Model Driven Abstraction of Enterprise Tasks and Processes for User Applications

Using Microsoft Domain Specific Language Tools

Ruskin Dantra

This thesis is the entirety to fulfil the requirements to achieve a Masters in (Software) Engineering at The University of Auckland, New Zealand.

December 2008

© 2008 Ruskin Dantra

Abstract

Enterprise tasks and processes tend to be complicated and consume considerable resources in terms of time and personnel. They also involve repetitive information with a small amount of user entered data. Writing a report to extract information from a database is an example of such an enterprise task. This task can further be made complicated if the report writing language is unique to the enterprise and if the database in which the information is stored is complicated.

Current practice involves end-users manually creating the report. This manual process is prone to simple errors such as spelling mistakes and other more complicated errors such as misunderstanding the language semantics.

This thesis attempts to solve this problem by raising the level of abstraction using a model driven approach coupled with a Domain Specific Language. Our approach involves designing a meta-model of the enterprise task and then exposing this meta-model to the end-user via a user interface. As a basis for the meta-model, we used a propriety report writing language developed by Prism NZ. Creating this meta-model enabled us to abstract irrelevant details of the task from the user. These details included language semantics, constraints, database tables and their corresponding relationships. Our process involved creating the meta-model using Microsoft DSL Tools, adding custom constraints, exposing this meta-model and eventually generating the required report. The end-user interface which exposed the meta-model was designed using visual notations further improving usability and making the report writing process intuitive.

Although this thesis does solve the problem by visually exposing a complicated reporting language it has a major limitation which is often experienced by most visual languages, it does not cater for complicated combinations of the reporting constructs which expert users may use. The thesis is therefore aimed to help novice and intermediate users to quickly start designing reports.

We evaluated our solution using the cognitive dimensions framework and surveying live users. Our results showed that even though writing a report is usually a complicated task it can be made easier by increasing the level of abstraction using models and allowing the end-user to create a report visually as opposed to creating it textually.

Acknowledgements

I would like to pass on my warmest regards to my supervisor Professor John Grundy who spent a lot of time guiding me before and during the course of the thesis. He helped me with the initial idea and helped me during the remainder of the thesis by providing sound advice and input.

I would also like to thank my co-supervisor Professor John Hosking for providing valuable input on the design of the visual elements of the model.

Thank you to the team at Prism New Zealand, Auckland who assisted me during the entire course of the thesis and were patient and understanding. Would like to thank Grant Davidson and Steve Pearce for their encouragement and feedback.

Also want to show my appreciation towards UniServices Auckland and The University of Auckland for providing valuable resources for my use during the year.

Last but not least, would like to thank my family and friends for their support and help in every aspect during the past year.

People/Organisations Involved

- Professor John Grundy
  HOD ECE
  University of Auckland
  New Zealand
  john-g@cs.auckland.ac.nz

- Professor John Hosking
  University of Auckland
  New Zealand
  john@cs.auckland.ac.nz

- Grant Davidson
  Production Team Leader
  Prism New Zealand, Auckland
  grant.davidson@prism-nz.com

- Steve Pearce
  Systems Team Leader
  Prism New Zealand, Auckland
  steve.pearce@prism-nz.com

- Prism New Zealand
  Unit 3b
  Saatchi & Saatchi Building
  Parnell
  Auckland, New Zealand

- Auckland UniServices
  Uniservices House, Level 10
  Auckland, New Zealand
  +64 9 373 7522

- University of Auckland
  Private Bag 92019
  Auckland Mail Centre
  Auckland 1142, New Zealand

Table of Contents

Abstract .... **i**

Acknowledgements .... **ii**

Table of Contents .... **iii**

List of Figures .... **ix**

List of Tables .... **xiv**

Chapter 1 - Introduction **1**

1.1 Introduction 1

1.2 Introduction to Prism 1

1.2.1 Core Business 1

1.2.2 Prism WIN 2

1.2.3 Business Domain 2

1.3 Report Writer Module 5

1.4 Report Writer Language (Prism New Zealand, 2005) 6

1.4.1 RWL Walkthrough 7

1.5 Introduction to the Problem Domain 7

1.5.1 Database Complexity 7

1.5.2 Report Writer Language Complexity 8

1.6 Motivation 9

1.6.1 Reduce database complexity 9

1.6.2 Intuitive RWL tool 9

1.7 Thesis Ambition 9

1.8 Thesis Overview 10

1.9 Summary 11

Chapter 2 - Related Work **12**

2.1 Introduction 12

2.2 Development Methodologies 12

2.2.1 Incremental Model 12

2.2.2 Agile Development Model 13

2.3 Model-Driven Software Development 14

2.3.1 Introduction 14

2.3.2 MDSD Approach and Terminology 14

2.3.3 Adaptive-Object modeling 18

2.3.4 Metamodeling 19
2.3.5 Model Transformations 21
2.3.6 Software Factories (Greenfield, 2005) 22
2.4 Domain Specific Development and Languages 23
2.4.1 Introduction 23
2.4.2 Textual DSL 27
2.4.3 Graphical DSL (Visual Languages) 28
2.4.4 Advantages/Disadvantages of DSLs 31
2.5 DSL Tools 31
2.5.1 Microsoft DSL Tools 31
2.5.2 Marama 34
2.5.3 Comparison (MS DSL Tools Vs Marama) 36
2.5.4 Other DSL Tools 37
2.6 Prism WIN Scribe IDE 40
2.6.1 Features 41
2.6.2 Limitations 44
2.7 Summary 44
Chapter 3 - Our Approach 46
3.1 Introduction 46
3.2 High Level Approach View 46
3.3 Software Technologies Used 46
3.3.1 Microsoft DSL Tools 46
3.3.2 Windows Presentation Framework 48
3.3.3 Language Integrated Querying 48
3.4 Methodologies and Standards 48
3.4.1 Unified Modeling Language (UML) 48
3.4.2 Model-Driven Software Development 48
3.4.3 Agile development 51
3.5 Gathering Requirements 51
3.5.1 Tool Requirements 51
3.5.2 Meta-Model Requirements 51
3.6 Summary 51
Chapter 4 - Requirements 52
4.1 Introduction 52

4.2 Functional Requirements 52
4.2.1 Report Writer Meta-Model 52
4.2.2 RWM Shell Host 58
4.3 Non-functional Requirements 64
4.3.1 Report Writer Meta-Model 64
4.3.2 RWM Shell Host 67
4.4 Summary 70
Chapter 5 - Design 72
5.1 Introduction 72
5.2 Stakeholders 72
5.2.1 Meta-Model 72
5.2.2 Shell Host 72
5.3 Use Cases 72
5.3.1 Developer 72
5.3.2 Report Designer 73
5.4 Object oriented Design/Analysis 74
5.5 Overall architecture 86
5.5.1 Class Diagram Approach 86
5.5.2 RWM Shell Approach 87
5.6 Approach Overview 87
5.7 Class Diagram-Based Design 88
5.7.1 Overview 88
5.7.2 Meta-Model Design 88
5.7.3 Shell Host Design 98
5.8 RWM Shell Approach Design 99
5.8.1 Overview 99
5.8.2 Meta-Model Design 100
5.8.3 Shell Host Design 104
5.9 Design for users (non-functional perspective) 107
5.9.1 Developer 108
5.9.2 Report Designer 110
5.10 User Interface Design 111
5.10.1 Class Diagram Approach UI 111
5.10.2 RWM Shell Approach User Interface 113

5.11 Summary 115
Chapter 6 - Implementation 116
6.1 Introduction 116
6.2 DSL Terminology 116
6.2.1 The DSL Project 116
6.2.2 The DslPackage Project 117
6.2.3 Path Syntax 117
6.2.4 The Toolbox 117
6.2.5 Element Merge Directives 118
6.2.6 The Model Explorer 119
6.2.7 Connection Builders 119
6.2.8 External Types 120
6.2.9 Serialization 121
6.2.10 Text Templates 122
6.3 Class Diagram Approach Implementation 123
6.3.1 Meta-Meta-Model Development 123
6.3.2 Meta-Model Development 135
6.3.3 Shell Host Development 143
6.4 RWM Shell Approach Implementation 149
6.4.1 Meta-Model Development 149
6.4.2 Shell Host Development 162
6.5 Challenges Faced 163
6.5.1 Class Diagram Approach 163
6.5.2 RWM Shell Approach 163
6.5.3 Approach Comparison 164
6.6 Summary 165
Chapter 7 - Case Studies 166
7.1 Introduction 166
7.2 Scenario 1: Designing a report 166
7.2.1 Requirements 166
7.2.2 Requirement analysis 166
7.2.3 Meeting the requirements 167
7.2.4 Conclusion 170
7.3 Scenario 2: Adding new model information 171

7.3.1 Requirements 171
7.3.2 Requirement analysis 171
7.3.3 Meeting the requirements 172
7.3.4 Conclusion 178
7.4 Summary 178
Chapter 8 - Evaluation 179
8.1 Introduction 179
8.2 Design Evaluation 179
8.2.1 Champagne Prototyping 179
8.2.2 Cognitive Dimensions 180
8.3 Survey Evaluation 188
8.3.1 Developer 188
8.3.2 Report Designer 193
8.4 Cognitive Dimensions V.s. Survey Evaluation 198
8.5 Comparison to Scribe 200
8.5.1 Notation 201
8.5.2 Wizards 201
8.5.3 Data Dictionary 201
8.5.4 Error Checking 201
8.5.5 Help 202
8.5.6 Database Assistance 202
8.5.7 Modular 202
8.5.8 Code Folding 202
8.5.9 Serialization 202
8.5.10 Code Libraries (Snippets) 203
8.5.11 Technology 203
8.6 Summary 203
Chapter 9 - Conclusions and Future Work 204
9.1 Introduction 204
9.2 Thesis Contributions 204
9.2.1 Prism Software 204
9.2.2 Prism Customers (End-Users) 205
9.3 Conclusions and results 205
9.4 Current Limitations 206

9.4.1 Meta-Model 206
9.4.2 RWM Shell Host 206
9.5 Future Work/Enhancements........ 206
9.5.1 Expand and Refine the Meta-Model 206
9.5.2 Improve Visual Notation 207
9.5.3 Improve Model Layout Algorithms 207
9.5.4 Versioning 207
9.5.5 Edit Points 207
9.5.6 Import RWL Script 208
9.5.7 Wizards 209
9.5.8 Code Snippets 209
9.5.9 Debug Generated RWL Script 209
9.5.10 WYSIWYG Output Layout 209
9.6 Summary 210
References 211
Appendices I
Appendix A Report Writer Factsheet by (Prism Group, 2008) I
Appendix B Core RWL Constraints II
Appendix C Case Study 1 Screenshots III
Appendix D *ExpandCollapseBase* Hide/Show children X
Appendix E Compartment Child Ordering XI
Appendix F Automatic Layout of Child Shapes XII
Appendix G Survey Request Letter XIII
Appendix H Survey Participation Information Sheet XIV
Appendix I Consent Form XV
Appendix J Survey XVI

List of Figures

Figure 1: Prism modules and integration reproduced from (Prism Group, 2008)................................2
Figure 2: Prism (R & D) - Prism WIN MIS User - Customer relationship 3
Figure 3: Prism WIN and the Report Write Module 6
Figure 4: RWL code 7
Figure 5: Incremental Model reproduced from (Sorensen)..12
Figure 6: Agile development model reproduced from (Klein)..14
Figure 7: Abstraction layers of MDSD reproduced from (Brown, Conallen, & Tropeano, 2005) 15
Figure 8: The modeling spectrum reproduced from (Brown, Conallen, & Tropeano, 2005)...............16
Figure 9: MDSD and application development reproduced from (Stahl, Völter, Bettin, Haase, & Helsen, 2003) 17
Figure 10: AOM approach reproduced from (Balaguer & Yoder, 2001)..19
Figure 11: Meta-model → Model → Real world reproduced from (Stahl, Völter, Bettin, Haase, & Helsen, 2003) 20
Figure 12: OMG metalevels reproduced from (Object Management Group, 2008) 20
Figure 13: MOF specification excerpt reproduced from (Stahl, Völter, Bettin, Haase, & Helsen, 2003) (Object Management Group, 2008)...21
Figure 14: Formalism, static/dynamic semantics and conrete/abstract syntax reproduced from (Metzger, 2005) 22
Figure 15: Software factory with a DSL reproduced from (Microsoft, 2007) 23
Figure 16: Himalia model of a UI reproduced from (Himalia, 2006) 24
Figure 17: Anjuta, an IDE for GNOME reproduced from (Naba kumar, 2007) 24
Figure 18: Qt C++ GUI designer for Eclipse reproduced from (Alessandro, 2007) 25
Figure 19: An HTML page...25
Figure 20: SQL snippet 26
Figure 21: SQL snippet represented visually..26
Figure 22: Textual DSL..27
Figure 23: Textual DSL using host language...27
Figure 24: Textual DSL using XML 27
Figure 25: Microsoft DSL Tools overview reproduced from (Microsoft, 2007) 32
Figure 26: Microsoft DSL tool wizards 32
Figure 27: Microsoft DSL Tools graphical designer...33
Figure 28: T4 text template snippet...34
Figure 29: Marama architecture reproduced from (Grundy, Hosking, Huh, & Li)..............................34
Figure 30: Marama Meta-Model Designer reproduced from (Grundy, Hosking, Huh, & Li) 35
Figure 31: Marama Shape Designer reproduced from (Grundy, Hosking, Huh, & Li) 35
Figure 32: Marama View Designer reproduced from (Grundy, Hosking, Huh, & Li) 36
Figure 33: MetaEdit+ Diagram Editor reproduced from (MetaCase, 2008) 38
Figure 34: MetaEdit+ Matrix Editor reproduced from (MetaCase, 2008) 38
Figure 35: MetaEdit+ Table Editor reproduced from (MetaCase, 2008) 39
Figure 36: MetaEdit+ Browsers reproduced from (MetaCase, 2008) 39
Figure 37: BPMN editor designed using GMF reproduced from (Eclipse.org, 2008)...........................40
Figure 38: Scribe wizards 41

Figure 39: Scribe report partitioning 41
Figure 40: Scribe meta-data library 42
Figure 41: Scribe function library 42
Figure 42: Scribe code library 43
Figure 43: Scribe syntax colouring/completion 43
Figure 44: Scribe non-context sensitive auto completion 44
Figure 45: RWL hard constraint 47
Figure 46: Class Diagram Approach 49
Figure 47: RWM Shell Approach 50
Figure 48: Meta-model representing RWL 52
Figure 49: RWL soft constraint 53
Figure 50: RWL meta-model and its instantiated model 54
Figure 51: Difference in generated RWL 54
Figure 52: Prism meta-data section 55
Figure 53: UI example showing Prism meta-data information 56
Figure 54: Example transformation service 57
Figure 55: Tree view representation of RWL snippet 58
Figure 56: Example of a visualized RWL script 59
Figure 57: Example of an intuitive RWL concept 59
Figure 58: Shell constraint validation 60
Figure 59: Shell "lazy" constraint validation 60
Figure 60: Showing RWL program flow 61
Figure 61: RWL composite construct notation example 62
Figure 62: Show/Hide child constructs notation example 63
Figure 63: Intuitive RWL meta-model extension point 65
Figure 64: RWL model variations 66
Figure 65: Simple implementation of the RWL meta-model 67
Figure 66: Sample shell interface 68
Figure 67: Sample shell interface with error detection 69
Figure 68: Notation using stereotyping 70
Figure 69: Use case developer perspective 73
Figure 70: Use case end-user perspective 74
Figure 71: Class Diagram Approach: Architecture 86
Figure 72: RWM Shell Approach: Architecture 87
Figure 73: Class Diagram approach: Initial class diagram (meta-meta-model) 89
Figure 74: Built-in templates 90
Figure 75: Embedding relationship reproduced from (Microsoft, 2007) 91
Figure 76: Reference relationship 91
Figure 77: Class Diagram Approach: Class Diagram relationship constraint 92
Figure 78: Class Diagram Approach: Class Diagram override 92
Figure 79: Class Diagram Approach: Code generator 93
Figure 80: Class Diagram Approach: Mandatory fields design 93
Figure 81: Class diagram approach: Field editors design 94
Figure 82: Prism meta-data 95
Figure 83: Generated mapping from LINQ tool 96

