appendix1.2.2)
  ENDFOR
END

```

Appendix 1.3 Algorithm to find E-Rs from nested DFY Containers

This algorithm is the detailed version of the algorithm in Section 5.1.3.1. The algorithm uses the data-structure represented in Figure A-1 to store each first level nested DFY containers (if any) for a given DFY Container from which the E-R can be established.

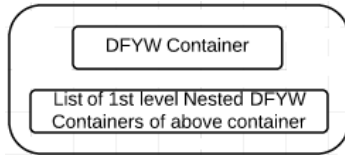


Figure A-1: Data structure to store first level nested DFY Containers in DFY Containers

In the above figure, the first attribute is the DFY Container. It represents a DFY Container which may contain other nested DFY Containers at a depth of first level. The second attribute represents the list of first level nested DFY Containers of the given DFY Container. A collection of objects of the type indicated in Figure A-1 represents all first level nested relationships among the DFY Containers in an application.

populateNestedContainerRelationships(*listOfNestedDFYContainers*,
listOfDFYContainers)

PRE-CONDITIONS:

listOfNestedDFYContainers is an empty array list of an object that can contain references to a DFY Container and its potential nested DFY Containers.

Each DFY Container in the web application has been identified and each DFYW too has been identified.

listOfDFYContainers contains a list of references to all DFY Containers in the *web application*.

POST-CONDITIONS:

Each object in *listOfNestedDFYContainers* will contain a reference to a container and a list of its 1st level nested DFY Containers. That is *listOfNestedDFYContainers* will be a list of objects each of which has a data structure shown in Figure A-1

BEGIN

FOR each *DFYContainer* in *listOfDFYContainers*

 Create an uninitialized *NestedDFYContainer* object (see Figure A-1)

 Add the *DFYContainer* reference to *NestedDFYContainer*

 FOR each *widget* in *DFYContainer*

 IF *widget* is a container and isDFYContainer(*widget*) (see Section Appendix 1.2)

 THEN

 Add the *widget* to the list within *NestedDFYContainer*

 ENDIF

 ENDFOR

 IF the list in *NestedDFYContainer* is not empty

 THEN

 Add *NestedDFYContainer* to *listOfNestedDFYContainers*

 ENDIF

ENDFOR

END

Appendix 1.4 Algorithm to find E-Rs from Search Container

This algorithm is the detailed version of the algorithm in Section 5.1.3.2. It uses the algorithms in Sections Appendix 1.4.1 to Appendix 1.4.3 to accomplish this.

Appendix 1.4.1 Algorithm to find Search Containers

This algorithm finds all Search Containers.

```
populateSearchContainerList (listOfSearchContainers, webApplication)
PRE-CONDITIONS:
  listOfSearchContainers is an empty list of search containers.
  webApplication is a tree data structure of a web application's
  visual model.
POST-CONDITIONS:
  listOfSearchContainers contains reference to all Search Containers
  in webApplication.
  All the Search Containers in webApplication are identified.
BEGIN
  FOR each page in webApplication
    FOR each container in page
      FOR each widget in container
        IF widget is Navigation Type AND widget text is "search"
          THEN
            Mark container as Search Container
            Add container to listOfSearchContainers if not already added
          ENDIF
        ENDFOR
      ENDFOR
    ENDFOR
  ENDFOR
END
```

Appendix 1.4.2 Algorithm to find a DFYW's Container name

This algorithm finds the container name of a DFYW.

```
STRING getDFYContainerName (listOfDFYContainers, fieldname)
PRE-CONDITIONS:
  listOfDFYContainers is a pre-populated list of DFY Containers in
  the web application
  Each DFYW in listOfDFYContainers has a unique name
  fieldname is not null and should refer to one of the DFYW in
  listOfDFYContainers
BEGIN
  FOR each dfyContainer in listOfDFYContainers
    IF fieldname in dfyContainer THEN return name of dfyContainer ENDIF
  ENDFOR
END
```

Appendix 1.4.3 Algorithm to find E-Rs from Search Containers

This algorithm finds all E-Rs from Search Containers.

```
findERFromSearchContainers (listOfDFYContainers,
  listOfSearchContainers, listOfERsFromSearchContainers)
PRE-CONDITIONS:
  listOfDFYContainers is a pre-populated list of DFY Containers in
  the web application (read Appendix 1.2.3 for algorithm to get this
  pre-populated)
  listOfSearchContainers is a pre-populated list of Search Containers
```

in the web application (read Appendix 1.4.1 for algorithm to pre-populate this)

listOfERsFromSearchContainers is an empty list of all E-Rs from Search Containers in the web application

POST-CONDITIONS:

listOfERsFromSearchContainers contains a list of all E-Rs from Search Containers in the web All

BEGIN

FOR each **searchContainer** in **listOfSearchContainers**

Set **listDfyContainerNames** to empty list

FOR each **widget** in **searchContainer**

IF **widget** is a look-up widget THEN

dfyContainerName = getDFYContainerName(**listOfDFYContainers**,
fieldname) (see Appendix 1.4.2)

IF **dfyContainerName** not in **listDfyContainerNames**

THEN

add **dfyContainerName** to **listDfyContainerNames**

ENDIF

ENDIF

ENDFOR

IF count of items in **listDfyContainerNames** is more than one

THEN

Sort **listDfyContainerNames**

Create entity relationship pairs for all permutations of names in

listDfyContainerNames

Add each pair to **listOfERsFromSearchContainers** if not in

listOfERsFromSearchContainers

ENDIF

ENDFOR

END

Appendix 1.5 Algorithm to find E-Rs from “temporarily store for insert” annotations

This section contains the detailed version of the algorithm in Section 5.1.3.3. It uses the helper functions algorithms from Appendix 1.5.1 to Appendix 1.5.7

createERFromTemporarilyStoreForInsertLinks

(**listOfTempStoreForInsertLinks**,
listOfStartingContainersOfTempStoreForInsertLinks,
listOfDFYContainers,
listOfSearchContainers,
listOfSelectForInsertLinks)

PRE-CONDITIONS:

listOfTempStoreForInsertLinks contains references to “temporarily store for insert” links

listOfStartingContainersOfTempStoreForInsertLinks contains references to all containers that are not targets of “temporarily store for insert” nor targets of “select for insert” links.

The source container of each “temporarily store for insert” link is either a Data View Container or a Database Field Yielding Widget Container (See Section 5.1.3.3)

listOfDFYContainers is a pre-populated list of DFYW Containers

listOfSearchContainers contains reference to all Search Containers in the application

POST-CONDITIONS:

The **widget** is marked as “data view widget” type if it is so.

```

BEGIN
FOR each tempStoreInsertlink in listOfTempStoreForInsertLinks
  Set srcCnrOfTempStoreInsertlink to source of tempStoreInsertlink
  IF srcCnrOfTempStoreInsertlink in
    listOfStartingContainersOfTempStoreForInsertLinks
  THEN
    Set listOfSrcDFYCnrNames to an empty list
    Set listOfTargetDFYCnrNames to an empty list
    IF isDataViewContainer(srcCnrOfTempStoreInsertlink) (see Section
      Appendix 1.5.7)
    THEN
      getDFYContainerNamesReferencedByADataViewContainer(listOfDFYCo
        ntainers,srcCnrOfTempStoreInsertlink,listOfSrcDFYCnrNames) (See
        Appendix 1.5.5)
    ELSE IF isDFYContainer(srcCnrOfTempStoreInsertlink) (Appendix 1.2)
    THEN
      Add srcCnrOfTempStoreInsertlink to listOfSrcDFYCnrNames if not
        already added
    ENDIF
    Set targetCnrOfTempStoreInsertlink to target of
      tempStoreInsertlink
    IF isDFYContainer(targetCnrOfTempStoreInsertlink) (Appendix 1.2)
    THEN
      Add targetCnrOfTempStoreInsertlink to listOfTargetDFYCnrNames
        if not already added
    ELSE IF isDataViewContainer(targetCnrOfTempStoreInsertlink) (see
      Appendix 1.5.7)
    THEN
      getDFYContainerNamesReferencedByADataViewContainer (
        listOfDFYContainers,targetCnrOfTempStoreInsertlink,
        listOfTargetDFYCnrNames) (see Appendix 1.5.5)
    ELSE IF targetCnrOfTempStoreInsertlink in listOfSearchContainers
    THEN
      FOR each searchCnr in listOfSearchContainers
        IF targetCnrOfTempStoreInsertlink equals searchCnr
        THEN
          Set selectForInsertTargetCnr from
            getSelectForInsertTargetFromSearchCnr(searchCnr)
              (see Appendix 1.5.4)
          getDFYContainerNamesReferencedByADataViewContainer (
            listOfDFYContainers,selectForInsertTargetCnr,
            listOfTargetDFYCnrNames) (see Appendix 1.5.5)
          ENDIF
        ENDFOR
      ENDIF
      Create E-R from the DFY Container names in listOfSrcDFYCnrNames
        and listOfTargetDFYCnrNames
    ENDIF
  ENDFOR
END

```

Appendix 1.5.1 Algorithm to find all “Temporarily Store for Insert” annotated navigation widgets

This algorithm finds all “temporarily store for insert” annotated navigation widgets.

```

populateTemporarilyStoreForInsertLinksList (

```



```

    listOfTempStoreForInsertLinks, webApplication)
PRE-CONDITIONS:
    listOfTempStoreForInsertLinks is an empty list of "temporarily
store for insert" links.
    webApplication is a tree data structure containing all widgets in a
visual model of a web application. This includes the "temporarily
store for insert links".
    A "temporarily store for insert link" does not exist in nested
containers
POST-CONDITIONS:
    listOfTempStoreForInsertLinks contains reference to "temporarily
store for insert" links from webApplication.
BEGIN
FOR each page in webApplication
FOR each container in page
FOR each widget in container
IF widget is Navigation Type AND widget text is "temporarily
store for insert"
THEN
Add unique widget to listOfTempStoreForInsertLinks
ENDIF
ENDFOR
ENDFOR
ENDFOR
END

```

Appendix 1.5.2 Algorithm to find all "Select for Insert" annotated navigation widgets

This algorithm finds all "select for insert" navigation widgets in the mock-up.

```

populateSelectForInsertLinksList(listOfSelectForInsertLinks,
listOfSearchContainers)
PRE-CONDITIONS:
    listOfSelectForInsertLinks is an empty list
    listOfSearchContainers is a pre-populated list of Search
Containers in the web application (read Appendix 1.4.1 for
algorithm to pre-populate this)
    A "select for insert link" can exist only in the nested
container of Search Result Container
POST-CONDITIONS:
    listOfSelectForInsertLinks contains references to select for
insert links
BEGIN
FOR each searchContainer in listOfSearchContainers
Get innerContainer in searchContainer
FOR each widget in innerContainer
IF widget is Navigation Type AND widget text is "select for
insert"
THEN
Add unique widget to listOfSelectForInsertLinks
ENDIF
ENDFOR
ENDFOR
END

```

Appendix 1.5.3 Algorithm to find containers that are not targets of “select for insert” or “temporarily store for insert” annotated navigation widgets

This algorithm finds all containers that are not targets of “select for insert” or “temporarily store for insert” annotated navigation widgets.

```
getStartingContainersOfTemplyStoreForInsertLinksList(  
    listOfTempStoreForInsertLinks, listOfSelectForInsertLinks,  
    listOfStartingContainersOfTempStoreForInsertLinks)  
PRE-CONDITIONS:  
    listOfTempStoreForInsertLinks contains references to “temporarily  
    store for insert” links (see Appendix 1.5.1)  
    listOfSelectForInsertLinks contains references to “select for  
    insert” links (see Appendix 1.5.2)  
    listOfStartingContainersOfTempStoreForInsertLinks is an empty list  
    Each “temporarily store for insert” and “select for insert” link  
    contains references to its source and target widgets  
    Each “temporarily store for insert link” contains a reference to  
    its immediate parent container  
    Target of “temporarily store for insert link” is always a container  
    Target of “select for insert link” is always a container  
POST-CONDITIONS:  
    listOfStartingContainersOfTempStoreForInsertLinks contains  
    references to all containers that are not targets of “temporarily  
    store for insert” nor targets of “select for insert” links.  
BEGIN  
    FOR each tempStoreLink in listOfTempStoreForInsertLinks  
        Set flag to “not found”  
        Set sourceTempStoreLink from source of tempStoreLink  
        Note: tempStoreLink shouldn't be target of “select for insert”  
        For each selectInsertLink in listOfSelectForInsertLinks  
            Set targetOfSelectInsertLink from target of selectInsertLink  
            IF sourceOfTempStoreLink equals targetOfSelectInsertLink  
            THEN  
                Set flag to “found container” and Break this loop  
            ENDIF  
        ENDFOR  
        Note: tempStoreLink shouldn't be target of “temporarily  
        store for insert”  
        IF flag equals “not found”  
        THEN  
            FOR each tempStoreLink2 in listOfTempStoreForInsertLinks  
                Set targetOfTempStoreLink from target of tempStoreLink2  
                IF sourceOfTempStoreLink equals targetOfTempStoreLink  
                THEN  
                    Set flag to “found container” and Break this loop  
                ENDIF  
            ENDFOR  
        ENDIF  
        IF flag equals “not found”  
        THEN  
            Add unique sourceTempStoreLink to  
            listOfStartingContainersOfTempStoreForInsertLinks  
        ENDIF  
    ENDFOR  
END
```

Appendix 1.5.4 Algorithm to find target of “Select for Insert” annotated navigation widget starting from a Search Container

This algorithm finds the target of a “select for insert” annotated navigation widget in a given search container.