Figure 84: Example code to query a given Prism meta-data view or table 96
Figure 85: Class Diagram Approach: Versioning 97
Figure 86: Class Diagram Approach: Meta-model explorer 97
Figure 87: Class Diagram Approach: Visual designer design 98
Figure 88: Class Diagram approach: Shell host validation method calls 99
Figure 89: RWM Shell Approach: Initial meta-model 100
Figure 90: RWM Shell Approach: Relationship constraint 101
Figure 91: RWM Shell Approach: Two phase code generator 101
Figure 92: RWM Shell Approach: Mandatory attributes to mark mandatory fields 102
Figure 93: RWM Shell Approach: Specifying field editors 103
Figure 94: RWM Shell Approach: Model explorer 104
Figure 95: RWM Shell Approach: Visual notation via shape mapping 104
Figure 96: RWM Shell Approach: Enabling validation 105
Figure 97: RWM Shell Approach: Program flow 106
Figure 98: RWM Shell Approach: Containment 106
Figure 99: RWM Shell Approach: Ordering 107
Figure 100: RWM Shell Approach: Show/Hide children 107
Figure 101: Class Diagram Approach: Automation for inherited elements before 108
Figure 102: Class Diagram Approach: Automation for inherited elements after 108
Figure 103: Class Diagram Approach: Method generator 109
Figure 104: RWM Shell Approach: Model hierarchy 109
Figure 105: Class Diagram Approach: WPF UI 110
Figure 106: RWM Shell Approach: Shell UI 111
Figure 107: Modern UI toolbar 112
Figure 108: Docked windows 112
Figure 109: WPF UI tooltips (Help) 113
Figure 110: RWM Shell Approach: Property grid 114
Figure 111: Allowed/Not allowed cursor icons 114
Figure 112: Executing text templates 115
Figure 113: DSL Solution: The DSL Project 116
Figure 114: DSL Solution: The DslPackage Project 117
Figure 115: Path syntax example 117
Figure 116: Toolbox creation example 118
Figure 117: Toolbox example 118
Figure 118: Model Explorer configuration 119
Figure 119: Example Model Explorer 119
Figure 120: Connection builder 120
Figure 121: External types 120
Figure 122: External type usage 121
Figure 123: Serialized model information 121
Figure 124: Serialized model layout information 121
Figure 125: Customizable serialization 122
Figure 126: Text template example model 122
Figure 127: Text template 122
Figure 128: Text template output 123

Figure 129: Class diagram common hierarchy 124
Figure 130: Layer support 124
Figure 131: ModelType inheritance 125
Figure 132: Enumerand support 125
Figure 133: EnumerandValue properties 125
Figure 134: Canvas swimlane support 126
Figure 135: Class Diagram Approach: Validation implementation 127
Figure 136: Class Diagram Approach: Validation results 127
Figure 137: Class Diagram Approach: Inheritance cycle 128
Figure 138: Class Diagram Approach: Self inheritance 128
Figure 139: Class Diagram Approach: Multiple inheritances 128
Figure 140: Class Diagram Approach: Custom field editor implementation 129
Figure 141: Class Diagram Approach: Operation signature custom editor 129
Figure 142: Class Diagram Approach: Shape definition 130
Figure 143: Class Diagram Approach: Shape definition variability 130
Figure 144: Class Diagram Approach: Connector definition 131
Figure 145: Class Diagram Approach: Custom context-menu 131
Figure 146: Class Diagram Approach: VSCT code snippet 132
Figure 147: Class Diagram Approach: Method automation example 133
Figure 148: Class Diagram Approach: Method automation helper, step 1 133
Figure 149: Class Diagram Approach: Method automation helper, step 2 134
Figure 150: Class Diagram Approach: Meta-meta-model toolbox 134
Figure 151: Class Diagram Approach: Meta-Model model explorer 135
Figure 152: Class Diagram Approach: IModelElement interface 136
Figure 153: Class Diagram Approach: ModelElementBase abstract class 137
Figure 154: Class Diagram Approach: CoreModel class 137
Figure 155: Class Diagram Approach: ControlLineModel class 138
Figure 156: Class Diagram Approach: Added helper classes 138
Figure 157: Class Diagram Approach: Meta-model 139
Figure 158: Class Diagram Approach: Meta-model code generation 141
Figure 159: Class Diagram Approach: INotifyPropertyChanged attribute on a ModelClass object 142
Figure 160: Class Diagram Approach: Automated method generator for CheckMandatoryFields 143
Figure 161: Class Diagram Approach: WPF UI architecture 144
Figure 162: Class Diagram Approach: UI toolbox population 145
Figure 163: Class Diagram Approach: UI Toolbox 145
Figure 164: Class Diagram Approach: Adding a model element to canvas 146
Figure 165: Class Diagram Approach: WPF UI validation check 147
Figure 166: Class Diagram Approach: WPF UI requesting validation of soft constraints 147
Figure 167: Class Diagram Approach: WPF UI validation results 147
Figure 168: Class Diagram Approach: Property Editing 148
Figure 169: Class Diagram Approach: Custom editor 148
Figure 170: Class Diagram Approach: Generated RWL 149
Figure 171: Shell Approach: Report sections in the RWL meta-model 150
Figure 172: Shell Approach: Report sections hierarchy 150
Figure 173: RWM Shell Approach: Named element 151

Figure 174: RWM Shell Approach: Common hierarchy 152
Figure 175: RWM Shell Approach: Program flow meta-model 153
Figure 176: RWM Shell Approach: Scan-Print-Column relationship 154
Figure 177: RWM Shell Approach: ColumnSelectionEditor custom editor 154
Figure 178: RWM Shell Approach: Scan-Print-Column instantiated 155
Figure 179: RWM Shell Approach: ColumnSelectionEditor for view RM 155
Figure 180: RWM Shell Approach: Implementation of containment/ordering and hiding of child elements 156
Figure 181: RWM Shell Approach: Relationship ordering 157
Figure 182: RWM Shell Approach: Model constraints 158
Figure 183: RWM Shell Approach: Multiple reference constraints 159
Figure 184: RWM Shell Approach: Instantiated RWM model example 160
Figure 185: RWM Shell Approach: Toolbox/Explorer view 161
Figure 186: RWM Shell Approach: RWL script generation 162
Figure 187: Scenario 1 – Initial RWL Model 167
Figure 188: Scenario 1 – Partial RWM Model with connectors 168
Figure 189: Scenario 1 - Joining two Scan model elements 169
Figure 190: Scenario 1 - Complete RWL Model 170
Figure 191: ColumnReferencesFunction relationship 172
Figure 192: Embedding Select within the ControlLine 172
Figure 193: Select references columns relationship 173
Figure 194: Select relationship hierarchy 173
Figure 195: Select model element shape 174
Figure 196: Select model element shape definition 174
Figure 197: Select toolbox item definition 175
Figure 198: Select toolbox item 175
Figure 199: Various RWL function types 176
Figure 200: Select-Column functions annotated using attributes 177
Figure 201: Maximum level of abstraction 180
Figure 202: *ControlLine* construct minimum level of abstraction 181
Figure 203: *Scan* construct minimum level of abstraction 181
Figure 204: Illegal RWL drag and drop 184
Figure 205: Dependencies within the RWL notation 185
Figure 206: Variable instantiation/usage hidden dependency 186
Figure 207: Graph of solution technology familiarity 189
Figure 208: Graph of solution technology proficiency and meta-model perception rating after tutorial 190
Figure 209: Developers proficiency with RWL 190
Figure 210: Current report design (IDE) tool usage 194
Figure 211: Current report design (IDE) tool satisfaction 194
Figure 212: End-users familiarity with the new tool after guided demonstration 195
Figure 213: End-user rating visual notation compared to textual representation and its help in terms of the Prism DB 196
Figure 214: Edit points within RWL 208
Figure 215: Sample architecture of RWL Script to Model Handler 208

List of Tables

Table 1: Microsoft DSL Tools Vs Marama 36
Table 2: Microsoft DSL Tools vs. Eclipse GMF.. 40
Table 3: Class Diagram Approach: Code generation mapping.. 140
Table 4: Class Diagram Approach vs. RWM Shell Approach 164
Table 5: Developers familiarity with solution technology 188
Table 6: Developers proficiency with solution technology and meta-model perception rating after tutorial 189
Table 7: Cognitive dimensions convergence table 198
Table 8: Scribe vs. Our Solution 200

Chapter 1 - Introduction

1.1 Introduction

Enterprise tasks and processes tend to be complicated and require a fair amount of end-user training and support. Writing reporting scripts to query databases and display the extracted information is a good example of this. Industrial databases are complicated and hold huge amounts of data, the majority of which will never be relevant to the user at hand. The databases also tend to be unfriendly towards direct data manipulation for end-users in terms of how the tables are named and implicit relationships and cascading associations.

These complications play a big part while writing queries, report scripts and other general data extraction tasks. Thus the database generally proves to be the bottleneck of the system and to some extent also the bottleneck for user interaction.

If we could give the user the ability to easily access data and support them to think of the database and their reporting tasks in terms of the domain they are familiar with we would potentially increase the throughput of the data and information coming in and out of the enterprise system. This would have the additional benefit of improving end-user interaction and aiding their business. This is in short the goal of this thesis.

The rest of the chapter gives a brief introduction to the company for which this research project was carried out with a detailed introduction to the core problem, closely followed by the thesis motivation and ends with an overview of the thesis chapters.

1.2 Introduction to Prism

1.2.1 Core Business

Prism Group offers a fully integrated and configurable business Management Information System (MIS) for the printing and graphics arts industry. This includes the graphics and arts industry and has integrated modules for estimating, inventory, production management, shop floor management, cashbook, general ledger, accounts payable/receivable, sales order processing, facilities management and sales management. This is shown in Figure 1.

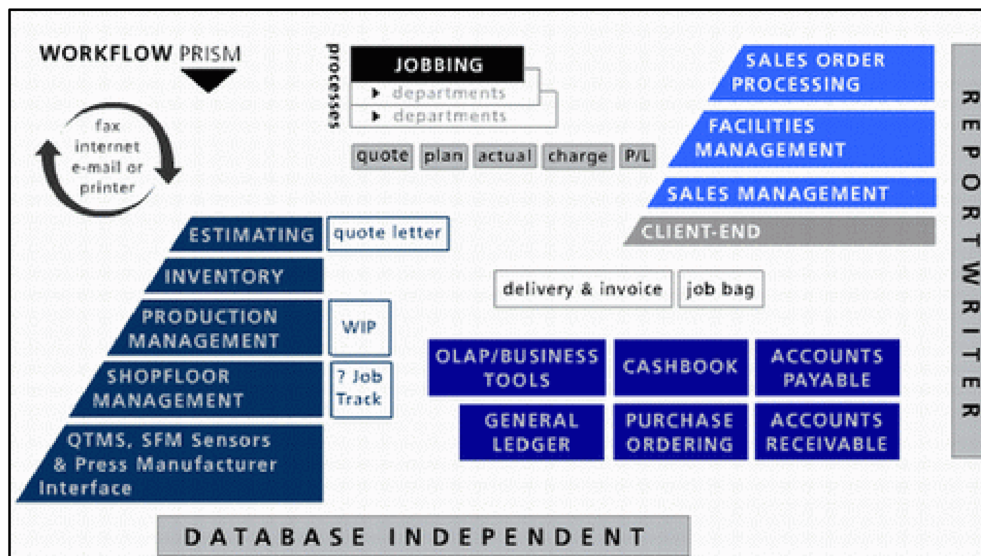


Figure 1: Prism modules and integration reproduced from (Prism Group, 2008)

Prism group has three main products: Prism WIN, Prism QTMS and Prism WIN SBE. Prism WIN is the flagship product which encapsulates all of the above modules (and other secondary modules totalling 20 altogether), although the user can pick and choose which modules they would like included. Prism QTMS (Quoting and Time Management System) provides a system which monitors and collects diverse information on machines operating on the shop floor in a print shop. Prism WIN SBE is the Small Business Edition of the flagship product in which users can select from a range of 14 modules which are designed to provide management and control for most printing processes and functions.

Prism has two development centres based in New Zealand and the United Kingdom and has three other business units in Australia, South Africa and the United States of America. The New Zealand development centre is primarily focussed on working on the flagship product, Prism WIN, whereas the development in the United Kingdom is centred on the QTMS product.

1.2.2 Prism WIN

Prism Group provides a heavyweight system called Prism WIN. This MIS comprises of 20 modules which can be enabled as required to meet the customers' needs. All modules integrate seamlessly with other Prism products such as the QTMS. Prism WIN fully integrates all processes of the print trade and allows easy management and control for management and shop floor staff (Prism Group, 2008).

Prism WIN is a Windows-based system written using Microsoft's .NET technology and can run on either Microsoft SQL Server or Oracle as its backend database. In the next section we briefly look at some of the core modules which Prism WIN is comprised of as we elaborate on the business domain this thesis is based around.

1.2.3 Business Domain

The domain we are concerned about during the course of the thesis will be centred on the Prism WIN application and its interaction with the printing industry. Below is a description of those domains with respect to Prism WIN. These domains are reflected in the modules catered for by the Prism WIN system. The modules outlined in the following subsections are only a subset of all the

modules offered by Prism WIN. The facts and information portrayed in the following sections is extracted from (Prism Group, 2008).

Each aspect of Prism WIN and its included modules are fully integrated with each other. To assist readers, Figure 2 demonstrates what the following sections refer to as the Prism WIN user and the customer relationship. It can be seen from the diagram that Prism (R & D) designs and implements the Prism WIN MIS which is used by a printing plant which in turn offers printing capabilities to customers. A Prism WIN MIS consumer (printing plant) consists of several departments; a subset of these departments is portrayed in the figure below.

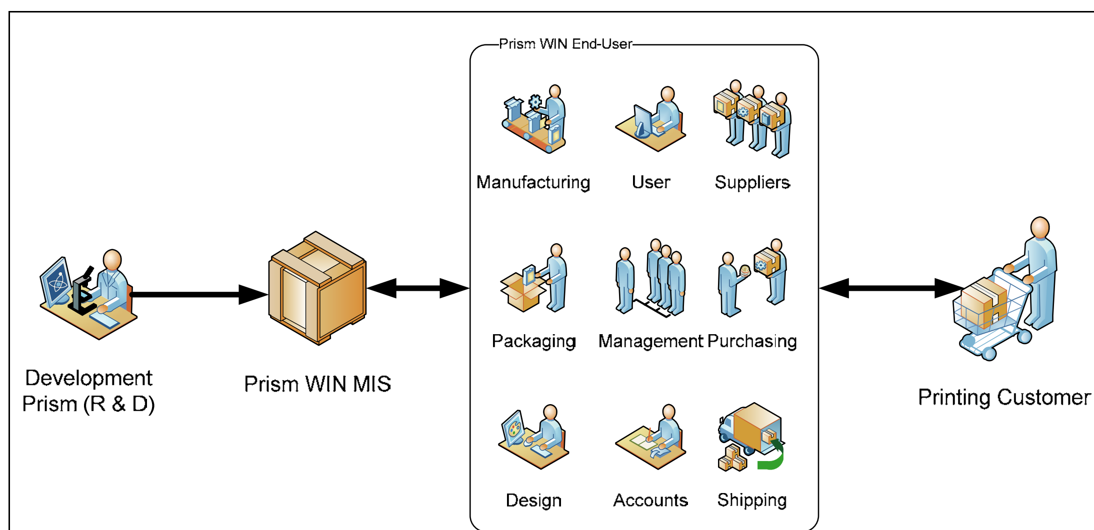


Figure 2: Prism (R & D) - Prism WIN MIS User - Customer relationship

1.2.3.1 Finance

The financial aspect of Prism WIN encapsulates four main modules. These modules are briefly described below:

- Accounts Receivable:
  This mainly contains cash flow into the system in the form of invoices, receipts, statements and historic trial balances. Other facilities include tax return assistance and customer maintenance.
- Accounts Payable:
  The AP module contains information about the cash flow out of the system, in the form of supplier invoices, transactions and balances.
- Cashbook:
  This module gives the Prism WIN MIS user control of their bank balances by tracking payments and receipts and reconciling them to the actual physical bank statement.
- General Ledger:
  General Ledger is responsible to encapsulate all the above modules (AP, AR and CB). The GL module assists management by providing reports on balance sheets, revenue statements, trading account and trial balance.

1.2.3.2 Inventory

The inventory modules of Prism WIN ensure that the MIS user always has the right and correct amount of stock at hand before starting a print job. It also assists the user to reduce inventory levels thus reducing costs. This is done by storing extensive information about stocks and their suppliers. The inventory also offers storing price and quantity breaks which allow the user to see when suppliers are offering stocks at low prices and when various quantities amount to a difference in price. This improves profit margins and giving MIS users and their customers' value for their money.

1.2.3.3 Quoting

The QM module is designed for simplicity which allows estimators and sales personnel to respond to customers request promptly and efficiently. Customised quote templates and screens allow for easy information entry and minimises the response time. Quotes can also be copied from existing jobs or other pre-existing quotes, quotes can also be combined into a new quote thus saving time and increasing the accuracy of the estimates and safeguarding the bottom line of the company. The QM module can also automatically assign quote numbers according to a user specified formula.

1.2.3.4 Job Costing

Job costing module is at the centre of the Prism WIN MIS system and ensures that each job is produced exactly to specification. It also records vital statistics which are generated during a job run which provides the user control over various cost centres during the printing process. The job costing module is very flexible and is fully configurable for the user's needs.

1.2.3.5 Production Management

The PM module contains a set of tools which allows the end user to maximise resource utilisation and production efficiency. The centre piece of the PM module is the Gantt chart which allows the end user to visualise where each job is scheduled to run. Through this chart schedulers and planners can plan and effectively utilise a resource to its maximum capabilities, thus maximising profits. The PM module also has information about the working time of machines (resources) and can calculate the projected finish date and time of a job at runtime, by doing so the PM module can alert users if a potential deadline is likely to be exceeded. This allows the user to make appropriate changes to the plan before any problems escalate.

1.2.3.6 Purchasing

The purchasing module allows quotes to be transferred to purchase orders which can directly be added to inventory to fulfil stock requirements. The purchasing module also regulates inventory levels to eliminate excessive stock ordering and maintains stock at its most efficient level. It also provides a mechanism in which staff members who do not have any purchasing rights to raise purchase requests which can then be converted to a purchase order.

1.2.3.7 Sales Management

The SM module allows frictionless functioning between the sales process and customer and prospect relationships. It helps schedule various sales representatives between contacts, sales and prospective customers. Any interaction can also be recorded via the SM module. The SM module also allows easy transfer of prospect customer status to a full-fledged customer after a contract has been won. This change then propagates throughout the entire Prism WIN system. The SM module user interface is flexible and allows the management to drive the collection and retention of required information about prospective customers.

1.2.3.8 Sales Order Processing

The SO module gives the MIS user fully automated control over the entire print process for a job from the inventory through to the delivery of finished goods to the customer. The module has built-in validation and audit mechanisms which allow the Prism WIN MIS user to keep the end user informed at all stages about stock levels and delivery dates. Back orders can also be placed along with automatic or repeat forward orders for recurring stock. Priorities can also be assigned to stock via the SO module. The SO module also allows the user to set up a structured and elegant pricing and discount system for customers. Sales reports and other marketing campaign results can also be generated via the SO module which gives businesses the edge and allows them to improve their bottom line and maximises profits.

1.2.3.9 Shop Floor Management

The SFM module provides a process in which data is electronically collected from the shop floor to feed into the rest of the Prism WIN MIS system to allow real-time tracking and monitoring. The core of the SFM module is the Virtual Time Manager (VTM) which tracks a creative or production workers time and also captures what task their time was spent on. The SFM module can also track machine and material usage, waste, time taken for chargeable and non-chargeable work. This data is then recorded against the job it was running against thus keeping the MIS user fully informed about its status and statistics. Monitoring and recording of actual data posted on running jobs allows the user to give better estimations at the quoting stage of the job thus improving productivity and giving more accurate estimates to customers.

1.2.3.10 Report Writer (RW)

This thesis is based on the report writer. The RW language (RWL) is a powerful interpreted language designed by Prism engineers to allow Prism WIN users to write powerful data mining queries. Via the RW module users can extract data and use Prism WIN's Meta-Model to combine data from multiple database tables to give a detailed report with a configurable layout. The RW module and language also has the ability to invoke action scripts[1] within Prism WIN. As this is an important sub-section of this thesis I have included the actual Report Writer Factsheet from (Prism Group, 2008) with this document as Appendix A.

This subsection is the core of the thesis and we explore further into the module and its corresponding RW language in the following sections, Section 1.3 and Section 1.4 respectively.

1.3 Report Writer Module

Prism WIN's Report Writer (RW) module is the interface between the database and the printing plants' (end-user) management team. The smooth flow of data in and out of a system is the primary requirement for any successful business; Prism WIN caters for this by exposing key areas of the database via the RW module.

End-users can simply write a set of commands (formally known as the Report Writer Language, RWL) and execute these against the Prism WIN system. The system interprets these commands and if need be, queries the database to retrieve the information which the user requested. The module

[1] Action scripts are similar to macros. They allow the user to define a set of legal actions within Prism WIN which can be run automatically.

and the RWL are flexible and gives users complete control on not only what information they want extracted but also on how they want this information to be displayed visually.

The figure below demonstrates the above mentioned concepts and shows how the WIN system can simply import the report script and then utilise it to query the database and give user the required information in a desired visual form (printer/screen/file).

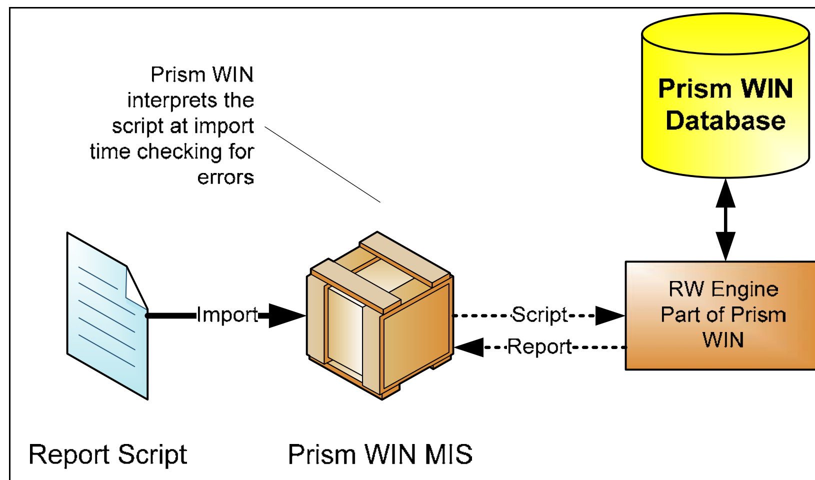


Figure 3: Prism WIN and the Report Write Module

Furthermore the RW module is fully integrated with WIN and is accessible system wide. This allows the user to invoke action scripts[1] via the RW. In the following section we look at the syntax of the RWL and also go through a simple example to give us a feel of how the language works.

1.4 Report Writer Language (Prism New Zealand, 2005)

The Report Writer Language (RWL) is complementary to the RW module. RWL is an interpreted procedural programming language and to some extent, also object oriented. It is intuitive but like any programming language takes a fair amount of learning and experience. Below we see a small RWL example script.

```
1.   Code          RW_EXAMPLE
2.   Name          “Report Writer Example”
3.   Type          Standard
4.   Access        STSR
5.   Select        RM_CUST.Range + RM_BAL_TYPE.Match
6.
7.   Clump         cust_clump = RM_CUST + RM_NAME + RM_BAL_TYPE + RM_BAL_OWE.SetNDP(2);
8.   Clump         tran_clump = RT_DATE + RT_TRAN_TYPE + RT_OUR_REF + RT_INCL.SetNDP(2);
9.
10.  PageHeader
11.                Print StandardPageHeader;
12.                Print cust_clump.ColDesc;
13.                Print;
14.  End
15.
16.  Scan RM
17.                Print cust_clump;
18.                Scan RT Choose(RT_CUST, Match, RM_CUST)
19.                            Print tran_clump AtCol(10);
20.                End
21.                Print;
22.  End
23.
24.  Print StandardReportFooter;
```

Figure 4: RWL code

Figure 4 shows some of the key points of the RWL; it demonstrates a subset of functions such as *SetNDP*, *ColDesc*, *Choose* etc. It can be seen that these functions are executed on objects such as a *Clump*, *Scan*, *Variable* etc. In the following sub-section (Section 1.4.1) we go through a simple RWL example to give us a better understanding on how the language works.

1.4.1 RWL Walkthrough

Let us systematically look through RWL example introduced in Figure 4. This example is not the simplest example available but gives us a brief introduction into the important concepts we need to understand the rest of the thesis. Minute irrelevant details will not be examined.

The first five lines (Line 1 – 5) of the report contain the header definition. This is how the Prism WIN system maintains this report and allows users to find it and execute it. This header information is called the *ControlLine* structure of the report.

The next two lines (Line 7 – 8) declare two variables, in this example the variables are of type *Clump*. Clumps are a grouping of arbitrary objects like database columns, other variables or numbers. In this example the clump contains a list of database columns.

The next section (Line 10 – 14) contains the page header. The page header is just a block of code which can hold other RWL statements. As the name suggests, the page header will be shown on each page of the report (if the report spans more than one page).

Lines 16 – 22 show the most vital part of the RWL language, a *Scan* object. A scan object is inherently a loop, although it loops through a Prism database view. In this example it loops through the RM view and also has an inner scan which loops through the RT view. RT and RM are joined on the condition *RM_CUST == RT_CUST*, this is done by the *Choose* statement. In short, a Scan is to RWL what a *Select* statement is to Structured Query Language (SQL). Therefore, nested scans are nothing but an inner join defined by the *choose* condition.

The last line is the report footer.

Other secondary objects of concern in this particular script are functions such as *AtCol*, *ColDesc* and *SetNDP*. Specific functions can be executed on specific types of objects.

1.5 Introduction to the Problem Domain

This thesis deals with two core problems which are at the heart of improving Prism productivity and product quality, namely the complexity of the Prism database and the complexity of the Report Writer language. The following sub-sections look more closely at these two core problems.

1.5.1 Database Complexity

An enterprise database is the heart and soul of a business and it is fair to assume that they are usually very complex and require constant maintenance and user training. Extracting information from the database is equally important as putting information into it. Due to this reason, an interface to the database should be as user friendly as possible.

The Prism database is very complex and has approximately 350 tables and views. The Prism database initially started on a Velocis server which did not cater particularly well for implicit

referential integrity checks. The naming scheme followed business rules explained further into this sub-section. Although the Prism system moved apace along with new technology and so did the database, the way the database was designed never changed and started lagging behind. Due to this we have the following issues:

1. *Foreign key constraints*
   The Prism database does not have foreign key constraints and due to this, integrity checks have to be done manually.
2. *Meta-Model*
   The Prism database maintains its own meta-model. Some of the information held by the meta-model includes table relationships, number of columns, data types, etc. Therefore, the meta-model is not intuitive and needs users to have in-depth training and at times some assistance.
3. *Naming conventions*
   Prism database tables and views are named using a two-part convention. For example, in the RWL Walkthrough (Section 1.4.1) we saw two views, RM and RT. Both these views belong to the AR (Accounts Receivable) module. RM is the customer master record and RT holds the transactions. Details about these views are not relevant at this stage. Although notice that these names are not intuitive and will most likely require a user to know comprehensive information about the database. Also note that all columns within a given table are prefixed with the table name as its first two characters, for e.g. view RM (view on table RM) will only contain columns like RM_*.
4. *Views split over multiple tables*
   View RM and RT were introduced in Section 1.4.1 and further developed in the previous paragraph. Note that both of these views have underlying database tables. Each view may have more than one underlying table. This came about as some tables have over 250 columns; this is not allowed as part of the database schema in some database systems. Therefore, the table was split. For example, the RT view has two underlying tables, RTI1 and RTI2. Some views are split over as many as five tables. This makes database updates complicated as it is not always clear in which split section the updated column resides in.

Due to the above mentioned issues a novice end user of the Prism database will thus have many challenges when trying to understand how to use the tables and their relationships.

1.5.2 Report Writer Language Complexity

Like any programming language there is an obvious learning curve with the Prism RWL. As the RWL is an interpreted language, it is very easy for a user to make an error and not pick it up until the report is imported by the Prism WIN system or worse, when the report is actually executed.

The construct which is at the heart of the RWL (*Scan*) is database dependant, thus all the problems detailed in the preceding sub-section will be translated and added to the RWL complexity.

Functions in the RWL are also data-type specific thus a user needs to know what data-type a particular column in a database is to perform any data manipulation on it.

Thus, in order for a user to write an effective Prism report, they need to have intimate knowledge on how the database is assimilated, what foreign key to do joins on (e.g. *RM_CUST==RT_CUST*), what functions are available and what data-types the RWL exposes.

Moreover, there is no dedicated IDE (Integrated Development Environment) for the RWL. This statement is not entirely true as there is an in-house IDE called *Scribe* although it is not as sophisticated as one would expect. We will be looking further into it in Section 2.6.

Thus users do not have a high level support tool for designing, implementing and testing Prism RWL reports on top of the very complex Prism database.

1.6 Motivation

The motivation for the thesis was to research and develop a model-driven approach to systematic database object manipulation in terms of the printing reporting domain, thus I decided that if a framework could be developed which modeled part of the database which was used by the RW module we could possibly achieve two key goals. These two goals are the main motivation behind this research and are explained in the following sub-sections:

1.6.1 Reduce database complexity

Personally, I have always found that it makes it simpler to understand the workings of a particular system if I know how the database integrates together. Due to this reason, the initial motivation was to make the Prism database user-friendly. Thus, reducing the database complexity to some level or at least show that it is possible was the main motivation for this thesis.