```
getSelectForInsertTargetFromSearchCnr(searchCnr)
```

PRE-CONDITIONS:

searchCnr is a Search Container that has “search” link.
The “search” link points to a Search Result Container.
The Search Result Container has an inner container that has a widget which is the source of “select for insert” link

POST-CONDITIONS:

Returns the target container of the “select for insert” link in the Search Result Container corresponding to this Search Container.

```
BEGIN
```

```
FOR each widget in searchCnr
```

```
IF widget is Navigation Type and text is “search”
```

```
THEN
```

```
Set target to target of “search” link
```

```
Set innerCnr from the inner container in target
```

```
FOR each innerWidget in searchCnr
```

```
IF innerWidget is Navigation type with text “select for insert”
```

```
THEN
```

```
Return target of “select for insert”
```

```
ENDIF
```

```
ENDFOR
```

```
ENDIF
```

```
ENDFOR
```

```
Return not found
```

```
END
```

Appendix 1.5.5 Algorithm to get all DFY Container names referenced in a Data View Container

This algorithm finds all DFY Container names referenced in a Data View Container.

```
getDFYContainerNamesReferencedByADataViewContainer(  
    listOfDFYContainers, dataViewCnr, listOfDFYCnrNames)
```

PRE-CONDITIONS:

listOfDFYContainers is a pre-populated list of DFY Containers.

Each DFYW in *listOfDFYContainers* has a unique name.

dataViewCnr is a Data View Container.

Each data view widget in *dataViewCnr* refers to one of the DFYW in *listOfDFYContainers*.

listOfDFYCnrNames is an empty list.

POST-CONDITIONS:

listOfDFYCnrNames contains a list of DFY Container names referenced by *dataViewCnr*.

```
BEGIN
```

```
FOR each dataViewWidget in dataViewCnr
```

```
Set fieldname to referenced DFYW name in dataViewWidget
```

```
Set dfyCnrName from getDFYContainerName(listOfDFYContainers,  
    fieldname) (see Appendix 1.4.2)
```

```

    Add dfyCnrName to listOfDFYCnrNames if not already added
  ENDFOR
END

```

Appendix 1.5.6 Algorithm to find whether a Widget is a Data View Widget

This algorithm is used as a helper function by the algorithms in Section 5.1.3 while identifying the database relationships among containers.

```

Boolean isDataViewWidget (widget)
PRE-CONDITIONS:
  widget is NOT NULL and represents either a data input or data view
  or navigation widget.
POST-CONDITIONS:
  widget is marked as data view type if it is a data view widget.
BEGIN
  IF widget is data input widget OR widget is navigation widget THEN
    Return false
  ENDIF
  IF widget has "=data source" notational text
  THEN
    Mark widget as data view type
    Return true
  ENDIF
  Return False
END

```

Appendix 1.5.7 Algorithm to check whether a Container is a Data View Container

This algorithm is used as a helper function by the algorithms in Section 5.1.3 while identifying the database relationships among containers.

```

Boolean isDataViewContainer(container)
Pre-condition:
  container has zero or more widgets
Post-condition:
  container will be marked as Data View Container if it is a Data
  View Container.
BEGIN
  FOR each widget in container
    IF isDataViewWidget (widget) (see Appendix 1.5.6)
      THEN
        Mark container as Data View Container
        Return true
      ENDIF
    ENDFOR
  Return false
END

```

APPENDIX 2 ALGORITHMS FOR GENERATING COMPONENTS FOR SEARCH OPERATION

This section contains the detailed version of the algorithms in Section 5.2 for deriving the components required for search operation.

Appendix 2.1 Deriving Server-Side Model algorithms for search operation

This is the detailed version of the algorithm in Section 5.2.1.2 for searching on the server side.

```
getSearchResult(postedData, searchResult)
PRE-CONDITIONS:
  postedData is a list of key value pairs posted from the client
  side, representing the search criteria. Each key is in the form of
  DFYContainername.fieldName and each value is of string type.
  searchResult is an empty string
POST-CONDITIONS:
  searchResult is a JSON string containing search results in the form
  [{"DFYW name11: data type",... "DFYW name1n: data type",...}] [{"Value
  of field11",... "Value of field1n"},... [{"Value of fieldm1",... "Value
  of fieldmn"}], where n is the number of search result fields and m
  is the number of search result sets found matching the search keys
BEGIN
  Set aMapOfTableNameAndFields to an empty list
  getPostedTablesAndFields(postedData, aMapOfTableNameAndFields) (see
  Appendix 2.1.1)
  Create a SQL query string using SELECT clause and a WHERE clause
  for each table entry in aMapOfTableNameAndFields, in the form:
  "Select table1.field1Name,...table1.fieldxName,...
  tabley.field1Name,...tabley.fieldzName where table1.field1Name LIKE
  DATA11 and table1.field2Name LIKE DATA12 and ... tabley.field1Name
  LIKE DATAy1 and ...tabley.fieldzName LIKE DATAyz"
  Set search result as a JSON string in searchResult
END
```

Appendix 2.1.1 Helper function for deriving Server-Side Model algorithms for search operations

This is a helper function that parses the posted details in the form of table based key value pairs.

```
getPostedTablesAndFields(postedData, aMapOfTableNameAndFields)
PRE-CONDITIONS:
  postedData is a list of key value pairs posted from the client
```

side. Each key in the form of "DFY Containername.fieldName" and each value is of string type.

aMapOfTableNameAndFields is an empty list.

POST-CONDITIONS:

aMapOfTableNameAndFields is populated with a map containing each unique DFY Container name as the key with its associated value being a list of key-values corresponding to the table field names and values in **postedData** for the DFYContainer name

BEGIN

Create a list of key-values for each unique DFY Containername in **postedData**

Set a new key in **aMapOfTableNameAndFields** for each unique DFY Container name in **postedData** and link it to the list

END

Appendix 2.2 Deriving Client-Side Model attributes for search result operations

This algorithm initializes client side attributes for managing search results.

```
getClientSideAttributesForSearchResult(allData, pageData,  
                                       nextStartIndex, pageSize)
```

PRE-CONDITIONS:

All the parameters are not initialized

POST-CONDITIONS:

All the parameters are initialized

BEGIN

set **allData** to an empty list where **allData** represents search result response sets.

set **pageData** to an empty list where **pageData** represents paginated data to be displayed in Data View Container within the Search Result Container.

set **nextStartIndex** to zero where **nextStartIndex** represents an index to manage enabling and disabling buttons associated with sources of "previous" and "next" navigation widget in Search Result Container.

set **pageSize** to zero where **pageSize** represents the number of records to be displayed at a time in the Data View Container of the Search Result Container

END

Appendix 2.3 Deriving Client-Side Controller for search and search result

This section represents the finer details of the algorithm specified in Section 5.2.1.4. It also includes from Appendix 2.3.1 to Appendix 2.3.3

```
clientSideSearchControllerGenerator(searchCnr, searchResultCnr,  
                                    searchEventHandlerFunction, previousEventHandlerFunction,  
                                    nextEventHandlerFunction, pre-populateMultivaluedWidget )
```

PRE-CONDITIONS:

searchCnr represents a Search Container in the mock-up.
searchResultCnr represents a Search Result Container in the mock-up
searchEventHandlerFunction is a function variable that is not initialized.
previousEventHandlerFunction is a function variable that is not initialized.
nextEverntHandlerFunction is a function variable that is not initialized.
pre-populateMultivaluedWidget is a function variable that is not initialized.

POST-CONDITIONS:
searchEventHandlerFunction is initialized with a function definition.
previousEventHandlerFunction is initialized with a function definition.
nextEverntHandlerFunction is initialized with a function definition.
pre-populateMultivaluedWidget is initialized with a function definition.

```
BEGIN
  IF searchCnr contains "search" navigation widget THEN
    Delegate searchEventHandlerFunction to "search" click event (see
      Appendix 2.3.1)
  ENDIF
  IF searchResultCnr contains "previous" and "next" navigation
    widgets
  THEN
    (see Appendix 2.3.2)
    Delegate previousEventHandlerFunction to "previous" click event
    Delegate nextEverntHandlerFunction to "next" event click event
  ENDIF
  IF searchCnr contains multi-valued widgets THEN
    Delegate pre-populateMultivaluedWidget on "on-load event" (see
      Appendix 2.3.3)
  ENDIF
END
```

Appendix 2.3.1 Client-Side Controller helper function for search operation

This section discusses the algorithm for managing the search operation on the client side.

```
searchEventHandlerFunction (searchCnr, searchResultCnr, allData,
  pageData, nextStartIndex, pageSize)
```

PRE-CONDITIONS:
searchCnr represents a Search Container in the mock-up.
searchResultCnr represents a Search Result Container in the mock-up
allData, **pageData**, **nextStartIndex**, **pageSize** are defined and initialized as discussed in Appendix 2.2

POST-CONDITIONS:
pageSize is reset with a value from within the [] brackets in **searchResultCnr**.
allData, **pageData**, **nextStartIndex** are reset with values from server response data.

```
BEGIN
```

```

Get widget name and value as key value pairs for each input widget
in searchCnr.
Reset pageSize with a value within the [] brackets in
searchResultCnr (see Section 4.2.2).
Make an AJAX (see Section 2.2.1) call to perform search
From Ajax response initialize search result response model
attributes as follows:
set allData with response data corresponding to query result.
set pageData with a sub-set of a maximum of pageSize count from
allData.
set nextStartIndex to the actual count of record sets in pageData.
IF nextStartIndex is equal to count of allData
THEN
  disable source button of "next" navigation widget in
  searchResultCnr
ELSE
  enable
ENDIF
Disable source of "previous" navigation widget in searchResultCnr
END

```

Appendix 2.3.2 Client-Side Controller helper functions for search result traversals

This section discusses the algorithm for managing the traversal of research results using two helper functions, namely `nextEventHandlerFunction` and `previousEventHandlerFunction`.

```

nextEventHandlerFunction (searchResultCnr, nextStartIndex,
pageData, pageSize, allData)
PRE-CONDITIONS:
searchResultCnr represents a Search Result Container in the mock-up
that contains a "next" navigation widget.
allData, pageData, nextStartIndex, pageSize are initialized as
discussed in Appendix 2.3.1
POST-CONDITIONS:
pageData and nextStartIndex are reset for each click of the source
of the "next" navigation widget and depending on pageSize and data
in allData.
The source button of the "next" navigation widget is either
disabled or enabled
BEGIN
IF nextStartIndex less than allData count
THEN
  set pageData from allData [nextStartIndex] to
  allData [min(nextStartIndex+pageSize, allData count)]
  set nextStartIndex to previous nextStartIndex + pageData count
  IF nextStartIndex is equal to allData count
  THEN
    disable source widget of "next" navigation widget
  ELSE
    Enable
  ENDIF
ENDIF
END

```


previousEventHandlerFunction (*searchResultCnr*, *nextStartIndex*,
pageData, *pageSize*, *allData*)

PRE-CONDITIONS:

searchResultCnr represents a Search Result Container in the mock-up that contains a "previous" navigation widget.

allData, *pageData*, *nextStartIndex*, *pageSize* are initialized as discussed in Appendix 2.3.1

POST-CONDITIONS:

pageData and *nextStartIndex* are reset for each click of the source of the "previous" navigation widget and depending on *pageSize* and data in *allData*.

The source of the "previous" navigation widget is either disabled or enabled.

BEGIN

Reset *nextStartIndex* to max(0, current value of *nextStartIndex* minus count of *pageData*)

Reset *nextStartIndex* to max(0, current value of *nextStartIndex* minus *pageSize*)

Reset *pageData* from *allData* [*nextStartIndex*] to *allData* [*nextStartIndex* + *pageSize*] provided *nextStartIndex* + *pageSize* is less than the count of *allData*

IF *nextStartIndex* is equal to zero

THEN

 disable source widget of "previous" navigation link

ELSE

 enable

ENDIF

Set *nextStartIndex* to current value of *nextStartIndex* + *pageData* count

IF *nextStartIndex* is equal to count of *allData*

THEN

 disable source widget of "next"

ELSE

 enable

ENDIF

END

Appendix 2.3.3 Client-Side Controller helper functions for on-load event operation

This section discusses the algorithm for pre-populating multi-valued widgets.

pre-populateMultivaluedWidget (*searchCnr*, *selector₁Values*, ...
selector_nValues)

PRE-CONDITIONS:

searchCnr represents a Search Container in the mock-up.

selector₁Values, ... *selector_nValues* each are empty lists to contain values for each drop down widget in *searchCnr*. These represent the CSMPreSearch entities in Figure 28.

POST-CONDITIONS:

selector₁Values, ... *selector_nValues* each are populated with values for each drop down widget in *searchCnr*

BEGIN

Set *selectorQueryString* to query string with names of the selector widgets in *searchCnr*

Make an AJAX Call to search using *selectorQueryString*

From AJAX response populate *selector₁Values*, ... *selector_nValues*

END

APPENDIX 3 ALGORITHMS FOR GENERATING COMPONENTS FOR INSERT OPERATIONS

The algorithms in this section correspond to the discussion in Section 5.3 for deriving the components required for insert operations from the mock-up

Appendix 3.1 Algorithm to manage ‘select for insert’ action

This section uses algorithms in Appendix 3.1.1 to Appendix 3.1.3 for the auto-generation of the Model and Controller components for the management of “select for insert” action. No server-side components are involved since select for insert operation is purely a client-side service. The algorithms are discussed with respect to entities involved in the sequence diagram in Figure 33.

Appendix 3.1.1 Algorithm for CSC to manage ‘select for insert’ action

This is the algorithm for the Client-Side Controller for managing “select for insert” actions.