1.6.2 Intuitive RWL tool

The repercussion of the above motivation was to show that the framework (model) developed to reduce database complexity does actually assist software development and in-turn make systems intuitive. This motivated me to design a small scale tool to allow users to design reports using the RWL which would utilise the framework designed to reduce database complexity.

1.7 Thesis Ambition

The main goal of the research done was to reduce database complexity as mentioned in the preceding section and also to develop a tool which demonstrates the newly improved database model. We aim to design a model driven Prism Report Writer Language tool which allows users to visually create a Prism report using a suitable visual language and support tool. This will allow the user to work in terms of the domain they are familiar with to perform a task they initially had difficulty with. The domain in this instance is the Prism Report Writer and the Prism database and the task (enterprise process) is writing a Prism report.

We achieve these goals by using a model-driven software development (MDSD) approach coupled with the use of Domain Specific Languages (DSL). The end user can construct reports for the RWL using visual metaphors for both report components and database elements. The support tool will check for consistency errors and provide the user a database browser and set of available reporting elements in a structured and context sensitive way.

The MDSD approach enables us, as system developers, to enjoy some of the inherent advantages. Basic advantages such as increased development speed thus reducing the time to market for

products, increased software quality and most important of all, improving the manageability of complex enterprise tasks and processes through abstraction (Stahl, Völter, Bettin, Haase, & Helsen, 2003).

The DSL approach allows us to encapsulate and represent complex enterprise tasks and processes in terms of a particular domain, in this case, a domain the end-user would be familiar with.

1.8 Thesis Overview

Chapter 1. *Introduction*
Gives an introduction to the thesis and a background into Prism Group and the printing industry. Also gives an introduction to the problem domain and describes the motivation behind the research

Chapter 2. *Related Work*
A detailed outline of the work and technologies researched to assist the thesis in terms of requirements gathering, development, testing and evaluation.

Chapter 3. *Our Approach*
Outlines the approach taken to arrive at the solution. Also gives a brief introduction to the succeeding three chapters.

Chapter 4. *Requirements*
Lists all the requirements the solution described in this thesis caters for. The solutions are grouped by functional and non-functional requirements and further categorized by their target stakeholder, either the developer or the end-user. The developer is mainly concerned with the meta-model of the RWL where as the end-user is concerned with the shell hosting this meta-model assisting them to design RWL models.

Chapter 5. *Design*
The design chapter details all the design decisions taken with respect to our two approaches, The Class Diagram Approach and The RWM Shell Approach. The chapter also gives us details about the stakeholders and gives us a graphical perspective of the requirements using use-cases. The chapter eventually details the architecture and gives details about the user interface design for our Class Diagram Approach.

Chapter 6. *Implementation*
Expands on the details of the implementation for each of our approaches: Class Diagram Approach and the RWM Shell Approach.

Chapter 7. *Case Study*
Incorporates three case studies or tutorials which take us through various aspects of the solution. Primary case study has a tutorial on how the solution can be used by the end-user to visually design a RWL script followed by a couple of case studies on how a developer can expand the given meta-model if need be.

Chapter 8. *Evaluation*
This chapter evaluates our core approach: RWM Shell Approach (2nd Approach). It highlights the evaluation process during the design phase using cognitive dimensions and also through an end-user perspective. The meta-model of the RWL developed as part of the solution is also evaluated by developers to measure its expressiveness.

The chapter concludes by comparing the designed solution with an existing Prism reporting tool called Scribe.

Chapter 9. *Conclusions and Future Work*
This chapter concludes the thesis by giving us a list of the contributions made by our research in terms of the developer and the end-user. It also lists current limitations in terms of the technology and the solution and gives a set of future enhancements that can be made to make the solution better.

1.9 Summary

This chapter gave us an introduction to the thesis and the research involved. It also briefly introduced us to the company this research is based around and gave a detailed introduction to the problem. The problem introduced here was the inability for end-users to be able to carry out an enterprise level task such as report writing due to the complexity of the Prism database and the reporting language. Therefore out motivation for our research was to provide a tool which could abstract some of these details from the user and allow them to design reports using a higher level model of the database and the reporting language.

Chapter 2 - Related Work

2.1 Introduction

This chapter gives an introduction to the background research done in order to complete this thesis. Research work in the fields of software development methodologies, model-driven software development/engineering, and current state of art of domain specific and visual languages, DSL Tools and existing reporting tools was examined. The findings for each of these are outlined in the remainder of this chapter.

2.2 Development Methodologies

This sub-section describes two software development methodologies researched which assists the development of the solution for the thesis. As we are developing a model based solution it was important to study how it varies from and in some ways similar to traditional development methodologies. We examine two orthodox approaches and in the following sub-section examine the approach taken.

2.2.1 Incremental Model

Incremental software development methodology is described as the process in which each small section of the system is independently designed, implemented and tested (Sorensen) (Shown in Figure 5). We looked at this approach as it fits in nicely with the thesis as we planned to develop the model and the tool using an iterative approach where we add more detail and complexity while moving forward.

Due to the time constraints it was nearly impossible to model the entire Prism database process and all constructs of the RWL. Thus, we applied an incremental approach to the problem at hand, in which we extract small aspects of the database and the RWL and model them.

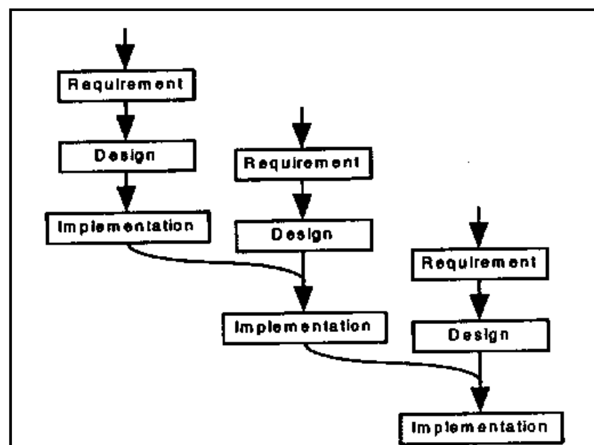


Figure 5: Incremental Model reproduced from (Sorensen)

The main drawback of the incremental model is the integration aspect in which different sections of the system have to be integrated together to make the full system. We would not have this problem during the course of the thesis as we would only build on the existing or previous builds as the implementation was only done by a single person. Although care had to be taken that this does not develop into a problem in the future if the development of the product is done on a larger scale.

2.2.2 Agile Development Model

Information for this sub-section is extracted from (Subramaniam & Hunt, 2006).

This thesis relies heavily on the work done and the ideas based around the agile development model. Agile software development is ideally defined as a process which uses feedback and analytical results to constantly adjust the development in a team environment. The approach taken by the thesis is very similar to some of the principles which are the heart of the agile software development model.

We designed a tool to assist users in enterprise level tasks, thus the first aspect of the agile development used was the need for constant end user (customer) feedback. The thesis also aimed at giving the development team a tool which was designed using models to reduce the time to market new features and increase software quality. Thus, the feedback principle in this case is two-folds in terms of the thesis: feedback from the end-users and feedback from the developers.

In terms of users, agile development stresses that software, no matter how well written will only be useful as the users perceive it. It describes ways in which to involve users throughout the development lifecycle by allowing them to make decisions, informing them about what technology was used and why, getting frequent feedback using demonstrations and the other obvious pointers such as prototype evaluations and beta releases.

In terms of implementation, agile development model stresses the need for understandable and simple code. It promotes the need to write cohesive code which avoids deep coupling.

Agile development promotes the idea of development in small increments, although this is explained in the previous sub-section (Section 2.2.1) where we examined the incremental model of software development.

Although the methodology used by the thesis strongly resembles that of the agile model, it is important to point out the differences. This thesis, as it stands currently, is not a team project therefore the aspects of collaboration have not been targeted.

The figure below shows concisely the above mentioned principles and is in effect the software development process used for the course of the thesis.

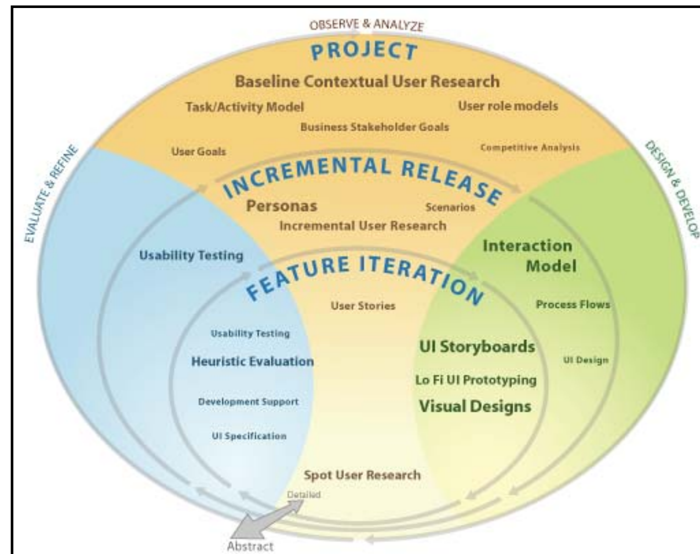


Figure 6: Agile development model reproduced from (Klein)

2.3 Model-Driven Software Development

2.3.1 Introduction

The core of the thesis was to build a model of a reporting language to assist developers add new functionality and in turn assist the end-users in using this new functionality. Due to this factor, it was important to research what model-driven software development (MDSD) is all about and the current state of art.

As described in Section 1.7 MDSD has several advantages over traditional software development methodologies. MDSD reduces development time by allowing for automation of code generation from formally defined models (Beydeda, Book, & Gruhn, 2005). It also improves software quality as a model once designed will maintain a consistent architecture and present that in its implementation (Beydeda, Book, & Gruhn, 2005). MDSD makes software maintainability easy; a change can be made in the code generation algorithm which will propagate in all parts of the implemented system (Beydeda, Book, & Gruhn, 2005). Code generation libraries and patterns can be reused assisting developers and reducing redundant code (Stahl, Völter, Bettin, Haase, & Helsen, 2003) (Beydeda, Book, & Gruhn, 2005). Most importantly, MDSD allow the developers to design high level objects abstracting the irrelevant details away from the core problem. Due to this reason models are best described in a language suitable for a specific problem, this is where Domain Specific Languages (Section 2.4) (DSLs) play an important role (Stahl, Völter, Bettin, Haase, & Helsen, 2003).

The following sub-sections outline some of the research done in individual areas of interest during the course of this thesis.

2.3.2 MDSD Approach and Terminology

2.3.2.1 Terminology

Let us start by looking at the four core keywords in MDSD: CIM, PIM, PSM and ISM (Code), shown below in Figure 7.

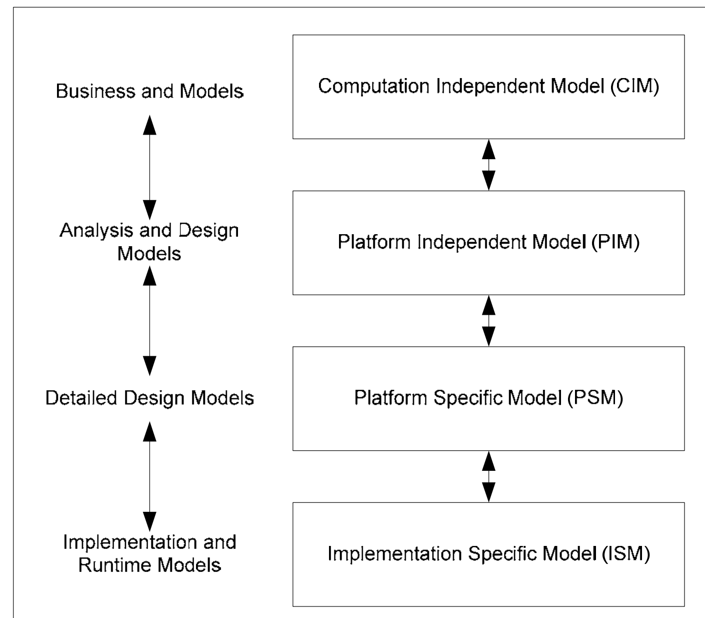


Figure 7: Abstraction layers of MDSD reproduced from (Brown, Conallen, & Tropeano, 2005)

The definitions of the above principles are extracted from (Afonso, Vogel, & Teixeira, 2006):

- *CIM (Computation Independent Model):* Concentrates on modeling the business aspects, i.e. the domain specific nature of the application.

  The specification of the RWL we are concerned with during the course of this thesis is an example of a CIM. It does not involve any details about technology or implementation and only specifies business constraints and semantics.

- *PIM (Platform Independent Model):* Shows a model which sits at an abstraction level which excludes any details on the platform on which it depends.

  Going from this RWL specification and representing in a static UML structure is an example of a PIM. The process via which we go from the business rules (CIM) to UML (PIM) is detailed in Section 5.4.

- *PSM (Platform specific Model):* A model which encapsulates details from the PIM into platform specific implementation constructs.

  After the UML (PIM) is established we then translated this into our DSL Tools. Therefore the model represented via the DSL Tools is essentially a PSM. This is not a “true” PSM as we are using Microsoft DSL Tools are we are bound by the platform constraints imposed by the Microsoft CLR Framework.

- *ISM (Implementation Specific Model or Code):* Final level of the MDSD transformation process containing executable code.

  The Microsoft DSL Tools generate executable code from the model designed (PIM). This

code essentially is the ISM. This executable code is what is used by end-users to instantiate a given model.

2.3.2.2 Approach

Moving in a tangential aspect from the terminology described before, let us look at different degrees of MDSD. Paper (Brown, Conallen, & Tropeano, 2005) from (Beydeda, Book, & Gruhn, 2005) describes this and shows how a model and code can co-exist. These are show in the figure below:

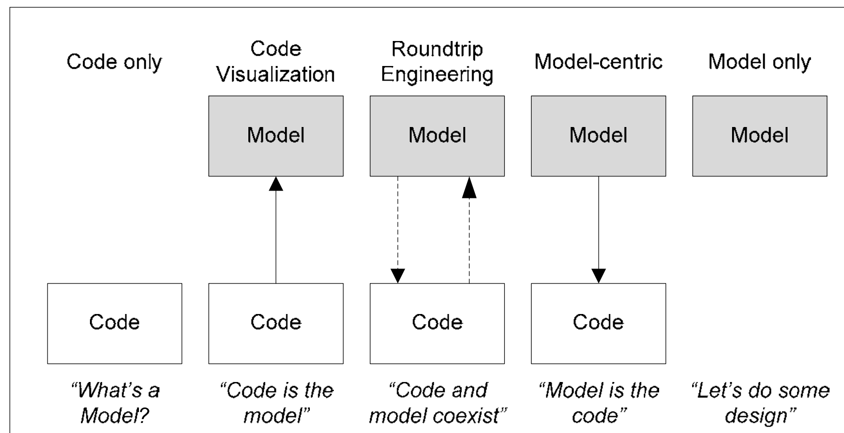


Figure 8: The modeling spectrum reproduced from (Brown, Conallen, & Tropeano, 2005)

Let us use Figure 8 to further explain some common modeling techniques. (Brown, Conallen, & Tropeano, 2005) claim that majority of the software systems in existence currently have no notion of model as such (code-centric). They have code which encapsulates business logic and constraints and will commonly represent this directly in a third-generation programming language (3GL). (Brown, Conallen, & Tropeano, 2005) rightly points that this is not entirely true as a 3GL is inherently a model sitting on top of the machine language which abstracts out the details for developer ease. Although for this discussion we will ignore this. In this scenario models may be exist but at best they are informal and usually are disparate from the code.