```
manageSelectForInsertAction(searchResultCnr, pageData,  
                           CSMSearchSelections)
```

PRE-CONDITIONS:

searchResultCnr is the mock-up of the search container that has a source widget of “select for insert” event.

pageData is the CSM containing data for display in **searchResultCnr** (see Appendix 2.2 for definition and Appendix 2.3.1 for its initialization)

CSMSearchSelections is the uninitialized Client-Side Model for selections from search result

insertKey_Values is initialized to a key

POST-CONDITIONS:

CSMSearchSelections is initialized with selections from **pageData**

BEGIN

```
initializeCSMSearchSelections(searchResultCnr, pageData,  
                             CSMSearchSelections) (see Appendix 3.1.3)
```

Temporarily store the data in **CSMSearchSelections** on the client side

Navigate to target of “select for insert”

END

Appendix 3.1.2 Defining CSM for storing data on a 'select for insert' action

This algorithm is for the definition of CSM for managing selections from search result. In Figure 33 it refers to the definition of SCMSearchSelections.

```
defineCSMSearchSelections(searchResultCnr, CSMSearchSelections,  
    listOfDFYContainers)  
PRE-CONDITIONS:  
    searchResultCnr is the mock-up of the search container that has a  
        source widget of "select for insert" event  
    CSMSearchSelections is the undefined Client-Side Model for  
        selections from search result  
    listOfDFYContainers is pre-populated with the list of all DFY  
        Containers in the web app (see Appendix 1.2.3)  
POST-CONDITIONS:  
    CSMSearchSelections is defined and initialized  
BEGIN  
    Set selectionsCnr as the Data View Container within searchResultCnr  
    FOR each dataViewWidget in selectionsCnr  
        Add a unique attribute to CSMSearchSelections in the form  
        "DFYContainer name__ DFY attribute name" using  
        listOfDFYContainers to find the DFY Widgets and Container name  
    ENDFOR  
END
```

Appendix 3.1.3 Initializing CSM with user selected data on a 'select for insert' action

This algorithm is for the initialization of CSM for managing search selections in Figure 33.

```
initializeCSMSearchSelections(searchResultCnr, pageData,  
    CSMSearchSelections)  
PRE-CONDITIONS:  
    searchResultCnr is the mock-up of the Search Container that has a  
        source widget of "select for insert" annotated navigation widget  
    searchResultCnr is the search result container and displayed in  
        html table form within a <div>  
    pageData is the CSM containing data for display in searchResultCnr  
        (see Appendix 2.2 for definition and Appendix 2.3.1 for its  
        initialization)  
    CSMSearchSelections is the defined but not initialized Client Side  
        Model for selections from search result (see Appendix 3.1.2 for  
        its definition)  
POST-CONDITIONS:  
    CSMSearchSelections is initialized with selections from pageData  
BEGIN  
    Get the clicked html table row number in searchResultCnr  
    Set CSMSearchSelections to selected data from pageData using the  
        clicked row number. This is available since CSM attributes are  
        data-bound to CSV in Knockout.js  
END
```

Appendix 3.2 Algorithm to manage ‘temporarily store for insert’ and “commit insert” actions

This section addresses “temporarily store for insert” as well as “commit inserts” actions together. This is because generally they occur together in a single business transaction. Furthermore, only the Client-Side Controller for insert operation (that is CSCInsert in Figure 33) is discussed because it contains the main logic of the algorithm.

Appendix 3.2.1 Defining Client-Side Controller for insert operation

CSCInsert is the Client-Side Controller to manage an insert business transaction. Defining CSCInsert involves identifying the control actions required to manage an insert transaction.

```
defineCSCInsert(linkedListCrnsInInsert, CSMInsert, CSVInsert)
```

PRE-CONDITIONS:

linkedListCrnsInInsert is a linked list of containers of the form shown in Figure 35. The list may be in one of the four forms shown in the figure and is created from the mock-up. The source container of “temporarily store for insert” or “commit insert” can either be a DFYW Container or a Data View Container and the source container of “select for insert” is always a Data View Container.

CSMInsert is the Client-Side Model for insert operations.

CSVInsert is the Client-Side View for insert operations.

POST-CONDITIONS:

Temporarily stored data on the client side is deleted on successful completion of an insert business transaction

BEGIN

IF **linkedListCrnsInInsert** is of type 1 in Figure 35

THEN

Get user entered data from the DFY Container hosting the source of “commit insert” link using **CSVInsert**

Store the data in temporary storage using **CSMInsert**

ELSEIF **linkedListCrnsInInsert** is of type 2 in Figure 35

THEN

Request **CSMInsert** to temporarily store the selected data on the client side as a part of an insert transaction

IF target of “select for insert” in **linkedListCrnsInInsert** is a Data View Container (DVC)

THEN

Request **CSCInsert** to update temporary storage on the client side with data in the DVC

ELSEIF target is DFY Container

THEN

Get user entered data from the DFY Container hosting the source of “commit insert” link using **CSVInsert**

Update the data in temporary storage with user entered data

Using **SMInsert**

ENDIF

ENDIF

ELSEIF **linkedListCrnsInInsert** is of type 3 in Figure 35

Request **CSMInsert** to temporarily store the selected data on the client side as a part of an insert transaction

FOR each “temporarily store for inserts” source container in

```

LinkedListCrnsInInsert
IF source of "temporarily store for insert" in
    LinkedListCrnsInInsert is a Data View Container
THEN
    Request CSCInsert to update temporary storage on the client side
    with data in the Data View Container
ELSEIF target is DFY Container
THEN
    Get user entered data from the DFY Container hosting the source
    of "temporarily store for insert" link using CSVInsert
    Update the data in temporary storage with user entered data
    using CSMInsert
ENDIF
ENDFOR
ENDIF
ELSEIF LinkedListCrnsInInsert is of type 4 in Figure 35
FOR each "temporarily store for inserts" source container in
    LinkedListCrnsInInsert
IF source of "temporarily store for insert" in
    LinkedListCrnsInInsert is a Data View Container (DVC)
THEN
    Request CSCInsert to update temporary storage on the client
    side with data in the DVC
ELSEIF target is DFYW Container
THEN
    Get user entered data from the DFY Container hosting the source
    of "temporarily store for insert" link using CSVInsert
    Update the data in temporary storage with user entered data
    using CSMInsert
ENDIF
ENDFOR
ENDIF
    In a AJAX call send the temporary data to the server side CSCInsert
    for committing to the database in an insert business transaction
IF server response is "successful"
THEN
    Delete temporarily stored data
ENDIF
END

```

APPENDIX 4 ALGORITHM FOR GENERATING COMPONENTS FOR REPORT MANAGEMENT

This section contains the finer details of the algorithm discussed in Section 5.4 for deriving the components required for report generation, from the mock-up.

Appendix 4.1 Algorithm for defining Client-Side Model for report generation

This section discusses how a client-side model is defined and assigned values.

```
defineCSMsForReportView(reportViewContainer, clientSideStoredData)
PRE-CONDITIONS:
  reportViewContainer is a report view container having at least one
  Label widget containing text notation of the form "=Container Name"
  clientSideStoredData is a map of stored data in the client side in
  the form of a map of "container name" to a list of "widget name -
  data" pairs
POST-CONDITIONS:
  A CSM is defined for each container referenced in
  reportViewContainer
BEGIN
  FOR each key in clientSideStoredData
    FOR each container name (say referenceCnrName) in "=Container
    Name" in reportViewContainer
      IF key equals reportReferenceCnrName
        THEN
          Set CSM to a new CSM
          FOR each pair of items reportReferenceCnrName [key]
            Define a widget name in CSM
            Initialize the widget in CSM
          ENDFOR
          Add CSM to a listOfCSMs
        ENDIF
      ENDFOR
    ENDFOR
  Return listOfCSMs
END
```

Appendix 4.2 Algorithm for defining a Client-Side View for Report generation

This section discusses how a client-side view is generated from the mock-up.

```
defineCSVsForReportView(reportViewContainer, listOfDFYContainers,
listOfDataViewContainers, listOfCSMsForReporting)
```

PRE-CONDITIONS:

reportViewContainer is a report view container having at least one Label widget containing text notation of the form "=Container Name" where "Container Name" refers to either a DFY Container or Data View Container.

listOfDFYContainers is list of all DFY Containers in the mock-up as discussed in Appendix 1.2.3

listOfDataViewContainers is the list of Data View Containers in the mock-up

Each DFY Container and Data View Container has a unique name in the mock-up

listOfCSMsForReporting is the list of client-side models required for **reportViewContainer** for reporting (see Appendix 4.1)

POST-CONDITIONS:

A CSV is defined for each container referenced in **reportViewContainer**

BEGIN

FOR each container name (say **referenceCnrName**) in "=Container Name" in **reportViewContainer**

Set CSV to a new CSV

IF **referenceCnrName** exists in *listOfDFYContainers*

THEN

Set **dfyCnr** from *listOfDFYContainers* for matched **referenceCnrName**

In CSV define an outer <DIV> and an inner <DIV> for the **dfyCnr**

utilizing *listOfCSMsForReporting* for data-binding

ELSE IF **referenceCnrName** exists in *listOfDataViewContainers*

THEN

Set dvCnr from *listOfDataViewContainers* for matched

referenceCnrName

Set *listDFYCnrs* to list of DFY Containers referenced by dvCnr

FOR each DFY Container (say **dfyCnr**) in *listDFYCnrs*

In CSV define an outer <DIV> and an inner <DIV> for the **dfyCnr**

utilizing *listOfCSMsForReporting* for data-binding

ENDFOR

ENDIF

ENDFOR

END

Appendix 4.3 Algorithm for defining a Client-Side Controller for Report generation

This algorithm defines the Client Side Controller for managing report generation.

```
defineCSCForReportView (reportViewContainer, clientSideStoredData,
listOfDFYContainers, listOfDataViewContainers)
```

PRE-CONDITIONS:

reportViewContainer is a Report View Container having at least one Label widget containing text notation of the form "=Container Name".

clientSideStoredData is a map of stored data in the client side in the form of a map of "container name" to a list of "widget name - data" pairs

listOfDFYContainers is list of all DFY Containers in the mock-up as discussed in Appendix 1.2.3

listOfDataViewContainers is the list of Data View Containers in the mock-up

POST-CONDITIONS

A Client-Side Controller is defined for **reportViewContainer**

BEGIN

Set **listOfCSMsForReporting** from `defineCSMsForReportView`
(**reportViewContainer**, **clientSideStoredData**) (see Appendix
4.1)

`defineCSVsForReportView` (**reportViewContainer**, **listOfDFYContainers**,
listOfDataViewContainers, **listOfCSMsForReporting**) (see Appendix
4.2)

END

APPENDIX 5 ALGORITHM FOR GENERATING COMPONENTS FOR UPDATE OPERATION

This section contains the detailed versions of the algorithms in Section 5.5 for deriving the components from the mock-up for update operations.

Appendix 5.1 Algorithm for defining Client-Side Controller for Update

This defines how the client-side controller for update is defined.

```
defineCSCForUpdate(clientSideModelAttributeList,updateContainer)
PRE-CONDITIONS:
  updateContainer contains the mock-up of the Update Container. It is
  used to define CSVUpdate in Figure 40.
  clientSideModelAttributeList is a map of "update container name" to
  a list of names of the form "container name__widget name".
  clientSideModelAttributeList is an attribute of "CSMUpdate" in
  Figure 40.
  The clientSideModelAttributeList has attributes for each data input
  widget in updateContainer.
POST-CONDITIONS
  The stored data is updated both on the server and client side
BEGIN
  defineCSVUpdate(updateContainer) See Appendix 5.2.
  defineCSMForUpdate(clientSideDataForUpdate,updateContainer) See
  Appendix 5.3
  initializeCSMUpdate(clientSideModelAttributeList) See Appendix 5.4
  Collect data from the updated form
  Send updated details to server side for storage
  On successful response from server
  Store updated values on the client side
  Navigate to the target page of "update" action
END
```

Appendix 5.2 Algorithm for defining Client-Side View for Update

This defines the attributes required for client-side view for update.

```
defineCSVUpdate(updateContainer)
PRE-CONDITIONS:
  updateContainer contains the mock-up of the Update Container.
POST-CONDITIONS
  Client-Side View for Update is defined (see Figure 42)
BEGIN
  Create a <div> for containing update widgets
  Create a <FORM> for data entry for update
  Define a view template as follows :
```

```

FOR each data input widget in updateContainer
  Set widgetName to name of the widget
  Set containerName to name of the container in which the data input
  widget exists
  IF containerName starts with "unique"
  THEN
    set containerName to its parent container name
  ENDIF
  Create a widget with id and name of the form
  containerName__widgetName
ENDFOR
END

```

Appendix 5.3 Algorithm for defining Client-Side Model for Update

This defines the attributes required for client-side model for update.