Moving one step further we look at tools which help us visualize application code. This is done by tools such as IBM WebSphere Studio, Borland Together/J and Visual Studio Class Designers (Brown, Conallen, & Tropeano, 2005). The code and visual representation are usually kept synchronized and give developers an alternative perspective on the code. This visualization can at best be defined as a "diagram" as opposed to a model as it is at the same abstraction level as the code it visualizes.

Roundtrip engineering (RTE) takes code visualization to the next level where developers design an abstract model and generate the underlying application code from it. Often the generation is manual but tools like IBM Rationale Rose allow automation. With manual generation any changes in the code need to be reconciled with the model and this brings us to the potential problem of the model and code going out of step (Brown, Conallen, & Tropeano, 2005).

In the next flavour, we have a model-centric approach in which the application code is fully generated from the model. In this case the model may have to represent information about the persistence, data access, visual representations and business logic. To aid the code generation process this approach often uses purpose based frameworks and languages which can be realized as a DSL. These purpose based frameworks thus constrain the code generated for an application thus

making the application purposed based too. For example IBM Rationale Rose has two separate products for two distinct domains, Technical Developer for real-time embedded system and Rapid Developer for enterprise IT systems (Brown, Conallen, & Tropeano, 2005).

This brings us to the right-hand side of the spectrum where a model-only approach is used. This is ideal for business logic communication, meetings and other high level architectural discussions.

The transformations (directional arrows) shown in Figure 8 are further researched and outlined in Section 2.3.5.

Research was also done on how MDSD relates to application development, Figure 9. This diagram can also be seen from a MDSD vs. orthodox application development perspective.

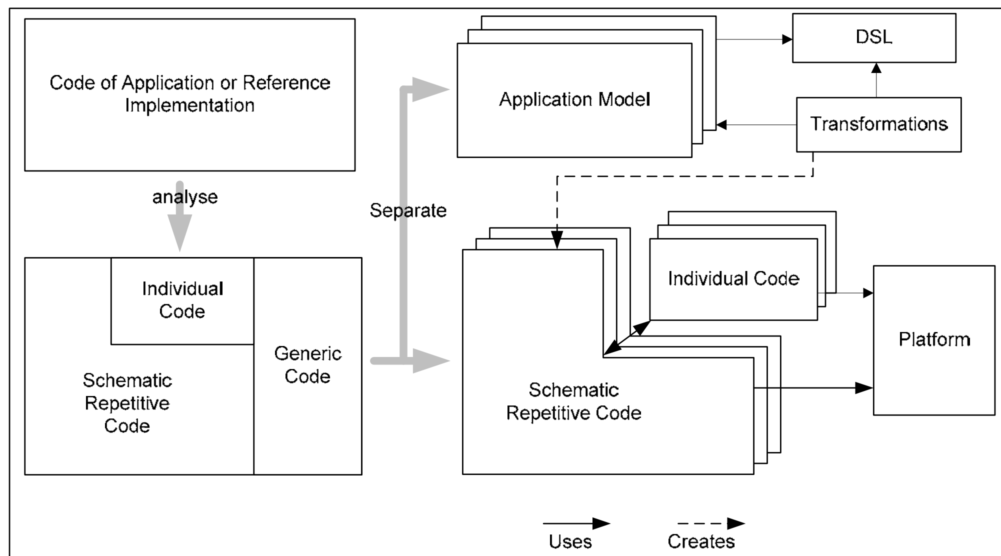


Figure 9: MDSD and application development reproduced from (Stahl, Völter, Bettin, Haase, & Helsen, 2003)

The top left corner of the diagram shows an application in its entirety. After analyzing the application we can virtually separate out three sections: a generic section which encapsulates all identical code throughout the application, a generic schematic section which encapsulates code which follows the same design patterns and a section which is application-specific and cannot be generalized. This brings us to the right hand part of the diagram, the MDSD approach. MDSD strives to develop the repetitive schematic section from the application model. Other aspects of MDSD are DSL and transformations which allow models to lend to other models which may eventually constitute the schematic section. Note that in MDSD, both, the schematic section and individual application-specific code section compose the platform (Stahl, Völter, Bettin, Haase, & Helsen, 2003).

A research article written by Afonso M. et al, (Afonso, Vogel, & Teixeira, 2006) follows on from the above section and describes what is required to traverse from code-centric approach to a model-centric approach. The findings from that paper are summarized in the remainder of this sub-section.

(Afonso, Vogel, & Teixeira, 2006) claims that the perceived value of MDSD within the organisation and actors within the organisation determines the success rate of the transition from a code-centric to a model-centric environment. It further explains that a change management system is mandatory

in order for the organisation to train the actors within it in terms of MDSD. The paper concludes with some strategies on how to make the flow from a code-centric environment to a model-centric environment successful. These strategies are outlined in the list below:

- *Guidance:* Leaders in the organisation should believe in the MDSD approach and provide guidance and feedback to lead the other actors within the organisation.
- *Incremental Approach:* As highlighted in Section 2.2.1, an incremental approach will allow the organisation to start in a place where using MDSD has immediate effects which would motivate the team.
- *Tool Selection:* Tool selection should be done in terms of not only cost but also on the learning curve it would imply and the features that would be vital to improve development in your organisation.
- *Modeling experience:* It is mandatory to involve actors with experience in MDSD from the preliminary phase and slowly train the remainder of the team.
- *Analytical and Business skills:* As MDSD does not only look at the implementation of a system but also its business aspects it is important that the team has strong analytical and business oriented skills.

2.3.3 Adaptive-Object modeling

To understand object modeling we look at a particular technique called Adaptive Object-Modeling. This is discussed in (Balaguer & Yoder, 2001) and the remainder of the sub-section relies heavily on the information from it. This sub-section is closely linked with the succeeding Metamodeling sub-section.

The paper has a concise quote by Ralph Johnson defining metadata. It states that, “If information is going to vary in a predictable way, store the description of the variation in a database so that it is easy to change” (reproduced from (Balaguer & Yoder, 2001)). So in short the Adaptive-Object Model (AOM) follows from that principle and is a model which provides enough information about itself so that it can be modified at runtime.

AOM solves some of the key modeling problems with a term called reflective-architecture. Some aspects of the reflective-architecture and the problems it strives to solve are briefly listed below Figure 10.

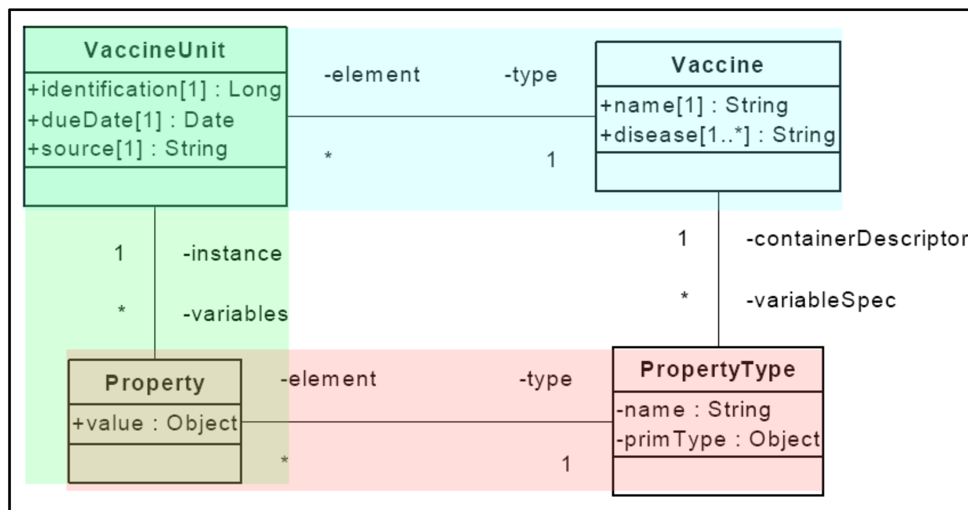


Figure 10: AOM approach reproduced from (Balaguer & Yoder, 2001)

- *TypeObject:* Multiple instances of each kind of relevant element in the domain. This allows a single kind to represent the actual object and each instance has variations on that object. Thus, if a new object variation has to be added it can be easily done without changing the database for example. Shown in Figure 10 by *Vaccine* and *VaccineUnit* relationship.
- *Properties:* Instances of a given object may have different attributes which vary at runtime. This is solved by having another object represent properties for a given instance of an object. Shown in Figure 10 by *VaccineUnit* and *Property* relationship.
- *TypeSquare:* Adding properties of different types to an existing system is cumbersome and will lead to inconsistency between instances linked to a particular object. This can be solved by having the TypeSquare which contains type information. Thus, each instance linked to a particular object will have the same properties and with matching types. Shown in Figure 10 by *Property* and *PropertyType* relationship.

The paper then follows on by listing some of the advantages and disadvantages of AOM. Some advantages are that the system can be easily adapted to a changing environment; changes do not require the entire system to be recompiled, business rules can be easily modified, time-to-market is reduced and increase in maintainability. Some disadvantages highlighted are that AOM has a steep learning curve and can be costly, requires skilled developers and may at times have poor performance. Note that the above advantages and disadvantages are analogous to the advantages and disadvantages of MDSD as a whole as described in (Afonso, Vogel, & Teixeira, 2006).

2.3.4 Metamodeling

Metamodeling is one of the core concepts of MDSD due to the following reasons (Stahl, Völter, Bettin, Haase, & Helsen, 2003):

- *Domain Specific Language:* A meta-model expresses a DSLs abstract syntax. The abstract syntax of a language details its structure as opposed to the concrete syntax which details what a specific language parser accepts.
- *Model Validation:* Metamodels make it possible for constraint definition, thus a model can be validated against its meta-model representation.

- *Transformations:* A model can be successfully transformed into another model by rules defined in its meta-model.
- *Code generation:* Code generation templates refer to the meta-model of a DSL.
- *Tool integration:* Modeling tools can be adapted to its corresponding domain using the metamodel.

As defined in the above list, the abstract syntax of a language is defined by its meta-model which in turn defines the interface between the model and various processing needs. As opposed to the concrete syntax which defines the interface to the modeler. Due to this separation a meta-model and the concrete syntaxes defining the model can maintain a 1:n relationship. This allows a meta-model to be recognized in both, textual and a visual form (Stahl, Völter, Bettin, Haase, & Helsen, 2003). Visual and textual DSLs are explored in Section 2.4.

Let us look at how a meta-model can relate to a model and in turn relate to the real world. This is shown in the figure below.

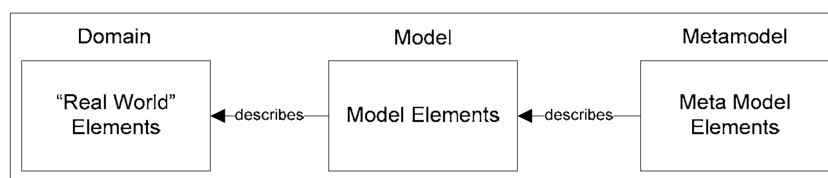


Figure 11: Meta-model → Model → Real world reproduced from (Stahl, Völter, Bettin, Haase, & Helsen, 2003)

The figure above shows how a meta-model describes the model which describes the elements defined in a given domain. Following on from the above diagram the Object Management Group (OMG) has a more technical diagram (Figure 12) which shows four metalevels describing the transition from the meta-model to the real world element.

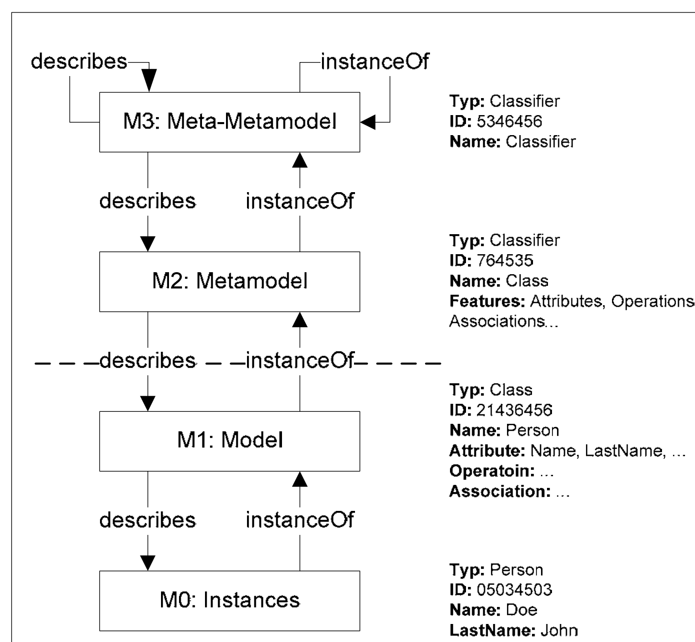


Figure 12: OMG metalevels reproduced from (Object Management Group, 2008)

Looking at the above diagram we can clearly see how an instance of an object is realized from a meta-meta-model. Let us step into familiar domain which is below the dotted line, this domain is where traditional applications are developed. The figure shows a *Class* of type *Person* in level M1, this is then instantiated in M0 which gives attributes defined in M1 specific values, in this example: Name: John; LastName: Doe. Moving up one level to M2, we can see how the type *Class* is defined using a type called *Classifier*, eventually leading to M3 which defines the type *Classifier*. M3 denotes the MOF (Meta Object Facility) layer. It can be quickly seen that this level hierarchy can go on ad infinitum, although the OMG simply states the MOF defines itself (Stahl, Völter, Bettin, Haase, & Helsen, 2003). MOF is an OMG endorsed meta-meta model standard. For further clarification on how MOF defines itself see Figure 13. Notice at the root of the hierarchy where the *ModelElement* defines itself.

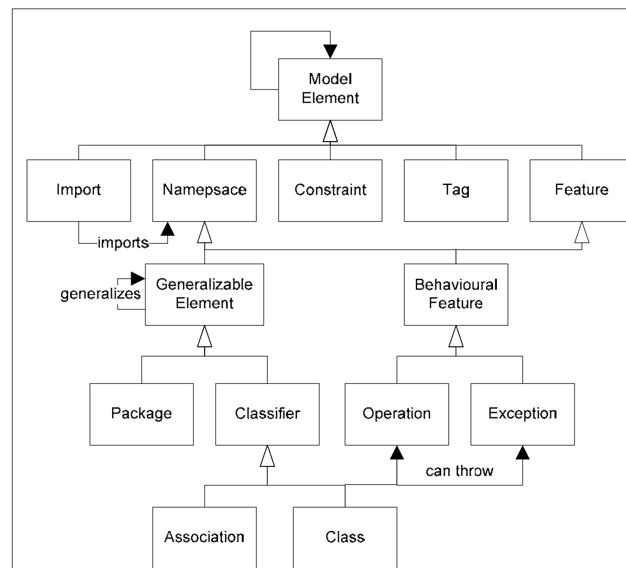


Figure 13: MOF specification excerpt reproduced from (Stahl, Völter, Bettin, Haase, & Helsen, 2003) (Object Management Group, 2008)

2.3.5 Model Transformations

Model transformations, like Metamodeling, are at the core of MDSD. Transformations make it possible for a given model at a specific abstraction level to be transformed into another model at a lower abstraction level. Transformations make the transitions shown in Figure 7 possible.

Model transformation is a vast research topic although as it is not the core topic of this thesis we will only briefly look at some of its terminology and some transformation classification techniques according to (Metzger, 2005).

To understand transformations it was necessary to understand some basic terminology such as formalism, abstract and concrete syntax and dynamic and static semantics. Abstract and concrete syntax was introduced in the previous section by (Stahl, Völter, Bettin, Haase, & Helsen, 2003). Although (Metzger, 2005) introduces a concise diagram which clarifies the above mentioned terminology. This diagram is illustrated in Figure 14.

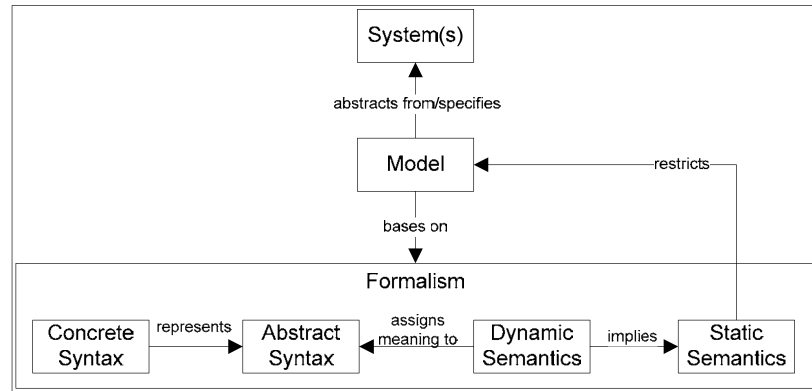


Figure 14: Formalism, static/dynamic semantics and conrete/abstract syntax reproduced from (Metzger, 2005)

Looking at Figure 14 *formalism* can be defined as a language which strictly defines a given model's syntax (abstract/concrete) and its semantics (static/dynamic). The concrete syntax defines the readability of the abstract notational elements. Static semantics defines the well-formedness of a model and is implied by the dynamic semantics which in turn defines the set of valid models for a given formalism. Given the above definitions and diagram we can define a transformation *(t)* as:

$t: M_1(S_1)|_{F_1} \rightarrow M_2(S_2)|_{F_2}$ Where *M* is a model of a system *S* and *F* is the formalism (language) for *M*. M_1 and M_2 are shown as the source and target model respectively (Metzger, 2005).

(Metzger, 2005) further describes a series of classification techniques for any given transformation depending on its degree of automation (fully automated, partially automated or manual), number of steps it has to undergo (monolithic or composite), the difference between source and target models (endogenic or exogenic) or the purpose of the transformation (vertical or horizontal). These classification techniques were briefly looked at although as this thesis does not have any direct relevance to model transformations we will omit further details.

2.3.6 Software Factories (Greenfield, 2005)

Software factories are like templates which have the ability to customize a given Integrated Development Environment (IDE) with code libraries, help files, wizards, project types and visual designers.

The research done around the idea of software factories was their relevance to MDSD. Software factories use MDSD to model information from a given implementation and also to provide automation services on that extracted information. These models used by a software factory are expressed in a DSL which are designed to support a specific given task. We see an example of a web service software factory below in Figure 15, which uses a DSL to configure and run a given IDE (Visual Studio in this case). We delve into Domain Specific Development and Languages in the following section.

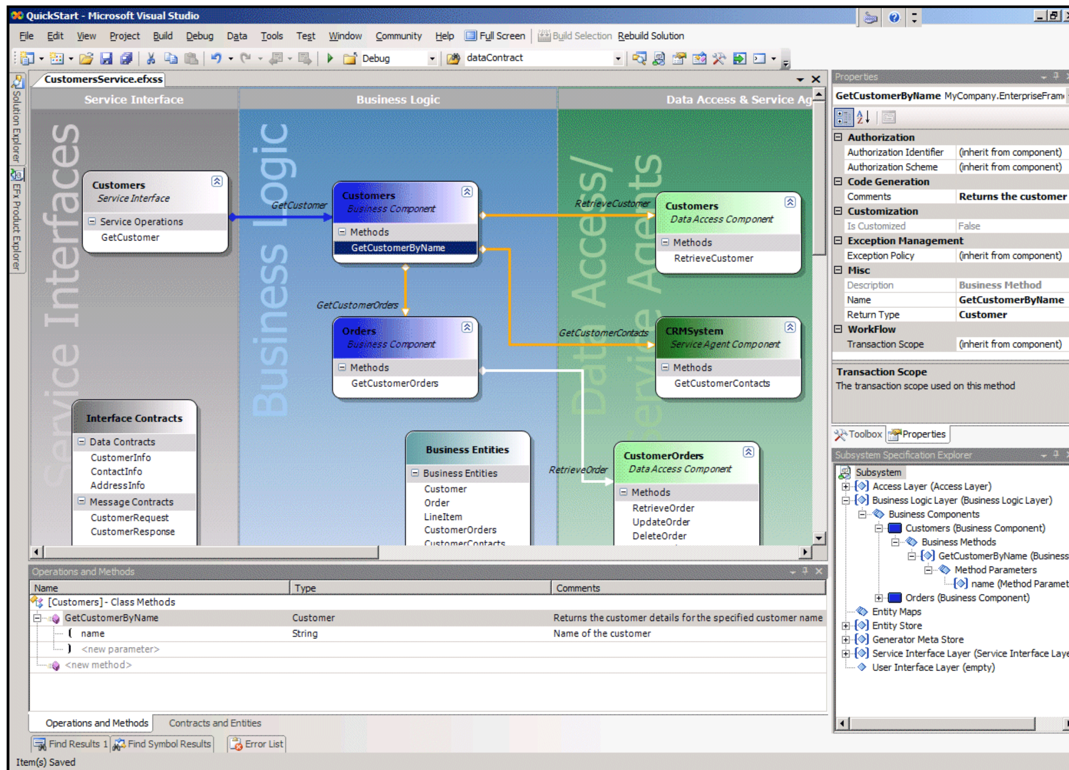


Figure 15: Software factory with a DSL reproduced from (Microsoft, 2007)

2.4 Domain Specific Development and Languages

2.4.1 Introduction

Described in Section 2.3, MDSD is at the heart of this thesis, and Domain Specific Development (DSD) and Domain Specific Languages (DSL) are the brains behind the thesis. Although DSD and DSL are two separate concepts this sub-section concentrates mainly on DSLs.

DSD is a software development methodology which relies on a DSL to provide high level abstractions than compared to traditional programming languages. Due to this reason DSD tends to reside on a higher level of abstraction, one which often facilitates development and improves productivity (DevSource, 2007).

DSLs are essentially a programming language. Unlike traditional programming languages these are designed to cater to a specific problem usually for a specific organisation or domain. To better understand this concept we introduce a few examples of commonly used DSLs before proceeding on with the rest of the introduction

2.4.1.1 Himalia Navigation Model (Himalia, 2006)

Himalia is a MDSD based user interface builder which uses a DSL to assist developers in designing high level models of a UI. An example of the DSL is shown in Figure 16 below.

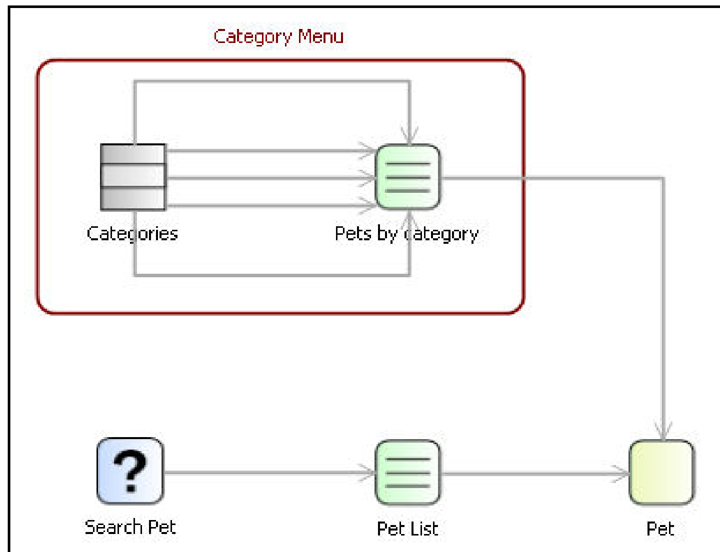


Figure 16: Himalia model of a UI reproduced from (Himalia, 2006)

The above figure shows a conceptual DSL via which a user can design web user interfaces. In this particular we see two basic functions: Searching for a pet and view pets via their categories. This DSL hides irrelevant details from the user and allows them to simply concentrate on the conceptual and business aspects of the user interface.

2.4.1.2 UI designers

Any graphical user interface designer is essentially a DSL where the domain of interest is the user interface. Although this is a more general purpose DSL than one described in Section 2.4.1.1 it is still essentially a DSL. The following figures are examples of various UI designers.

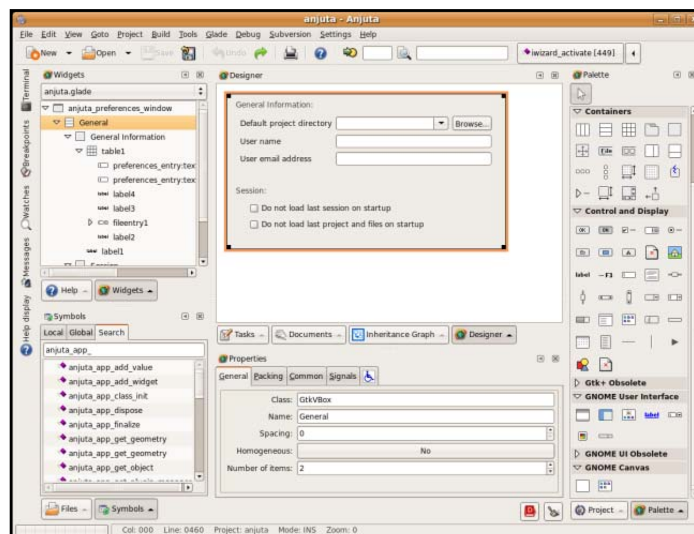


Figure 17: Anjuta, an IDE for GNOME reproduced from (Naba kumar, 2007)

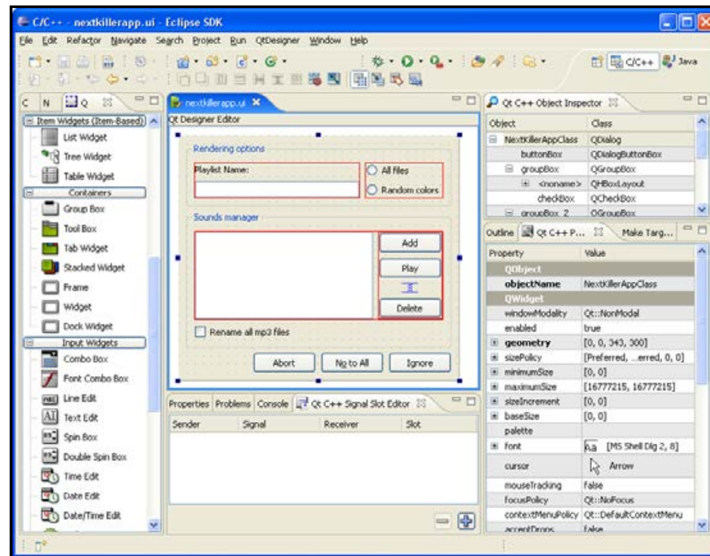


Figure 18: Qt C++ GUI designer for Eclipse reproduced from (Alessandro, 2007)

The above two figures show a UI designer, therefore the domain with respect to them is a user interface. They allow the developer to drag and drop UI components which abstracts them away from the details on how each component is instantiated and how their properties are integrated. This allows quick UI creation and is at a lower conceptual level than the example in the previous section (Section 2.4.1.1).

2.4.1.3 Regular Expressions (Regex)

Regex is a general purpose DSL. It is a textual DSL designed to solve string or text oriented problems. The domain of interest is any textual information. For e.g. in a given sentence “This is an excellent example”, regex allows us to search for all words beginning with “ex” by simply searching for “ex*”. Although regex is powerful it takes a lot to master it fully due to its textual nature and complicated notation.

2.4.1.4 HTML

Probably the most commonly used DSL would be the Hypertext Mark-up Language or HTML. HTML is a textual language (Figure 19) but can be argued otherwise as a graphical DSL as its primary use is to render WebPages on the World Wide Web (WWW). Thus, the specific domain for HTML is WWW.

```
1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
2  <html xmlns="http://www.w3.org/1999/xhtml" >
3  <head>
4      <title>Untitled Page</title>
5  </head>
6  <body>
7
8  </body>
9  </html>
10
```

Figure 19: An HTML page

HTML is an expressive language which tends to make it powerful although difficult to master due to its verbosity and enormous tag (language) library.

2.4.1.5 SQL

Structure Query Language is a language (Figure 20) used to query information from a database which makes it a DSL with databases as its specific domain. As any textual DSLs, SQL is powerful but hard to master language due to its textual nature and verbosity. In some database clients (Microsoft SQL Management Studio) SQL can be created visually. Figure 21 visually represents the SQL shown in Figure 20.

```
1  select * from Table1
2      inner join Table2 on Table1.Column1=Table2.Column1
3      where Table1.Column1='some value' and Table1.Column2=5
```

Figure 20: SQL snippet

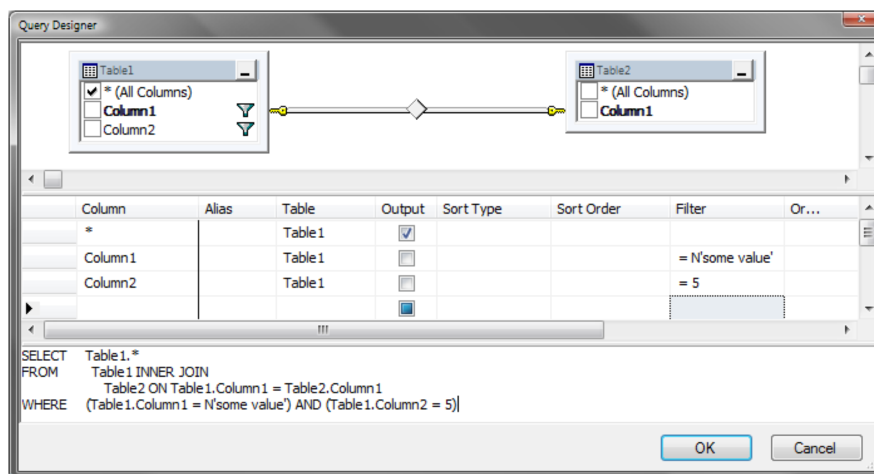


Figure 21: SQL snippet represented visually

(Cook, Jones, Kent, & Wills, 2007) gives a more in-depth definition of a DSL. It states that a DSL is a special purpose language which assists in designing the variable part of a given solution. The variability of a solution arises when several problems occur with the same aspects. A solution could then comprise of two parts, a fixed part which comprises of the framework, an Application Programming Interface (API), a platform and/or a compiler (interpreter) and the variable part which would be created using a DSL.

Further, to make a solution possible the variable part should be fully integrated into the fixed part. Two approaches are suggested by (Cook, Jones, Kent, & Wills, 2007), one approach involves the fixed part to expose an interpreter which takes the DSL (variable part) and integrates them together. Although this approach is flexible it proves problematic during debugging and has an adverse effect on performance. The second approach allows the DSL to be fully converted to code which could then be compiled together with the fixed part of the solution. Although a more complex approach it allows the solution to be scalable and allows debugging.