```

defineCSMForUpdate(clientSideModelAttributeList, updateContainer)
PRE-CONDITIONS:
  updateContainer contains the mock-up of the update container. It is
  used to define CSVUpdate in Figure 40.
  clientSideModelAttributeList is not defined.
POST-CONDITIONS
  clientSideModelAttributeList is a map of "update container name" to
  a list of names of the form "container name__widget name".
  clientSideModelAttributeList is an attribute of "CSMUpdate" in
  Figure 40.
BEGIN
  FOR each data input widget in updateContainer
    Set widgetName to name of the widget
    Set containerName to name of the container in which the data input
    widget exists
    IF containerName starts with "unique"
    THEN
      Set containerName to its parent container name
    ENDIF
    Define an attribute containerName__widgetName in
    clientSideDataForUpdate
  ENDFOR
END

```

Appendix 5.4 Algorithm for Initializing Client-Side Model for Update

This initializes the attributes of the client-side model for update.

```

initializeCSMUpdate(clientSideModelAttributeList)
PRE-CONDITIONS:
  clientSideModelAttributeList is a map of "update container name" to
  a list of names of the form "container name__widget name".
  clientSideModelAttributeList is an attribute of "CSMUpdate" in
  Figure 40.
  The client-side storage has data corresponding to each attribute in
  clientSideModelAttributeList.

```

POST-CONDITIONS:

Each attribute in **clientSideModelAttributeList** is initialized from client-side storage

BEGIN

FOR each item in local storage of client for this page.

 initialize corresponding **attribute in clientSideModelAttributeList**

ENDFOR

END

APPENDIX 6 USABILITY TESTING DETAILS

This section contains the biographical details of the usability testers and the details of the case studies for usability testing. Three case studies are discussed in three separate sections, followed by the visual modelling tasks for each case study in another set of three sections. Finally, the last three sections specify end user tasks for usability testing of each of the auto-generated applications corresponding to the three case studies.

Appendix 6.1 Biographical details of the usability testers

Table A-1: Biographical Details of Testers

Tester	Gender	Qualification	Experience	Case Study
1	Male	Degree in Computer Science	6 years as a web application developer	Question and Answer System
2	Male	4 th year student in Biomedical Engineering	6 months as a novice programmer	
3	Male	Degree in IT and Computer Science	1.5 years as a Business Analyst and Developer	
4	Male	Master's degree in IS and in Accounting	20 years as a lecturer in Accounting, Accounting Information Systems but no practical developmental knowledge	Student-Teacher Consultation System
5	Male	Degree in Physics, Graduate Diploma in IT	Novice	
6	Female	PhD in IS	4 years as BA, 20 years as Software Developer	
7	Male	Master's degree in IT, Doing PhD in Digital Eco Systems	4 years as a BA researcher and 2 years as a developer	Patient-Dietician Consultation System
8	Male	Master's degree in IT and in Education	Over 30 years as a lecturer and researcher in IS	
9	Male	Masters in CS	15 years as developer, BA and System Analyst	

Appendix 6.2 Usability Testing Case Studies

This section introduces three case studies specified by three BAs for usability testing of the mock-up language and the auto-generating tool. It also defines the visual modelling tasks and end user tasks for each of the case studies, for usability testing.

Appendix 6.2.1 Testing Case study a- Question Answer System

A Question-Answer System is required to permit questions to be answered by assigned experts. Users can create questions and search for questions. Each question should have information such as the text of the question, the text of the answer which may be blank, the date the question was asked, the date the question was answered, status indicating whether the question has been answered or not and a reference to an expert the question is assigned. The system should be able to create experts. Users should be able to search questions. The search results should display details such as the question text, the answer and the dates the questions were asked and answered. An expert once logged in may search for questions, answer a question, delete a question or assign another expert to answer questions. Figure A-2 illustrates the use cases in the Question Answer System. Note that specific actors are not shown in the use case diagram since user access control is not considered in the visual mock-up since it is outside the scope of this thesis. Please refer to my journal paper written in collaboration with other researchers on the management of access control through mock-ups (Caruccio et al. 2015)

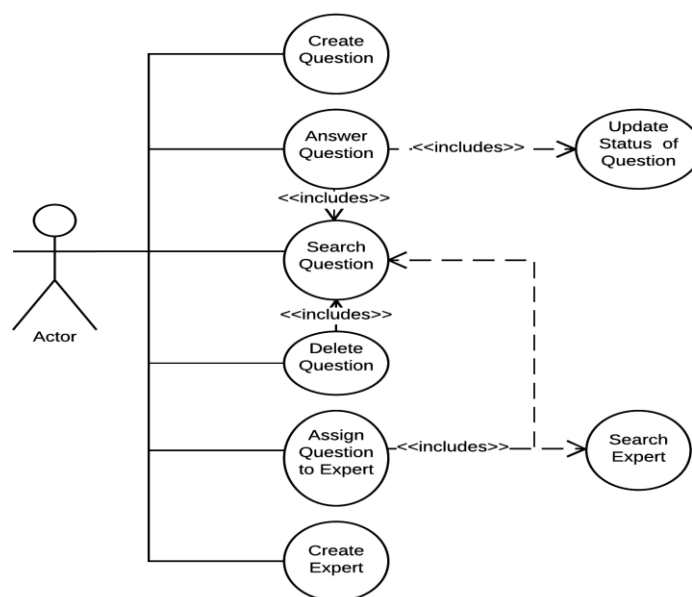


Figure A-2: Use Cases in Question Answer System

Appendix 6.2.2 Testing Case study b- Student-Teacher Consultation System

A Student Teacher Consultation System should permit students to create consultation appointments with teachers. The system should allow creation of student and teacher consultation entities independently of each other. Assume a teacher consultation entity has information such as teacher name, consultation date, time, duration and status as *available*, *not available* or *allocated*. The system should be able to search for a student and teacher consultation record

with **status** as *available* and allocate to a Student. Once this is done the **status** of the Teacher Consultation record assigned to a Student should be changed to *allocated*. The system should be able to delete a Teacher Consultation record where status is “available”. In addition, the system should be able to view all Teacher Consultation records where status is *allocated*. The corresponding use case diagram is shown in Figure A-3.

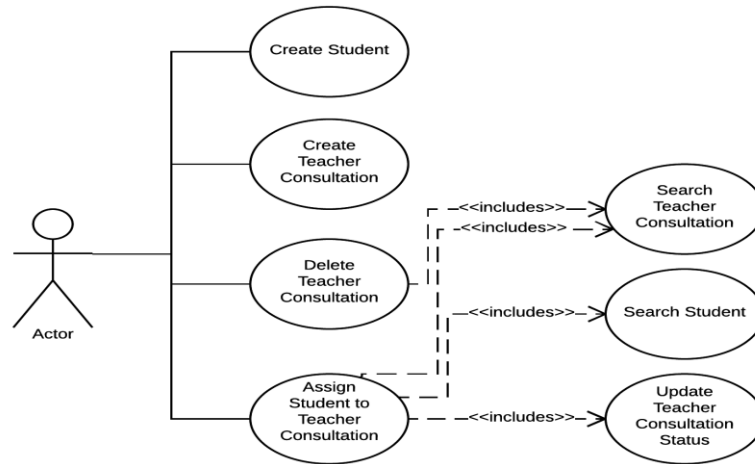


Figure A-3: Use Cases in Teacher Consultation System

Appendix 6.2.3 Testing Case study c- Patient-Dietician Consultation System

A Patient-Dietician Consultation System is required to manage dietician's consultations with patients. A patient has a name and government medical number which does not change. Whenever a patient meets dietician additional details such as height and or weight may be recorded. The system should be able to store a patient detail and as well as dietician's details independently of each other. The system should also be search and assign any patient to any dietician. Figure A-4 illustrates the corresponding use case diagram of the system.

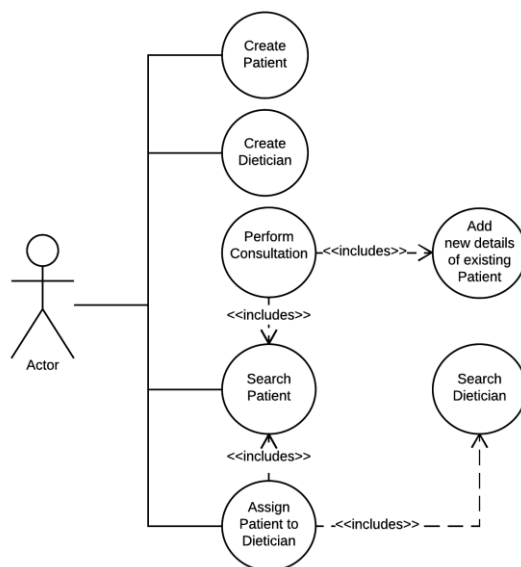


Figure A-4: Uses Cases in the Patient Dietician Consultation System

Appendix 6.2.4 Visual Modelling Tasks in the Question and Answer System

This section discusses the main visual modelling tasks in the Question and Answer System. Essentially it requires two DFY Containers for the creation of a Question and for an Expert. The tasks involved in the creation of these DFY Containers are shown in Table A-2 and Table A-3 respectively. It also requires a Search Container to manage Expert login action (see Table A-4). In addition, it requires tasks for managing a Search Container and a Search Result Container to search and display the various Questions, the details of which are provided in Table A-5. The other use case of this system is to search a Question with an intention of assigning it to an Expert or for answering by an Expert or for its deletion. The tasks involved in this activity are covered in Table A-9. Finally, any visual model of a web application would require at least one Navigation Only Container to manage high-level navigation at least from the main (index) page. Table A-15 captures the generic activities to model a Navigation Only Container. A complete mock-up of this system created by one of the testers is illustrated in Figure A-5. The corresponding screen-shots of the auto-generated instance of the system is provided in Figure 53 .

Table A-2: Actions in the “Creation of a Question entity” task

<p>Task: Creation of a Question. See mock-up of “Create Question Page” in Figure A-5, for reference.</p> <p>The following task action specifications are required:</p> <ul style="list-style-type: none"> A Page for creation of Question A unique page name for creation of Question A DFY Container for containing widgets required for creation of a Question entity A unique name for the DFY Container A unique label within the DFY Container for question A unique label within the DFY Container for answer A unique label within the DFY Container for status of question – whether answered or not A data input field within the DFY Container for question text A data input field within the DFY Container for answer A data input field within the DFY Container for status Button within the DFY Container to trigger insertion operation A "commit inserts" annotated navigation widget with the above-mentioned button as its source
--

Table A-3: Actions in the “Creation of an Expert entity” task

<p>Task: Creation of an Expert. See mock-up of “Create Expert Page” in Figure A-5, for reference.</p> <p>The following task action specifications are required:</p> <ul style="list-style-type: none"> A page for creation of an Expert A unique page name for creation of Expert A DFY Container for containing widgets required for creation of an Expert entity A unique name for the DFY Container A unique label within the DFY Container for expert first name of Expert A unique label within the DFY Container for expert last name A unique label within the DFY Container for expert user name A unique label within the DFY Container for expert password A data input field within the DFY Container for expert first name A data input field within the DFY Container for expert last name A data input field within the DFY Container for expert user name A data input field within the DFY Container for expert password Button within the DFY Container to trigger insertion of the newly created Expert details in the database A "commit inserts" annotated navigation widget with the above-mentioned button as its source
--

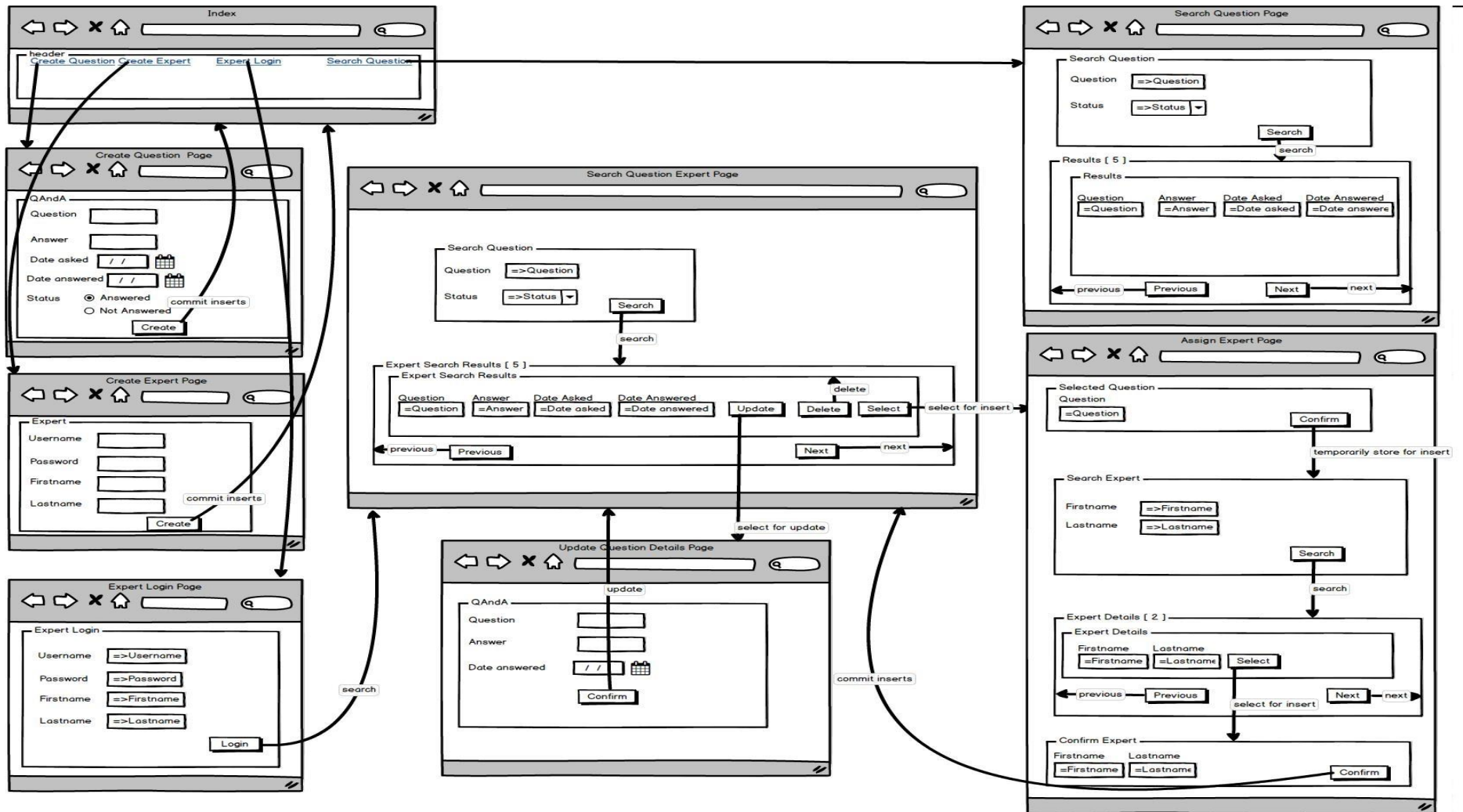


Figure A-5: A mock-up of the Question and Answer System

Table A-4: Actions in the “Expert Login” task

Task: Expert Login. See mock-up of “Expert Login Page” in Figure A-5, for reference.
The following task action specifications are required: A page for expert login A unique name for the page A container within the page The container has a name The container has label for expert user name The container has input widget with “=>Looked-up Widget” notation to search user name The container has a label for expert password The container has input widget with “=>Looked-up Widget” notation to search password The container has a button to trigger a search A "search" annotated navigation widget with the above-mentioned button as its source Target of the "search" annotated navigation widget is a page or a container

Table A-5: Sub-tasks for the “searching questions and displaying them” task

Task: Search and display Questions. See mock-up of “Search Question Page” in Figure A-5, for reference
Sub-task 1: Use Search Container to search Questions. See the sub-task actions specified in Table A-6. Use the mock-up of the “Search Question” container in “Search Question Page” in Figure A-5, for reference.
Sub-task 2: Use Search Result Container to manage display of searched Questions. See Table A-7 for the sub-task actions. Use the mock-up of the “Results[5]” container in “Search Question Page” in Figure A-5, for reference.
Sub-task 3: Use Data View Container within Search Result Container to display Questions. See Table A-8 for sub-task actions. Use the mock-up of the “Results” container within the “Results[5]” container in “Search Question Page” in Figure A-5, for reference.

Table A-6: Sub-task actions for searching a Question

Task: Use Search Container to search Questions. Use the mock-up of the “Search Question” container in “Search Question Page” in Figure A-5, for reference.
The following task action specifications are required: A page for Searching and Viewing question The page has a unique page name A Search Container within the page A name for the Search Container A label for each search criterion in Search Container A “=>Looked-up Widget” notation for search input criterion in Search Container An appropriate reference in “=>Looked-up Widget” notation in search input criterion in Search Container No usage of input widget without “=>Looked-up Widget” notation in Search Container A button for search in Search Container A "search" annotated navigation widget with the above-mentioned button as its source Target of "search" annotated navigation widget is a Search Result Container

Table A-7: Sub-task actions for managing results following the search of a Question

Task: Use Search Result Container for display of searched Questions. Use the mock-up of the “Results[5]” container in “Search Question Page” in Figure A-5, for reference.
The following task action specifications are required: A Search Result Container in the same page as the Search Container page A name for Search Result Container with the name containing a pagination value A button within Search Result Container for navigating to previous set of search result records A "previous" annotated navigation widget with the above-mentioned button as its source A button within Search Result Container for navigating to next set of search result records A "next" annotated navigation widget with the above-mentioned button as its source

Table A-8: Sub-task actions for the display Question(s) within a Search Result Container

Task: Use Data View Container within Search Result Container to display Questions. Use the mock-up of the “Results” container within the “Results[5]” container in “Search Question Page” in Figure A-5, for reference.
The following task action specifications are required: A nested Data View Container exists within the previously described Search Result Container A name for the nested container In nested container, use at least one label for each entity in search criteria In nested container, use at least one “=reference widget” notation for each entity in search criteria In nested container, no usage of input widgets without “=reference widget” notation

Table A-9: Tasks for answering or deleting a Question or for its assignment to an Expert

<p>Task: Search Question to answer it or to link with an Expert or to delete it.</p>
<p>Sub-task 1: Use Search Container to search a Question. This sub-task actions are same as those discussed in Table A-6. See the mock-up of the “Search Question” container in the “Search Question Expert Page” in Figure A-5, for reference.</p>
<p>Sub-task 2: Use Search Result Container to manage display of searched Question. Sub-task actions are same as in Table A-7. See the mock-up of the “Expert Search Results[5]” container in the “Search Question Expert Page” in Figure A-5, for reference.</p>
<p>Sub-task 3: Use of Data View Container within Search Result Container to display Questions with an intention of assigning to an Expert or for its update or its deletion. See Table A-10 for sub-task actions. See the mock-up of the “Expert Search Results” container within the “Expert Search Results[5]” container in the “Search Question Expert Page” in Figure A-5, for reference.</p>
<p>Sub-task 4: Use Update Container to update the status of a Question to indicate it has been answered. See Table A-11 for sub-task actions. See the mock-up of the “Update Question Details Page” in Figure A-5, for reference.</p>
<p>Sub-task 5: Use a Data View Container to display a Question with the aim of linking with an Expert. See Table A-12 for sub-task actions. See the mock-up of the “Selected Question” container within the “Assign Expert Page” in Figure A-5, for reference.</p>
<p>Sub-task 6: Use Search Container to search an Expert to assign a previously selected Question. The sub-task actions are same as in Table A-6 but the search criteria are with respect to an Expert. See the mock-up of the “Search Expert” container within the “Assign Expert Page” in Figure A-5, for reference.</p>
<p>Sub-task 7: Use Search Result Container to manage search results pertaining to an Expert. The sub-task actions are same as in Table A-7. but for managing display of Experts instead of a Question. See the mock-up of the “Expert Details[2]” container within the “Assign Expert Page” in Figure A-5, for reference.</p>
<p>Sub-task 8: Use of Data View Container within Search Result Container to display an Expert for potential linkage with a Question. See Table A-13 for sub-task actions. See the mock-up of the nested “Expert Details” container within the “Expert Details[2]” container in the “Assign Expert Page” in Figure A-5, for reference.</p>
<p>Sub-task 9: Use a Data View Container to display selected Expert to link with a previously selected Question. See Table A-14 for sub-task actions. See the mock-up of the “Confirm Expert” container in the “Assign Expert Page” in Figure A-5, for reference.</p>

Table A-10: Actions for managing the task of “assignment of a Question to an Expert” or for “updating” or “deleting a Question”

<p>Task: Use of Data View Container within Search Result Container to display the Question with an intention of assigning to an Expert or for its update or its deletion. See the mock-up of the “Expert Search Results” container within the “Expert Search Results[5]” container in the “Search Question Expert Page” in Figure A-5, for reference.</p>
<p>The following task action specifications are required:</p> <p>A nested container exists within a Search Result Container to display questions</p> <p>The nested container has a name</p> <p>In nested container, use at least one Label for each entity in search criteria</p> <p>In nested container, use at least one “=reference widget” notation for each entity in search criteria</p> <p>In nested container, no usage of input widgets without “=reference widget” notation</p> <p>In nested container, use a button for selection for update</p> <p>In nested container, use "select for update" annotated navigation widget with the above-mentioned button as its source</p> <p>In nested container, the target of "select for update" annotated navigation widget is a page. In Figure A-5, the target page is “Update Question Details Page”.</p> <p>In nested container, use a button for deletion of a question</p> <p>In nested c container, use a "delete" annotated navigation widget with the above-mentioned button as its source</p> <p>The target of "delete" annotated navigation widget is the nested container. In Figure A-5, the target container name is “Expert Search Results”.</p> <p>In nested container, use a button for selection for insert</p> <p>In nested container, use "select for insert" annotated navigation widget with the above-mentioned button as its source</p> <p>The target of "select for insert" is either a page or a container. In Figure A-5, the target is a page named “Assign Expert Page”.</p>

Table A-11: Actions in the “update of a Question” task

Task: Use of an Update Container to update a Question to indicate it has been answered. See the mock-up of the “Update Question Details Page” in Figure A-5, for reference.

The following task action specifications are required:

- An Update Container for updating a question
- Name of Update Container is same as name of a corresponding DFY Container for Question
- One or more labels within Update Container for question
- Labels within update question container has same name as corresponding label in create question DFY Container
- A data input widget next to each label within Update Container
- The data input widget does not contain “=>Looked-up Widget” notation
- A button within Update Container to trigger storage of updates made
- An "update" annotated navigation widget with the above-mentioned button as its source
- The target of "update" annotated navigation widget is a page. In Figure A-5, the target is the “Search Question Expert Page”.

Table A-12: Actions in the task of managing assignment of a displayed Question to an Expert

Task: Use a Data View Container to display Question to be assigned to an Expert. See the mock-up of the “Selected Question” container within the “Assign Expert Page” in Figure A-5, for reference.

The following task action specifications are required:

- Creation of a Data View Container to display selected Question
- The Data View Container has a name
- At least one label within the Data View Container
- A “=reference widget” notation in at least one label
- The reference in the “=reference widget” notation refers to a label name in the create question DFY Container
- A button within Data View Container to allow a user to select the question
- A "temporarily store for insert" annotated navigation widget with the above-mentioned button as its source

Table A-13: Actions in the task of selecting an Expert from Search Result Container for potential linkage with a previously selected Question

Task: Use of Data View Container within Search Result Container to enable selection of an Expert. See the mock-up of the nested “Expert Details” container within the “Expert Details[2]” container in the “Assign Expert Page” in Figure A-5, for reference.

The following task action specifications are required:

- A nested container within a Search Result Container for displaying expert entities
- A name for the nested container
- In nested container, the existence of at least one label for each entity in corresponding search criteria
- In nested container, the existence of at least one “=reference widget” notation for each entity in search criteria
- No usage of data input widgets within nested container
- In nested container, the existence of a button for selection for a search result data set for potential insert transaction
- A "select for insert" annotated navigation widget with the above-mentioned button as its source
- The target of "select for insert" is either a page or a container

Table A-14: Actions for displaying a selected Expert for linkage with a previously selected Question

Task: Use of a Data View Container to display selected Expert for linkage with a pre-selected Question. See the mock-up of the “Confirm Expert” container in the “Assign Expert Page” in Figure A-5, for reference.

The following task action specifications are required:

- A Data View Container to display selected expert
- A name for the Data View Container
- Existence of at least one Label within Data View Container
- Existence of at least one “=reference widget” notation within Data View Container
- The reference in the “=reference widget” notation refers to a label name in a corresponding DFY Container for expert creation
- A button within the Data View Container to confirm a selected data item for further operations in an insert transaction
- A "commit inserts" annotated navigation widget with the above-mentioned button as its source

Table A-15: Actions for creating a navigation only container as a header

Task: Use of main header for the app. See the mock-up of the “Header” container in the “Index” page in Figure A-5
The following task action specifications are required: A Navigation Only Container as a header of the home page of the application A button or a link to trigger navigation to each use case of the application A navigation widget with the above-mentioned button or link as its source Target of the navigation widget is either a page or a container

Appendix 6.2.5 Visual Modelling Tasks in the Teacher Consultation System

This section discusses the main visual modelling tasks in the Teacher Consultation System. Essentially it requires two DFY Containers for the creation of a Student and for a Teacher Consultation (TC) entity. The tasks involved in the creation of these DFY Containers are shown in Table A-19 and Table A-20 respectively. In addition, it requires tasks for managing a Search Container and a Search Result Container to search, display and or deletion of Teacher Consultation entities, the high level details of which are provided in Table A-21. Table A-16 provides specific actions for displaying and or deleting the TC entities. The other use case of this system is to search a Student with an intention of assigning it to a Teacher Consultation entity followed by an operation for update of the status of the assigned Teacher Consultation entity. The high-level tasks involved in this activity are covered in Table A-23 while the lower level actions for each task are covered in Table A-17 to Table A-18. Finally, any visual model of a web application would require at least one Navigation Only Container to manage high-level navigation from the main (index) page. The details of this activity are like that captured in Table A-15. A complete mock-up of this system as created by one of the testers is illustrated in Figure A-6. The corresponding screen-shots of the auto-generated instance of system is provided in Figure 55.

Table A-19: Actions for “Creation of a Student entity” task

Task: Creation of a Student Entity. (See the “Add Student” page mock-up in Figure A-6)