In the introduction we introduced two types of DSLs: visual and textual. We use a visual DSL throughout the course of this thesis although some research was done around what a textual DSL can offer. We look at textual and graphical DSLs in the following two sub-sections.

2.4.2 Textual DSL

A textual language like any programming language will need to be either parsed or interpreted, thus a custom parser which understands the language is needed. Figure 22 below shows such an example where we are creating a Rectangle shape object and setting some of its parameters.

```
1  Define AnnotationShape Rectangle
2      Width=1.5
3      Height=0.3
4      FillColor=khaki
5      OutlineColor=brown
6      Decorator Comment
7          Position="Center"
8      End Comment
9  End AnnotationShape
```

Figure 22: Textual DSL

Designing a parser from scratch is a major ask although a parser-generator may be used which works off the grammar of a given textual DSL, such as (University of Geneva). On the whole designing a textual DSL is more of an expert task, as it may require a text editor, syntax colouring, syntax checking and other modern IDE features. (Cook, Jones, Kent, & Wills, 2007) introduce two approaches to solve the aforementioned problems. The first uses the capabilities of an underlying host language, for e.g. C#. Figure 23 shows the same DSL fragment as introduced in Figure 22 using a pre-established host language.

```
1  Shape AnnotationShape = new Shape(ShapeKind.Rectangle,
2                                    1.5, 0.3,
3                                    Color.Khaki, Color.Brown);
4  Decorator Comment = new Decorator(Position.Center);
5  AnnotationShape.AddDecorator(Comment);
```

Figure 23: Textual DSL using host language

The code shown in Figure 23 is better known as configuration code as it provides a configuration for a specific set of objects and classes already existing within the host language. Another approach is to use the eXtensible Markup Language (XML). Figure 24 shows the above mentioned code fragments (Figure 22 and Figure 23) in XML.

```
1  <?xml version="1.0" encoding="utf-8" ?>
2  <Shapes>
3      <Shape name="AnnotationShape">
4          <Kind>Rectangle</Kind>
5          <Width>1.5</Width>
6          <Height>0.3</Height>
7          <FillColor>Khaki</FillColor>
8          <OutlineColor>Brown</OutlineColor>
9          <Decorator name="Comment">
10             <Position>Center</Position>
11         </Decorator>
12     </Shape>
13 </Shapes>
```

Figure 24: Textual DSL using XML

The XML approach can be made more intuitive by using an XML Schema which can contain rules on how a shape can be defined. This schema can provide type checking and suggestive help such as drop downs of potential parameters that can be supplied.

2.4.3 Graphical DSL (Visual Languages)

Any given graphical DSL will have at least one visual component to it thus making it a possible visual language. Due to this relationship, this sub-section gives a brief overview of the research carried out on visual languages and then proceeds to narrow it down to graphical DSLs.

A visual (programming) language is a process in which more than one dimension is used to convey semantics (Grundy, Visual Languages/Notations, 2008). Textual languages are regarded as one dimensional. A visual language supports the user via visual interaction and allows programming or programming like tasks using visual expressions. Visual languages can also be referred to as executable graphics, according to (McIntyre, 1994) the phrase, visual languages, raises ambiguity as a visual language could mean something that can be seen, which is trivial or it could mean a language used to program visual objects. Although for the remainder of this chapter and the thesis thereof we will be using visual languages as per its definition expressed in the beginning of this sub-section.

Not all visual languages are used for software or programming oriented tasks. Excel is a popular example which was designed to be used for financial tasks. As this thesis deals with software tasks, the research was narrowed down to only include software visual languages, mainly UML and its counterparts.

UML is a vast graphical modeling language and although it is not directly related to this thesis we still use it to show static content, therefore it is important we use this sub-section to briefly introduce what UML is all about.

UML is an approach used to graphically represent a given software system. It was formally published by the OMG in an effort to unify different diagramming approaches, namely the Booch method, the Object Modeling Technique and the Object-Oriented Software Engineering method (Cook, Jones, Kent, & Wills, 2007). The UML specification was primarily designed to express object-oriented systems, namely Java and it is not suitable for high level analysis although UML does offer extensibility points called profiles which are a packaged set of stereotypes, tagged values and constraints (Cook, Jones, Kent, & Wills, 2007). As systems become more distributed with the emergence of web services, web systems and smart clients they become more difficult to represent using UML. Although extensions such as profiles can be used to mould UML to these new emerging domains it becomes quite cumbersome and complicated. Graphical DSLs is a solution to this problem. Graphical DSLs are a flexible alternative to UML which is a rather fixed and arbitrary modeling language. They open the UML by allowing us to add extensions which we need and omit details which have no relevance to our domain.

Graphical DSLs are a major part of this thesis; we have a look at their various aspects in the following sub-sections according to (Cook, Jones, Kent, & Wills, 2007).

2.4.3.1 Notation

The notation of a graphical DSL is the interface to the real world thus it is important that they are intuitive and provide a clear meaning. UML has a notation of geometrical shapes, arrows, connectors and decorators and most graphical DSL Tools tend to inherit this behaviour. For example the Microsoft DSL Tools and the Marama Eclipse Designer both have a UML like notation. Various evaluation theories are proposed on how to make the notation of a graphical DSL more end-user

friendly; these theories include cognitive dimensions, attention investment and champagne prototyping. These theories are further explored in Section 2.4.3.6.

Metaphors are a common practice to use in software user interface design and (Blackwell & Green, 1999) introduces how metaphors if used correctly could prove advantageous for end-users while using the notation of a particular graphical DSL. Common metaphors include shape nesting to show containment, arrows and connectors to show relationships and associations and various layout techniques to show ordering.

2.4.3.2 Domain Model

The domain model for a graphical DSL is the set of domain classes and domain relationships described by the language (DSL). A domain class represents a concept within a given domain whereas a domain relationship represents the relationship between two domain classes. The domain model ties in with the notation mentioned in the previous sub-section as domain classes usually map to shapes and domain relationships map to connectors. A complete domain model will also contain a set of constraints against which an instance of the model can be validated.

2.4.3.3 Code Generation

Code generation is an integral part of a graphical DSL as at times that could be the sole reason for designing a graphical DSL. The word code in this context is used loosely and does not necessarily refer to program code. Code could represent any artifact, such as a Java/C# class, XML configuration file, a proprietary language, another diagram or a blend of all of the above.

2.4.3.4 Serialization

Persistence is an important criterion for any programming language. For a graphical DSL it is vital to save information about the domain concepts, the shapes each concept maps to, the location of these shapes and any other relevant information. This persistence can be achieved by serialization. With growing need for interoperability XML is becoming the most popular choice of technology for persisting information. The flexibility and scalability of XML also aid in the ability to mould it to suit any domain model.

2.4.3.5 Tool Integration

A graphical DSL can be hosted within its own isolated IDE although it is common practice to use the capabilities of a well established IDE, for e.g. hosting a graphical DSL within the Visual Studio Shell or in an Eclipse environment. Considering tool integration of a graphical DSL allows us to realize the non-vital segments of our model such as its file extension, custom property editors, visual appearance of the properties of domain concepts, shape repository or toolbox appearance, mouse and keyboard behaviour, model explorer and menu commands.

2.4.3.6 Evaluation techniques

Evaluation techniques were introduced in Section 2.4.3.1, these sub-section looks further into those evaluation techniques as per (Grundy, Visual Languages/Notations, 2008).

- ***Cognitive Dimensions*** (Green, 1996)*:* Cognitive dimensions are a set (framework) of dimensions which formally establishes trade-offs made during the design of the notation of a DSL. The following describes these dimensions and explains briefly what each of them represent:

- *Abstraction Gradient:*
  Quantify the level of abstractions possible, establish the maximum and minimum and determine whether fragments can be grouped.
- *Closeness of mapping:*
  Measures the proximity of a given notation to what it represents in the real world.
- *Consistency:*
  If part of a given notation can be learnt, is it possible to assume the rest?
- *Diffuseness:*
  The number of symbols required to express a concept.
- *Error-proneness:*
  Does the notation help users avoid making mistakes?
- *Hard mental operations:*
  Besides the concrete notation, does the user need an external resource to keep track of what is happening?
- *Hidden dependencies:*
  Are relationships explicitly stated in both directions, how is it stated? Perceptually or by using symbols?
- *Premature commitment:*
  Do users have all the information needed before making a decision?
- *Progressive evaluation:*
  Is execution of partial models allowed to determine progress and gain feedback?
- *Role-expressiveness:*
  From the model can each domain concept be scrutinized independently and its role within the model determined?
- *Secondary notation:*
  Is Layout and colour used effectively to convey extra information beyond the formal semantics of a given language.
- *Viscosity:*
  Effort required bringing about a change in the model.
- *Visibility:*
  Caters for the visibility of code and model, whether they can be viewed simultaneously, if not, is it possible to determine the flow of code while reading it.

- ***Attention Investment*** (Blackwell & Green, Investment of Attention as an Analytic Approach , 1999)*:* This provides a cost benefit analytical approach to programming and the reason why people spend time doing programming. Any programming activity can be regarded as having a cost in terms of attention units, an investment aspect which signifies the attention units which eventually pay-off a reward, a pay-off aspect which reduces future cost due to the investment aspect and a risk factor aspect in case there is a negative pay-off.
- ***Champagne Prototyping*** (Blackwell, Burnett, & Jones, Champagne Prototyping: A Research Technique for Early Evaluation of Complex End-User Programming Systems)*:* This approach assists in answering questions at an early stage of notation design. It follows a simple "look don't touch" approach where a simple prototype is designed and then evaluated using cognitive dimensions and attention investment strategies.

- ***User Survey/Evaluations**:* We will be evaluating our visual language using a subset of the end-user population and getting vital statistics on how effective the language and its notation are.

2.4.4 Advantages/Disadvantages of DSLs

This section briefly states some of the advantages and disadvantages of DSLs as a whole (Beydeda, Book, & Gruhn, 2005) (Cook, Jones, Kent, & Wills, 2007) (Wikipedia, 2007).

2.4.4.1 Advantages

- DSLs raise the level of abstraction for a particular problem and its corresponding domain, due to this any user in that domain could potentially understand, validate and develop solutions.
- As DSLs are defined using words specific to a domain they tend to be self-documented.
- DSLs enhance productivity, program quality, scalability, maintainability and portability.
- As DSLs models are validated against domain level constraints it is safe to assume that any expression or model designed using that DSL is a valid expression.

2.4.4.2 Disadvantages

- Using DSLs is not currently the norm, due to this reason the cost of designing, implementing and maintaining a DSL tend to be high.
- The domain on which DSLs are designed on have the potential of expanding and changing constantly, due to this reason it is often difficult to define a fixed scope for a particular DSL for that domain.
- Using DSLs comes hand in hand with code generation. This could potentially be a performance setback as generated code may not always be as high performing as hand-written code.
- DSLs usually tend to be hard and at times impossible to debug.

2.5 DSL Tools

The previous section outlined the research carried out in the textual and graphical DSL area. This section outlines the experimentation of two popular tools to create graphical DSLs, namely the Microsoft DSL Tools and Marama.

2.5.1 Microsoft DSL Tools

Microsoft has the DSL Tools as part of its extensibility framework known as the Visual Studio Software Development Kit (SDK). The DSL Tools provide the user with graphical designers, XML serializers, set of code generators and a framework for code generators. Figure 25 below shows how these components integrate together to form the DSL Tools.

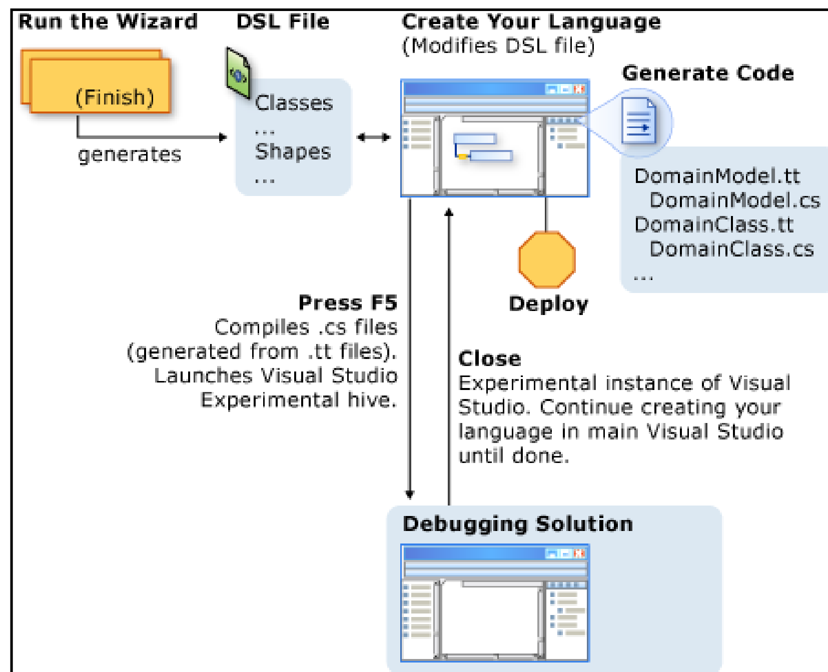


Figure 25: Microsoft DSL Tools overview reproduced from (Microsoft, 2007)

Let us systematically look at each of these components and briefly describe their role within the DSL Tools.

2.5.1.1 Wizards

The Microsoft DSL Tools ships with four DSL wizards shown in Figure 25.

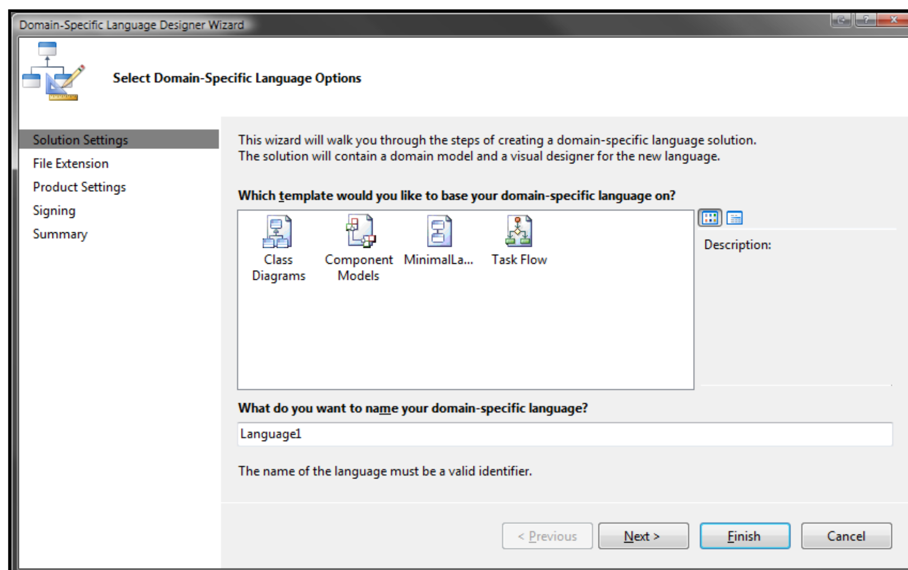


Figure 26: Microsoft DSL tool wizards

- *Class Diagrams:* The class diagrams wizard creates a UML type DSL model. Aids the user in creating class diagrams following the traditional UML notation.
- *Component Models:* Allows the user to create a DSL which can have interconnected ports, useful for creating circuit diagrams and other embedded DSLs.

- *Minimal Language:* Gives the user a basic executable DSL which can be used by advanced users to create custom DSLs.
- *Task Flow:* Typically used to create flow chart like DSLs.