The following task action specifications are required: A page for creation of Student Unique page name Creation of a DFY Container to assemble widgets for inserting details of a student A unique name for the DFY Container A unique label within the DFY Container for at least one detail of a student A data input field within the DFY Container for input of at least one attribute value of a student Button within the DFY Container to trigger an insert operation A "commit inserts" annotated navigation widget with the above-mentioned button as its source

Table A-20: Actions for the “Creation of a Question entity” task

Task: Creation of a Teacher Consultation (TC) entity. See the “Add Consultation” page mock-up in Figure A-6.
The following task action specifications are required: A page for creation of a TC entity A unique name for the page Creation of a DFY Container Unique name for the DFY Container Unique label within the DFY Container for teacher name Unique label each within the DFY Container for consultation date, time, duration and status Data input field within the DFY Container for teacher name A data input field within the DFY Container, each for consultation date, time, duration and status Button within the DFY Container to trigger an insert operation A "commit inserts" annotated navigation widget with the above-mentioned button as its source

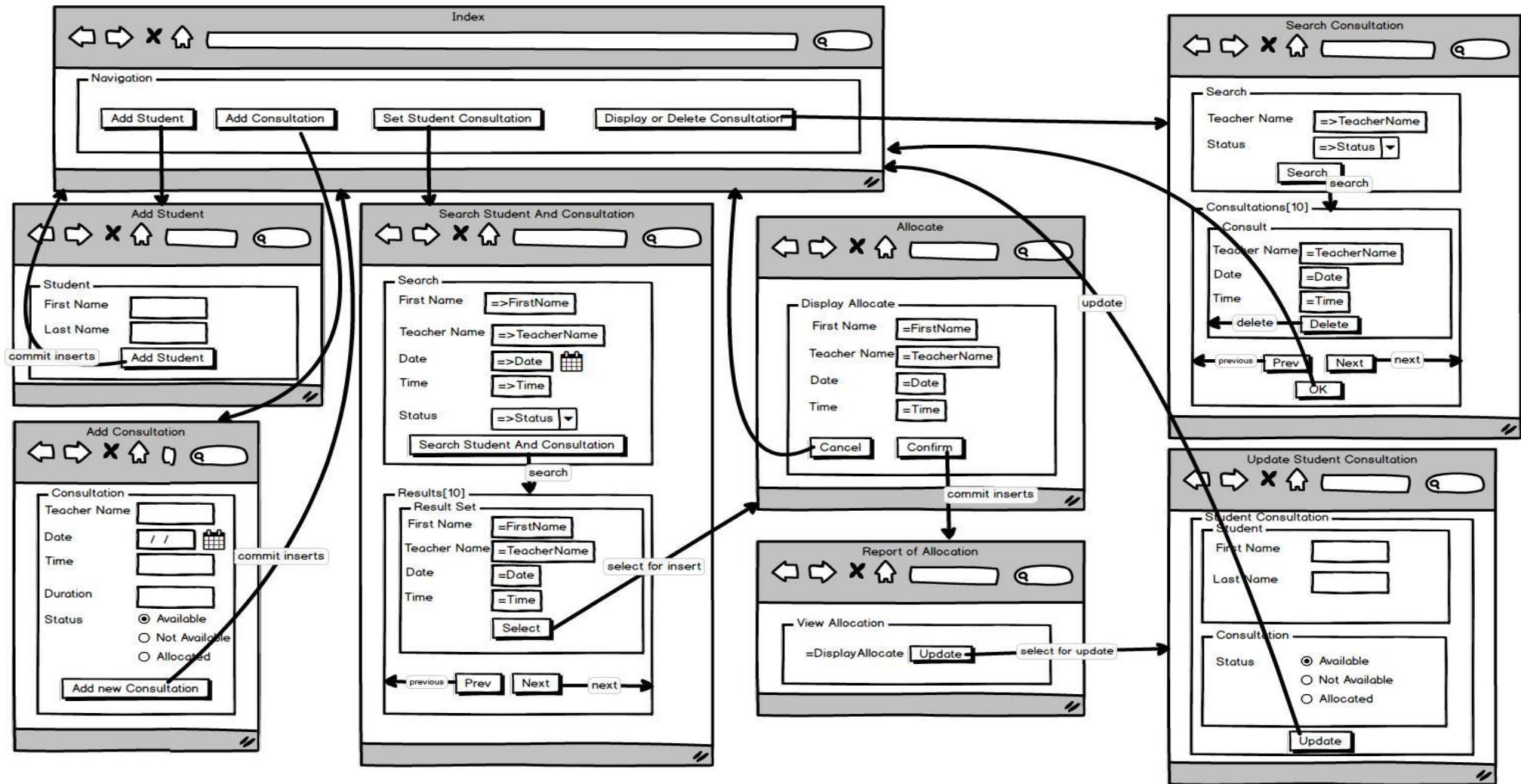


Figure A-6: A mock-up of the Teacher Student Consultation System

Table A-21: Sub-tasks for the “searching, displaying and or deletion of Teacher Consultation” task

Task: Search and View or Delete a Teacher Consultation (TC) entity. See the “Search Consultation” page mock-up in Figure A-6.
Sub-task 1: Use Search Container to search TC entity. The actions for this mock-up are same as in Table A-6 except this is with respect to TC entity. See the mock-up of the “Search” container in the “Search Consultation” page in Figure A-6.
Sub-task 2: Use Search Result Container to manage display of searched TC entities. The actions for this mock-up are same as in Table A-7 except this is with respect to TC entity. See the mock-up of the “Consultation[10]” container in the “Search Consultation” page in Figure A-6.
Sub-task 3: Use Data View Container within Search Result Container to view TC entity and or to delete it. The details are provided in Table A-22 See the mock-up of the “Consult” container within the “Consultation[10]” container in the “Search Consultation” page in Figure A-6.

Table A-22: Actions for displaying and or deleting one or more Teacher Consultation entities

Task: Use of Data View Container within Search Result Container to display the Teacher Consultation entity or to delete it. See the mock-up of the “Consult” container within the “Consultation[10]” container in the “Search Consultation” page in Figure A-6.
The following task action specifications are required: A nested container within Search Result Container The nested container has a name At least one label in nested container, for each entity in search criteria At least one “=reference widget” notation in nested container, for each entity in search criteria In nested container, no usage of input widgets without “=reference widget” notation A button in nested container, for deletion of a question A "delete" annotated navigation widget within the nested container The target of "delete" annotated navigation widget is the nested container and the source of the navigation widget is the above-mentioned button

Table A-23: Tasks for linking a Teacher Consultation (TC) with a Student and the update of TC status

Task: Link a Teacher Consultation (TC) With a Student and update TC status. See the mock-up of the “Search Student And Consultation” page, “Allocate” page, “Report of Allocation” page and “Update Student Consultation” page in Figure A-6.
Sub-task 1: Use Search Container to search TC and or Student entity. See Table A-24 for the actions required for this sub-task. See the mock-up of the “Search” container in the “Search Student And Consultation” page in Figure A-6, for reference.
Sub-task 2: Use Search Result Container to manage display of searched TC and or Student entities. The sub-task actions required are same as in Table A-7 except this is with respect to TC and or Student entity. See mock-up of the “Results[10]” container in the “Search Student And Consultation” page in Figure A-6.
Sub-task 3: Use Data View Container within Search Result Container to view TC and or Student entity. The actions required are shown n Table A-25. See the mock-up of the “Result Set” container within the “Results[10]” container in the “Search Student And Consultation” page in Figure A-6, for reference.
Sub-task 4: Use Data View Container to display and link selected Student with TC entity. The actions required for this sub-task are given in Table A-26. See the mock-up of the “Allocate” page in Figure A-6, for reference.
Sub-task 5: Use of Update Container to update status of Teacher Consultation. The actions required are shown in Table A-27. See the mock-up of the “Update Student Consultation” page in Figure A-6.

Table A-24: Actions in task to search for Student and or Teacher Consultation

Task: Use of Search Container(s) to search for Student and or Teacher Consultation. See the mock-up of the “Search” container in the “Search Student And Consultation” page in Figure A-6, for reference.
The following task action specifications are required: Creation of a Search Container for searching a Student and a Search Container for searching a TC or a single Search Container for searching both Student and TC together The Search Container has a name Use a label in Search Container for search criterion Use “=>Looked-up Widget” notation for searching criterion in Search Container Use appropriate reference in “=>Looked-up Widget” notation in search input criterion in Search Container No usage of input widget without “=>Looked-up Widget” notation Button in Search Container to trigger search A "search" annotated navigation widget within Search Container with the above-mentioned button as its source Target of "search" annotated navigation widget is a Search Result Container

Table A-25: Actions in task to select a data-set in in Search Result Container

Task: Use of Data View Container within Search Result Container to enable a selection. See the mock-up of the "Result Set" container within the "Results[10]" container in the "Search Student And Consultation" page in Figure A-6, for reference.

The following task action specifications are required:

- A nested container within a Search Result Container
- The nested container has a name
- At least one label for each entity in corresponding search criteria
- At least one "=reference widget" notation for each entity in corresponding search criteria
- A button within Search Result Container for selecting a search result data set for further processing
- A "select for insert" annotated navigation widget with the above-mentioned button as its source
- The target of "select for insert" annotated navigation widget is either a page or a container

Table A-26: Actions in task to display and link selected Student with Teacher Consultation entity

Task: Use of Data View Container to display and link selected Student with Consultation. See the mock-up of the "Allocate" page in Figure A-6, for reference.

The following task action specifications are required:

- Creation of a Data View Container to display selected entity. This
- The Data View Container has a name
- At least one label within the Data View Container
- At least one "=reference widget" notation within the Data View Container
- The reference in the "=reference widget" notation refers to a label name in the corresponding DFY Container
- The Data View Container has a button to confirm the displayed entity to be used for further processing in an insert business transaction
- A "temporarily store for insert" annotated navigation widget with the above-mentioned button as source, if another entity is to be linked. The mock-up in Figure A-6 does not have this since a Teacher entity and a Student entity will be displayed together in the "Display Allocated" container in the "Allocate" page and the only action required is to link them in storage using a "commit inserts" annotated navigation widget. The target of the above mentioned "temporarily store for insert" annotated navigation widget is a Search Container.
- A "commit inserts" annotated navigation widget with the above-mentioned button as source, if no more entity is required to be linked
- The target of "commit inserts" annotated navigation widget is a page. For example, in Figure A-6, the mock-up shows that "Report of Allocation" page is the target of the "commit inserts" annotated navigation widget.

Table A-27: Actions in task to update status of Teacher Consultation entity

Task: Use of Update Container to update status of Teacher Consultation. The actions required are shown in Table A-27. See the mock-up of the "Update Student Consultation" page in Figure A-6.

The following task action specifications are required:

- A page for update
- A unique name for the page
- The page is the target of a "select for update" annotated navigation widget. For example, in Figure A-6, the mock-up shows that "Update Student Consultation" page is the target of the "select for update" annotated navigation widget from the "Report of Allocation" page.
- An Update Container within the page for update of a consultation entity
- The name of the Update Container is same as the corresponding DFY Container for creation of consultation entity
- A status field within the Update Container so that the status can be updated
- A button within the Update Container to trigger the update operation
- A "update" annotated navigation widget with the above-mentioned button as its source

Appendix 6.2.6 Visual Modelling Tasks in the Patient-Dietician Consultation System

This section discusses the visual modelling tasks in the Patient-Dietician Consultation System. Essentially it requires a DFY Container each for the creation of a Patient and for a Dietician entity. The tasks involved in the creation of these DFY Containers are shown in Table A-28 and Table A-29 respectively. In addition, it requires tasks for managing a Search Container and a Search Result Container to search, view, and select a Patient and to add further details such as weight and or height. The high-level details of these tasks are provided in Table A-30 and some of the lower-level task actions are covered in Table A-31, Table A-32 and Table A-33. The other use case of this system is to search a Patient and a Dietician with an intention of linking the Patient with the Dietician. The high-level tasks involved in this activity are covered in Table A-34 while Table A-35 and Table A-36 provide some of the lower lever task actions. Finally, any visual model of a web application would require at least one Navigation Only Container to manage prominent level navigation at least from the main (index) page. The details of this activity are like that defined in Table A-15, so it is not reproduced in this section. A complete mock-up of this system as created by one of the testers is illustrated in Figure A-7. The corresponding screen-shots of the auto-generated instance of the system is provided in Figure 54.

Table A-28: Actions in “Creation of a Patient entity” task

<p>Task: Creation of a Patient Entity (See the “Add Patient” page mock-up in Figure A-7)</p> <p>The following task action specifications are required:</p> <ul style="list-style-type: none"> A page for creation of patient entity A unique name for the page Creation of a DFY Container Unique name for the DFY Container Unique label within the DFY Container for patient name Unique label within the DFY Container for patient medical number Data input field within the DFY Container for patient name Data input field within the DFY Container for medical number Button within the DFY Container to trigger an insert operation A "commit inserts" annotated navigation within with the above-mentioned button as its source A nested DFY Container for either height and or weight of the patient Unique label within the nested DFY Container for height and or weight Data input field within the nested DFY Container for either height and or weight
--

Table A-29: Actions in “Creation of a Dietician entity” task

<p>Task: Creation of a Dietician Entity (See the “Add Dietician” page mock-up in Figure A-7)</p> <p>The following task action specifications are required:</p> <ul style="list-style-type: none"> A page for creation of a dietician entity A unique name for the page Creation of a DFY Container Unique name for the DFY Container Unique label(s) within the DFY Container to represent a dietician Data input field(s) within the DFY Container enter details of a dietician Button within the DFY Container to trigger an insert operation A "commit inserts" annotated navigation widget with the above-mentioned button as its source
--

Table A-30: Sub-tasks for storing further details of existing Patient during a consultation

<p>Task: Search, view, select Patient and add further details (See the “Update Patient” page mock-up in Figure A-7)</p>
<p>Sub-task 1: Use Search Container to search Patient entity. See the “Search Patient” container mock-up in the “Update Patient” page in Figure A-7). This sub-task is same as in Table A-6 except this is with respect to Patient entity.</p>
<p>Sub-task 2: Use Search Result Container to manage display of searched Patient entities. See the “Patient[1]” container mock-up in the “Update Patient” page in Figure A-7. This sub-task is same as in Table A-7 except this is with respect to Patient entity.</p>
<p>Sub-task 3: Use Data View Container within Search Result Container to display the Patient entity with an intention to add additional details. See the “APatient” container within “Patient[1]” container mock-up in the “Update Patient” page in Figure A-7. Table A-31 contains the actions required.</p>
<p>Sub-task 4: Use Data View Container to display selected Patient and potentially add further details. See the not nested “APatient” container mock-up in the “Update Patient” page in Figure A-7. Table A-32 contains the actions required.</p>
<p>Sub-task 5: Use a DFY Container to insert further details such as Height or Weight of a patient during a consultation. See the “Height” container mock-up in the “Update Patient” page in Figure A-7. Table A-33 contains the actions required.</p>

Table A-31: Action for displaying a Patient entity in a Data View Container within Search Result Container

<p>Task: Use of Data View Container within Search Result Container to display the Patient entity with an intention of adding additional details. See the “APatient” container within “Patient[1]” container mock-up in the “Update Patient” page in Figure A-7.</p>
<p>The following task action specifications are required: A nested container exists within Search Result Container The nested container has a name At least one label within nested container, for each entity in search criteria At least one “=reference widget” notation within nested container, for each entity in search criteria A button in nested container, to trigger selection a search result data set as a part of an insert transaction A "select for insert" annotated navigation widget within the nested container with the above-mentioned button as its source The target of "select for insert" annotated navigation widget is either a page or a container</p>

Table A-32: Actions in task to display a Patient and potentially add further details

<p>Task: Use of Data View Container to display a Patient to potentially add further details. See the not nested “APatient” container mock-up in the “Update Patient” page in Figure A-7.</p>
<p>The following task action specifications are required: A Data View Container to display selected patient The container has a name At least one label within the container At least one “=reference widget” notation within the container The reference in the “=reference widget” notation refers to a Label name in a create Patient DFY Container A button within the container to trigger acceptance of the displayed patient entity for further processing as a part of an insert business transaction A "temporarily store for insert" annotated navigation widget with the above-mentioned button as its source The temporarily store for insert" annotated navigation widget targets either a height or weight DFY Container</p>

Table A-33: Actions in task to insert further details such as Height or Weight of a patient during a consultation

<p>Task: Use of Data Field Yielding Container to add further details such as Height or Weight of an existing Patient. See the “Height” container mock-up in the “Update Patient” page in Figure A-7.</p>
<p>The following task action specifications are required: A DFY Container The DFY Container name is same as in a corresponding Patient DFY Container The label for height or weight is same as in corresponding patient DFY Container A data input field for either height or weight A button to confirm new height or weight A "commit inserts" annotated navigation widget with the above-mentioned button as its source The "commit inserts" annotated navigation widget targets either a page containing a data view container or a data view container to display the newly added details of the existing patient</p>

Table A-34: Sub-tasks for assigning a Patient to a Dietician

Task: Assigning a Patient to a Dietician. See the mock-up in the "Search Dietician" page in Figure A-7.
Sub-task 1: Use Search Container(s) to search Patient and Dietician. See the "Search Dietician" and "Search Patient" container mock-up in the "Search Dietician" page in Figure A-7. Table A-35 discusses the actions required.
Sub-task 2: Use Search Result Container to manage display of searched entities. See the "Dietician[1]" and "Patient[1]" container mock-up in the "Search Dietician" page in Figure A-7. The discussion on the actions required are same as in Table A-7 except this is with respect to Patient or Dietician or both Patient and Dietician.
Sub-task 3: Use Data View Container within Search Result Container to display the Patient or Dietician or both Patient and Dietician. See the nested "ADietician" and the nested "APatient" container mock-up in the "Search Dietician" page in Figure A-7. The discussion on actions required are same as in Table A-31 except this is either with respect to Patient or Dietician or both.
Sub-task 4: Use of Data View Containers to display and link selected Patient with selected Dietician. See the "Dietician" and the not nested "APatient" container mock-up in the "Search Dietician" page in Figure A-7. Table A-36 discusses the actions required.

Table A-35: Action in task to search Patient and Dietician

Task: Use Search Container(s) to search Patient and Dietician. See the "Search Dietician" and "Search Patient" container mock-up in the "Search Dietician" page in Figure A-7.
The following task action specifications are required: A Search Container to search a Patient and a Search Container to search Dietician or single Search Container for searching both Patient and Dietician together The Search Container(s) has/have a name At least one label for search criterion in each Search Container At least one "=>Looked-up Widget" notation for search criterion in each Search Container Each "=>Looked-up Widget" notation has appropriate reference to DFYW A button within Search Container to trigger the search operation A "search" annotated navigation widget with the above-mentioned button as the source Target of "search" annotated navigation widget is a Search Result Container

Table A-36: Actions in task to display and link selected Patient with selected Dietician

Task: Use of Data View Containers to display and link selected Patient with selected Dietician. See the "Dietician" and the not nested "APatient" container mock-up in the "Search Dietician" page in Figure A-7
The following task action specifications are required: Two Data View Containers to display a selected entity. The container has a name At least one label within the Data View Container At least one "=>Looked-up Widget" notation within the Data View Container The reference in the "=>Looked-up Widget" notation refers to a label name in the corresponding DFY Container A button within the Data View Container to confirm acceptance of the displayed entity A "temporarily store for insert" annotated navigation widget with the above-mentioned button as source, if another entity is to be linked The target of the "temporarily store for insert" annotated navigation widget is a Search Container A "commit inserts" annotated navigation widget with the above-mentioned button as source, if no more entity is required to be linked The target of "commit inserts" annotated navigation widget is a page.

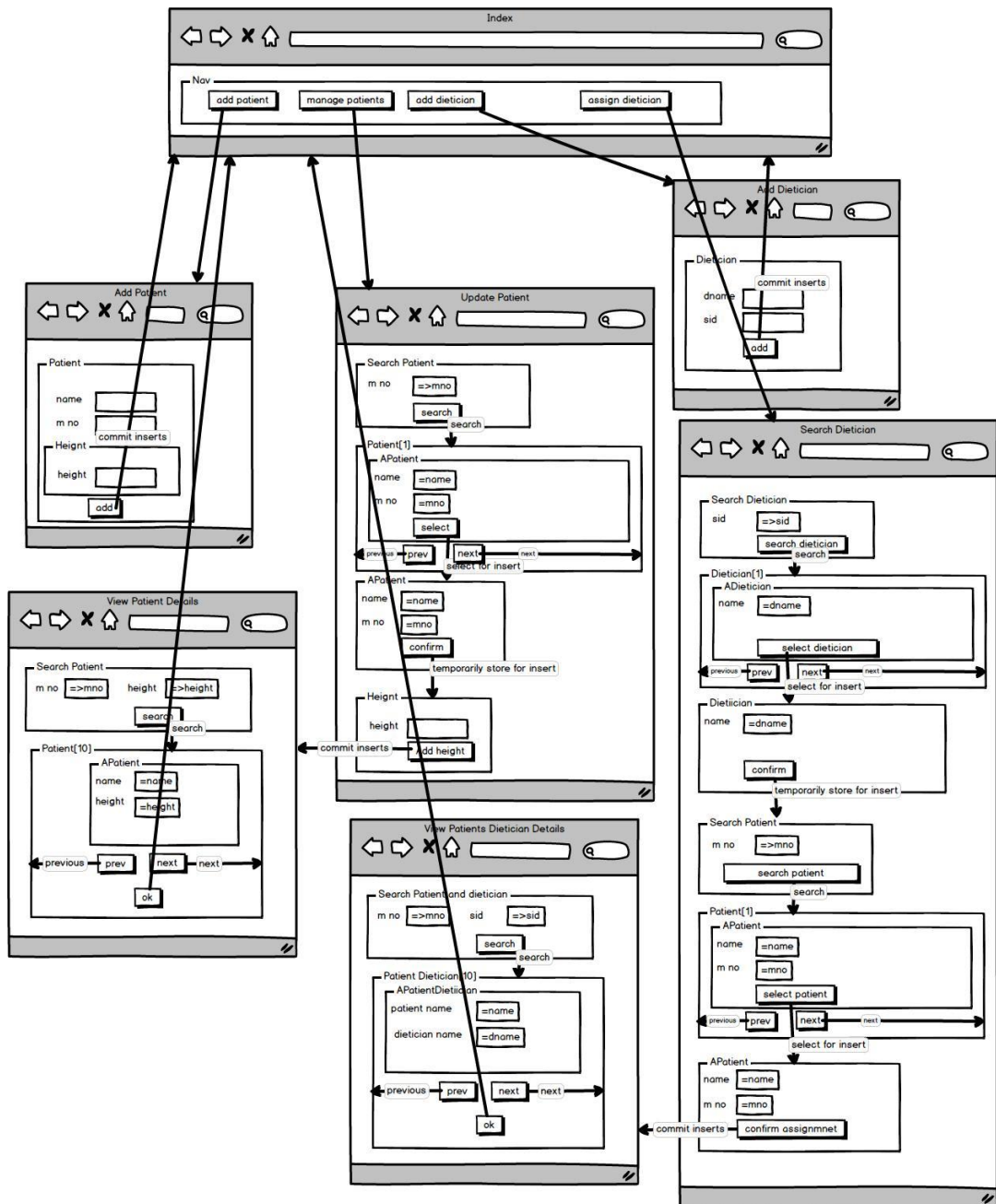


Figure A-7: A mock-up of the Patient Dietician System

Appendix 6.2.7 End user tasks in the auto-generated Teacher Consultation System

This section discusses the end-user tasks in the Teacher Consultation System. Essentially it requires creation of a Student and a Teacher Consultation entity. The tasks involved in the creation of these entities are shown in Table A-37 and Table A-38 respectively. In addition, it requires tasks for managing Teacher Consultation entities. This includes searching, displaying search result with an option to delete a selected Teacher Consultation entity, the details of which are provided in Table A-39. The other use case of this system is to search a Student with an intention of assigning it to a Teacher Consultation entity followed by an operation for update of the status of the assigned Teacher Consultation entity. The tasks involved in this use case are covered in Table A-40. Finally, any web application would require at least one Navigation Only Container to manage high-level navigation at least from the main (index) page. The user actions for this are found in Table A-41.

Table A-37: Actions in the “Creation of a Student entity” task

Task: Creation of a Student Entity
The following task actions are required: Enter data for a new Student entity Click a button to commit new Student entity to database

Table A-38: Actions in the “Creation of a Teacher Consultation entity” task

Task: Creation of a Teacher Consultation (TC) Entity
The following task action specifications are required: Enter data for a new Teacher Consultation entity Click a button to commit new Teacher Consultation entity to database

Table A-39: Actions in the task for “searching, displaying and or deletion of a Teacher Consultation entity”

Task: Search and View or Delete a Teacher Consultation (TC) entity
The following task action specifications are required: Enter the search criteria for a TC in a Search Container Click a button to perform the search Find details of the desired TC entity by traversing through the result in a Search Result Container using buttons associated with “previous” and or “next” annotated navigation widgets. Optionally delete a desired TC entity by selecting a button next to it in the Search Result Container

Table A-40: Actions in task for linking a Teacher Consultation (TC) with a Student and the update of TC status

Task: Link a Teacher Consultation (TC) With a Student and update TC status
The following task action specifications are required: Enter the search criteria for a TC and a Student entity in a Search Container. Depending on the design this may be done in two independent searches or in a combined search Click a button to perform the search Find details of the desired TC and Student entity by traversing through the result in a Search Result Container using buttons associated with “previous” and or “next” annotated navigation widgets. Select a TC and a Student entity by clicking a button in (each) Search Result Container. View the selected TC and the Student entity before clicking a button to confirm linking them together View the linked entities and click a button to enable update (editing) of status of the TC entity in an Update Container Enter updated data in the Update Container Click a button to confirm the update

Table A-41: Actions in task for using a main navigation header in the Teacher Consultation System

Task: Navigate from header section to sub-sections of the Teacher Consultation System
The following task action specifications are required: Click a button to navigate to the container for creation of Student entity Click a button to navigate to the container for creation of Teacher Consultation entity Click a button to navigate to the section for management of Teacher Consultation entities Click a button to navigate to the section for linking a Student entity and a Teacher Consultation entity together followed by the status update of the Teacher Consultancy entity

Appendix 6.2.8 End user tasks in the auto-generated Question & Answer System

This section discusses the end-user tasks in the Question & Answer System. Essentially it requires creation of a Question and an Expert entity to answer the question. The tasks involved in the creation of these entities are like that shown in Table A-37 and Table A-38 except they are done with respect to a Question entity and an Expert entity. It also requires a task to allow an Expert to login. The actions for logging in are like that of search in Table A-37 except that there is no Search Result container to display the result of the search. In addition, it requires tasks for managing Question entities. This includes searching and displaying the search result, the details of which are provided in Table A-39. The other use case of this system is to search a Question with an intention of assigning it to an Expert or for answering by an Expert or for its deletion. The tasks involved in this use case are covered in Table A-44. Finally, any web application would require at least one Navigation Only Container to manage high-level navigation from the main (index) page. The user actions for this task are found in Table A-45.

Table A-42: Actions in task for “searching and displaying Question entities”

Task: Search and View Question entities
The following task action specifications are required: Enter the search criteria for a Question in a Search Container Click a button to perform the search View Question entities by traversing through the result in a Search Result Container using buttons associated with “previous” and or “next” annotated navigation widgets.

Table A-43: Actions in task for answering or deleting a Question or for its assignment to an Expert

Task: Search Question to answer it or to link with an Expert or to delete it
The following task action specifications are required: Enter the search criteria for a Question in a Search Container Click a button to trigger the search operation Find details of the desired Question entity by traversing through the result in a Search Result Container using “previous” and or “next” buttons. Select a Question entity by clicking an appropriate button in the Search Result Container with an intention of assigning to an Expert or for its update or its deletion If the selected button is for update, perform the update in the Update Container and click a button to confirm the update If the selected button is for deletion observe that the selected Question gets deleted in the Search Result Container If the selected button is for linking with an Expert: Enter the search criteria for an Expert in a Search Container Click a button to trigger the search operation Find details of the desired Expert entity by traversing through the result in a Search Result Container using buttons associated with “previous” and or “next” annotated navigation widgets. Select a desired Expert entity by clicking an appropriate button in the Search Result Container with an intention of assigning to the previously selected Question Observe that the selected Question and Expert are displayed in Data View Container with a button next to it to confirm the linkage Click the button to confirm the linkage

Table A-44: Actions in task for using a main navigation header in the Question Answer System

Task: Navigate from header section to sub-sections of the Question Answer System
The following task action specifications are required: Click a button to navigate to the container for creation of Question entity Click a button to navigate to the container for creation of Expert entity Click a button to navigate to the section for management of Question entities Click a button to navigate to the section for linking a Question with a logged in Expert where the Expert may perform status update of the Question entity or delete a Question

Appendix 6.2.9 End User Tasks in the Patient-Dietician Consultation System

This section discusses the end user tasks in the Patient-Dietician Consultation System. Essentially it requires tasks for the creation of a Patient and for a Dietician entity. The tasks involved in the creation of these entities are like that shown in Table A-37 and Table A-38 except they are done with respect to a Patient entity and a Dietician entity. In addition, it requires tasks for adding additional information such as weight or height of existing patients during consultation with a Dietician. The details of these tasks are provided in Table A-45. The other use case of this system is to search a Patient and a Dietician with an intention of linking the Patient with the Dietician. The actions involved in this activity are covered in Table A-47. Finally, any web application would require at least one Navigation Only Container to manage prominent level navigation from the main (index) page. The details of this activity are in Table A-48.

Table A-45: Actions in task for storing further details of existing Patient during a consultation

Task: Search, view, select Patient and add further details
The following task action specifications are required: Enter the search criteria for a Patient in a Search Container. Click a button to trigger the search operation. Find details of the desired Patient entity by traversing through the result in a Search Result Container using buttons associated with “previous” and or “next” annotated navigation widgets. Select a Patient entity by clicking a button in the Search Result Container with an intention of adding additional information about the patient. This will cause the selected Patient’s details to be displayed in a separate Data View Container. In the Data View Container click a button to confirm further details of the selected Patient to be added. This will open a DFY Container to add the further details. Add the further details in the DFY Container. Click a button to confirm the insertion of further details, causing the details to be stored in the database.

Table A-46: Actions in task for assigning a Patient to a Dietician

Task: Assigning a Patient to a Dietician
The following task action specifications are required: Enter the search criteria for a Patient in a Search Container. This action may optionally include actions for entering search criteria for searching a Dietician as well. In such a case only one search is required. If not, a sequence of searches is required, one for Patient and one for the Dietician. The following actions assume both criteria are included together. Click a button to trigger the search operation. Find details of the desired Patient and Dietician entity by traversing through the result in a Search Result Container using “previous” and or “next” buttons. Select a Patient and Dietician entity by clicking a button in the Search Result Container with an intention of linking them together. This will cause the selected Patient and Dietician details to be displayed in a separate Data View Container In the Data View Container click a button to confirm the selection for linkage. This will create a link between the Patient and the Dietician in the database.

Table A-47: Actions in task for using a main navigation header in the Patient Dietician System

<p>Task: Navigate from header section to sub-sections of the Patient Dietician System</p>
<p>The following task action specifications are required:</p> <ul style="list-style-type: none">Click a button to navigate to the container for creation of Patient entityClick a button to navigate to the container for creation of Dietician entityClick a button to navigate to the section for adding additional details of a Patient during a consultation sessionClick a button to navigate to the section for linking a Patient with a Dietician

INDEX

- "=>looked-up widget" notation, 56
- "=reference widget" notation, 59
- "commit inserts" annotation, 54, 61, 64, 67, 111, 154
- "delete" annotation, 57, 59, 76, 155
- "next" annotation, 155
- "search" annotation, 54, 56, 79, 94, 154
- "select for insert" annotation, 60, 61, 69, 117, 154
- "select for update" annotation, 66, 67, 69, 154
- "temporarily store for insert" annotation, 61, 63, 64, 76, 114, 154
- "update" annotation, 66, 67, 79, 154
- A Client-side Controller (CSC), 94
- activity diagram, 28
- Agile software development process, 6
- analyst, 2
- architecture, 12
- Asynchronous communication, 14
- Asynchronous JavaScript and XML (AJAX), xi, 15
- Business Analyst, xi, 3, 7, 130, 191, 248
- business transaction, 8
- calculation method, 137, 138
- C-INCAMI, xi, 133, 135, 137, 138, 139, 142, 143, 147, 148, 160, 161, 162, 171, 174, 175, 176, 177, 183, 195
- class diagram, 25, 106
- client-side processing, 14
- client-side storage (local storage), 112
- Codelgniter, 99, 102, 202
- cognitive load, 24, 37, 172
- Computation Independent Models (CIM), 28
- Concept model, 136
- conceptual model, 20
- Container Widget, 26
- CRUD, xi, 38, 92, 104, 195
- CSM, xi, 93, 95, 96, 97, 98, 100, 102, 103, 104, 106, 107, 108, 112, 113, 114, 117, 119, 121, 122, 123, 125, 126, 128, 129, 238, 239, 242
- CSV, xi, 93, 95, 96, 97, 98, 99, 100, 104, 108, 112, 114, 121, 122, 125, 126, 128, 129, 239, 243
- CTT, 28
- Data Input Container*, 78, 79, 86
- Data View Selection Container, 76, 79
- Database Field Yielding Container*
 - DFYC, xi, 54, 56, 60, 62, 63, 64, 67, 77, 88, 147, 148, 149, 199
- database schema, 23, 32, 83, 84, 126, 201, 202, 204
- Data-binding, 100
- Design Cycle, 41, 46, 190
- designer, 2
- DFYC, xi, 54, 65
- DFYW, xi, 53, 54, 55, 62, 74, 75, 78, 84, 86, 87, 94, 97, 101, 105, 110, 222, 223, 224, 225, 226, 227, 231, 233, 240, 241, 262
- DSR, xi, 40, 41, 42, 44, 139, 192
- DSR in IS, 40, 41, 185
- dynamic view of UI, 25
- effectiveness in use, 136, 142, 143, 144, 155, 161, 163, 167, 171, 172, 173, 175, 177
- efficiency in use, 136, 142, 143, 144, 146, 155, 161, 163, 167, 171, 172, 173, 175, 177
- elementary indicator, 138
- E-R, xi, 85, 86, 88, 89, 91, 224, 228
- ERP, xi, 4, 5
- Evaluation indicator, 138
- For-Each Container, 76, 77, 79
- global indicator, 138, 172
- GUI, xi, 31
- HTML, xi, 11, 13, 14, 33, 77, 82, 97, 99, 101, 200
- implementer, 2
- Input widget
 - Data input widget, 25
- insert business transaction, 60
- ISO, xi, 133, 135, 136, 188, 190, 192, 195
- Iterative and Incremental development process, 189
- JavaScript, xi, 14, 15, 82, 98, 100, 109, 126, 202
- jQuery, 102, 109, 112, 126, 189, 202
- JSON, xi, 32, 82, 103, 106, 109, 198, 202, 233
- Knockout.js, 99, 100, 109, 112, 113, 122, 126, 189, 202, 239
- latency of response, 14
- learnability, 134
- Look-up widget*, 56
- M&E, xi, 133, 135, 137, 138
- MDA, xi, 28
- MDE, xi, 27, 28, 189
- measurement metric, 137
- meta-model, 24
- Multi-Row Container, 76, 77, 79
- MVC, 12

MVC-MC, 18
MVC-MVC, 17
Navigation Only Container, 54, 55, 79, 144, 147, 152, 169, 251, 256, 260, 264, 265, 266, 267
Navigation Widget, 25
non-functional features, 39
non-functional requirements, 2, 133, 183
OMG, 28
OODD, 19
OOHDM, xi, 20, 22, 27
OOWS, xi, 20, 22, 27
ORM, xi, 201
partial indicator, 138
physical (computational) transaction, 9
PIM, xi, 28
PSM, xi, 29
quality-in-use, 133, 134, 139, 192
RDBMS, xi, 11
Relevance cycle, 41
RIA, xi, xii, 8, 12, 14, 15, 17, 18, 19, 21, 22, 24, 25, 26, 27, 35, 36, 37, 38, 39, 40, 71, 96, 97, 126, 133, 186, 188, 189, 193, 195, 198
Rigor Cycle, 41
satisfaction in use, 136, 142, 143, 144, 155, 160, 161, 162, 163, 168, 171, 173, 175, 177, 183, 192
SBOML, xi, 23
Search Container, 54, 56, 57, 58, 63, 79, 88, 90, 91, 93, 94, 95, 96, 97, 98, 99, 100, 102, 105, 107, 108, 115, 129, 143, 147, 150, 154, 168, 225, 226, 231, 235, 237, 239, 251, 253, 254, 256, 258, 259, 260, 261, 262, 264, 265, 266
Search Result Container, 57, 58, 59, 76, 77, 79, 93, 94, 95, 96, 98, 99, 100, 103, 105, 106, 108, 112, 113, 119, 121, 122, 123, 125, 128, 144, 147, 148, 151, 169, 229, 231, 234, 235, 236, 237, 251, 253, 254, 255, 256, 258, 259, 260, 261, 262, 264, 265, 266
Server-side Controller
 SRS, 15
Server-Side Model
 SSM, 96
service widget, 72, 78, 79
SME, 4
software crisis
 software crisis, 2
SRS, xi, 2
static view of UI, 25
sub-task completeness effectiveness, 144, 145, 146, 155, 162, 167
sub-task completeness efficiency, 144, 146, 159, 162, 168
sub-task correctness effectiveness, 144, 145, 146, 155, 156, 162, 167, 168
sub-task correctness efficiency, 144, 146, 158, 162, 168
SUS, xii, 147, 160, 168, 173, 175, 177, 178, 183, 192
System Analysis, 7
task successfulness effectiveness, 144, 145, 146, 155, 162, 167, 170
task successfulness efficiency, 144, 146, 159, 162, 170
traditional web application, 11, 12, 22
transactional web, 9
UI mock-up
 Wireframe, 1
usability of the auto-generated application, 183, 192
usability of the mock-up language, 38, 39, 133, 168, 171, 183, 190, 191, 192, 193, 201
usability testing, 133, 135, 137, 138, 139, 140, 144, 162, 164, 174, 175, 176, 177, 183, 188, 191, 192, 195, 196, 201
use case, 31
UWE, 20
UWE-R, 22
Validating usability, 38
Waterfall, 3, 5
web 2.0, 14
web application server, 11
web service, 32, 73, 80, 196, 198, 200, 204
WebML, xii, 20, 22, 48
WSRequest, 72, 74, 78, 198, 200
WYSIWYG, xii, 21, 32, 35, 37
XHTML, 33
XML, 15