2.5.1.2 Graphical Designer

The essence of Microsoft DSL Tools lies within its graphical designer. It allows the DSL designer to drag-drop domain concepts, relationships, shapes and concept-shape maps to create a custom model. The designer also provides customization points via which a DSL designer can manipulate any aspect of the model such as how it is serialized, the shapes toolbox and the model explorer. An example of the graphical designer is shown below in Figure 27.

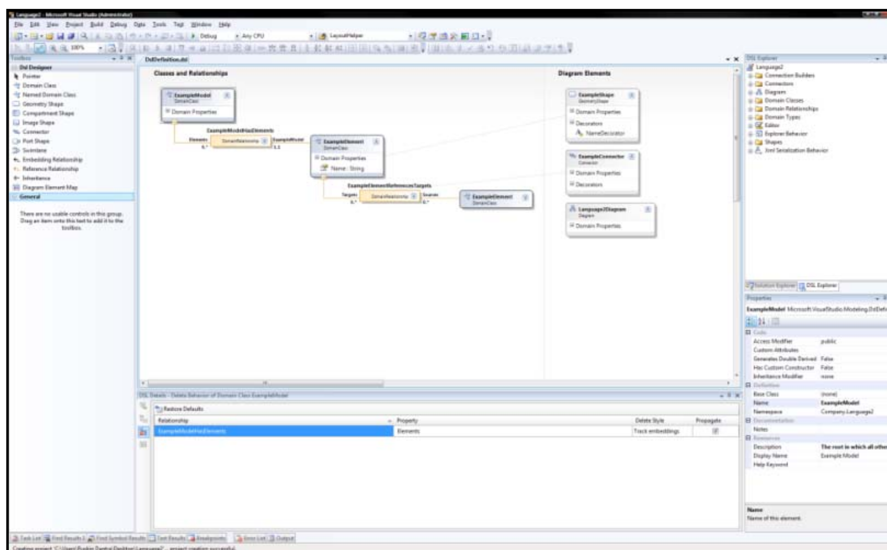


Figure 27: Microsoft DSL Tools graphical designer

2.5.1.3 Packaged Code Generators

The packaged code generators create an executable implementation as output by taking the domain model definition and the designer definition as its inputs. It also allows validation of the domain model and the designer against set constraints and raises errors and validation warnings if need be.

2.5.1.4 Experimental Hive

The experimental hive simulates the end-user environment by allowing DSL designers to experiment and test their newly created graphical language without actually deploying it. The experimental hive loads the model definition and the designer definition into memory off which the DSL designer can create sample models.

2.5.1.5 Text Templates

Microsoft DSL Tools ship with a powerful templating engine called the T4 Text Template Engine. This allows the DSL designer to write templates which can run off a model created by an end-user and output appropriate code. As suggested in Section 2.4.3.3, this code could be anything from an XML configuration file to another composite domain model. A code snippet of a T4 text template is shown below in Figure 28.

```
<#@ template inherits="Microsoft.VisualStudio.TextTemplating.VSHost.ModelingTextTransformation" #>
<#@ output extension=".txt" #>
<#@ Language2 processor="Language2DirectiveProcessor" requires="fileName='Sample.mydsl1'" #>

Generated material. Generating code in C#.

<#
  // When you change the DSL Definition, some of the code below may not work.

  foreach (ExampleElement element in this.ExampleModel.Elements)
  {
#>
    <#= element.Name #>
<#
  }
#>

```

Figure 28: T4 text template snippet

2.5.2 Marama

Marama is an Eclipse (a Java IDE) based meta-toolset for constructing multi-view graphical DSLs (Grundy, Hosking, Huh, & Li). Marama is designed for expert modelers who have experience in modeling concepts such as EER (Extended Entity Relationship) models, OCL (Object Constraint Language) and meta-models. Marama aims to provide these users with a quick way to create visual modelers without concentrating on other DSL tasks such as code-generation or behavioural constraints.

Marama is a composite tool comprised of a set of Eclipse plug-ins which leverages the GEF (Graphical Eclipse Framework) and EMF (Eclipse Modeling Framework). It also used Kaitiaki (Liu, Hosking, & Grundy, 2007) for view level behaviour (Grundy, Hosking, Huh, & Li). Figure 29 below shows a high level view of the Marama architecture.

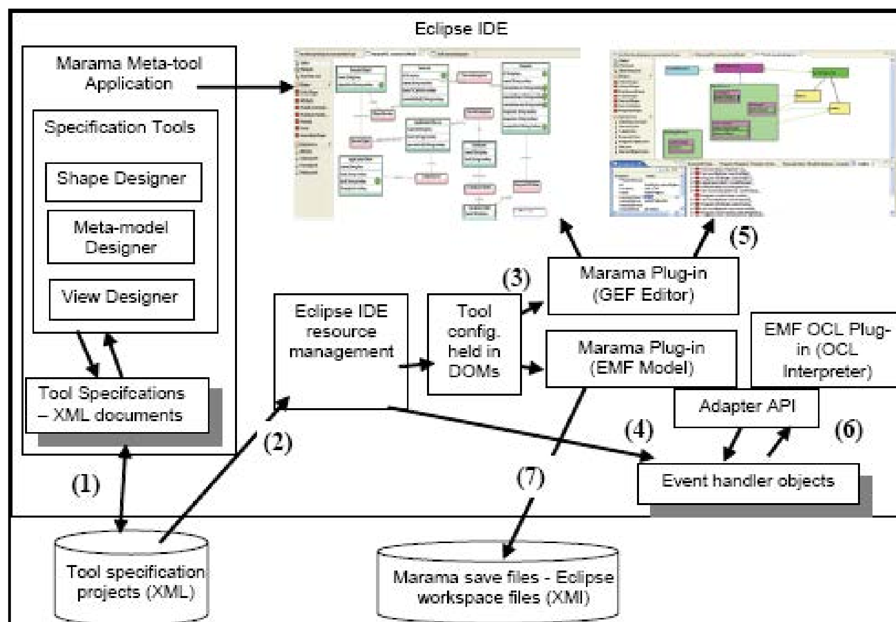


Figure 29: Marama architecture reproduced from (Grundy, Hosking, Huh, & Li)

Marama contains of three core concepts which are defined briefly below:

2.5.2.1 Meta-Model Designer

Figure 30 shows a screen shot of the Marama Meta-Model Designer. The designer is the central place where a DSL designer can create an EER representation of the system intended. It also couples with OCL to provide model level constraints which can be added or edited via a custom visual editor.

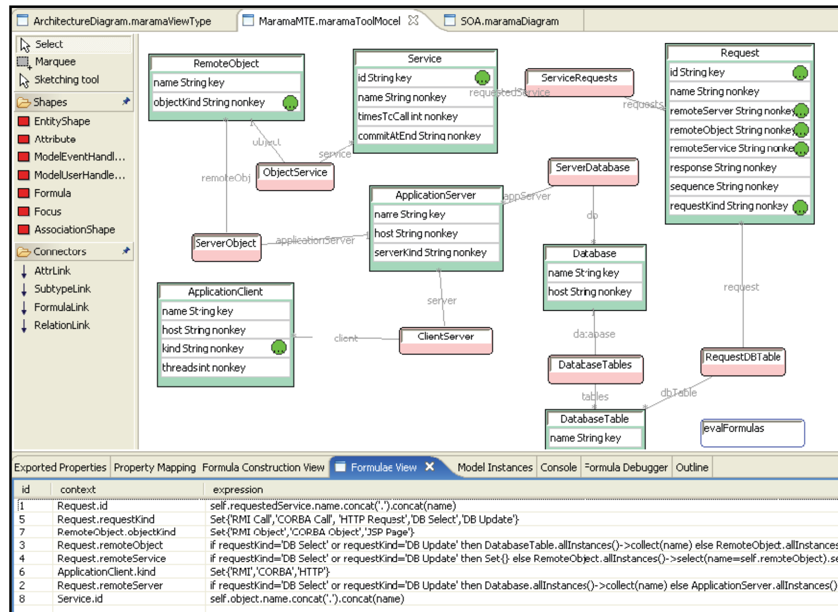


Figure 30: Marama Meta-Model Designer reproduced from (Grundy, Hosking, Huh, & Li)

2.5.2.2 Visual Shape Designer

The Marama Visual Shape Designer, as the name suggests allows the DSL designer to create shapes which can be used to represent domain concepts. The Visual Shape Designer is shown below in Figure 31.

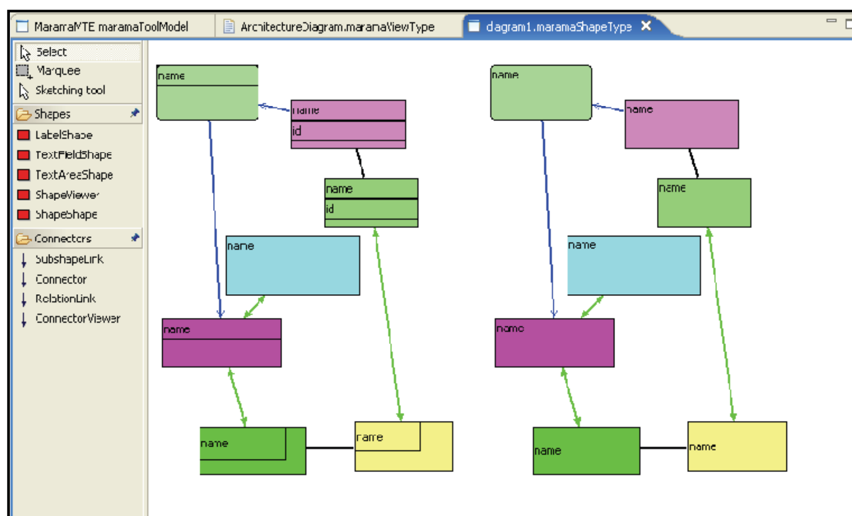


Figure 31: Marama Shape Designer reproduced from (Grundy, Hosking, Huh, & Li)

2.5.2.3 View Designer

The Marama View Designer brings the Marama Meta-Model Designer and the Marama Visual Shape Designer together. It allows the DSL designer to specify how and which domain concepts are mapped to which shapes. It also allows the specification of additional constraints on the designer such as various containment rules. An example of the View Designer is shown below in Figure 32.

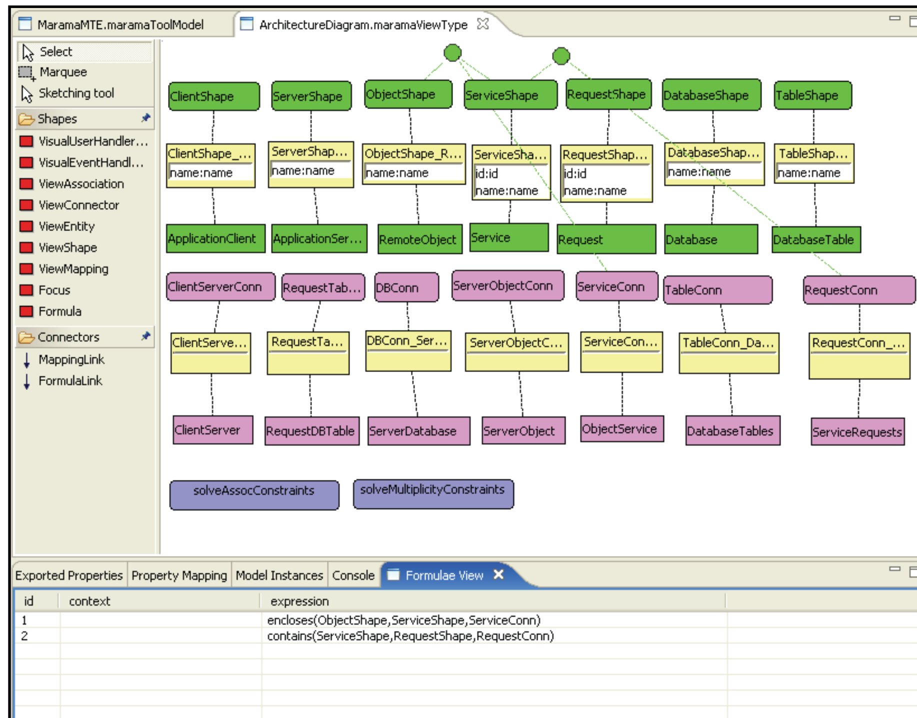


Figure 32: Marama View Designer reproduced from (Grundy, Hosking, Huh, & Li)

2.5.3 Comparison (MS DSL Tools Vs Marama)

The table below summarizes some of the core differences between the DSL Tools provided by Microsoft and Marama.

Table 1: Microsoft DSL Tools Vs Marama

Feature	Microsoft DSL Tools	Marama
Language	C#	Java
Provider	Microsoft	Auckland University
Help	Forums/Books	Research Papers
Code Generation	T4 Templates	3rd Party plug-ins. E.g. JET
Constraint Specification	C#	OCL (Object Constraint Language)

2.5.3.1 Language

The Microsoft DSL Tools inherently provide support for the C# programming language and Marama is designed for Java. I personally have intimate knowledge of the C# programming language and the .NET CLR framework so was more inclined towards the DSL Tools provided by Microsoft.

2.5.3.2 Provider

The Microsoft DSL Tools is a more established framework when compared to Marama. Due to this reason it is easier for new DSL designers to start using this framework as opposed to Marama which may require the developer to go through a learning curve.

2.5.3.3 Help

There is a learning curve involved before using either the Microsoft DSL Tools or Marama although what differentiates them is the amount of help offered on these technologies. Microsoft DSL Tools have a dedicated forum (http://forums.microsoft.com/msdn/) and an in-depth book (Cook, Jones, Kent, & Wills, 2007) outlining its details and how to use this new framework. The help provided for

Marama is either in forms of small tutorials or research papers which do help although fall short when compared to the resources provided by Microsoft on its DSL Tools framework.

2.5.3.4 Code Generation

The Microsoft DSL Tools allows developers to write script like code using the powerful T4 templating engine to traverse through a model designed by the end-user to output any textual artefact. Marama does not inherently provide any code generation frameworks although several 3rd party tools can be used such as JET (Java Emission Templates).

2.5.3.5 Constraint Specification

Constraint specification within Microsoft DSL Tools can be done via simply writing C# code which becomes supplementary to the domain model code. This makes it easier for the developer as the constraints are written in the same language as the rest of the system. Constraint specification within Marama is done via OCL which requires the developer to learn a new language thus increasing the learning curve.

2.5.4 Other DSL Tools

Section 2.5.1 and Section 2.5.2 introduced the two main DSL tools researched as part of this thesis. This section briefly looks at some of the other tools (Grundy, Other Meta Tools, 2008) which provide similar functionality. Note that these did not form the core of the research as we were strongly inclined towards the Microsoft DSL Tools due to the reasons described in Section 3.3.1.1.

2.5.4.1 MetaBuilder

MetaBuilder is described as a tool which enables rapid creation of diagram editors for structured diagrammatic notation using an object-oriented, graphical meta-modeling technique (Ferguson & Hunter).

2.5.4.2 MetaEdit+

MetaEdit+ is a cross platform modeling framework which allows developers to create graphical diagrams and expose them to other developers as matrices or table (MetaCase, 2008). MetaEdit+ has four parts to it: Digram Editor, Matrix Editor, Table Editor and various Browsers.

The diagram editor (shown below in Figure 33) allows the creation, management and maintainence of design information using a visual diagramming notation.

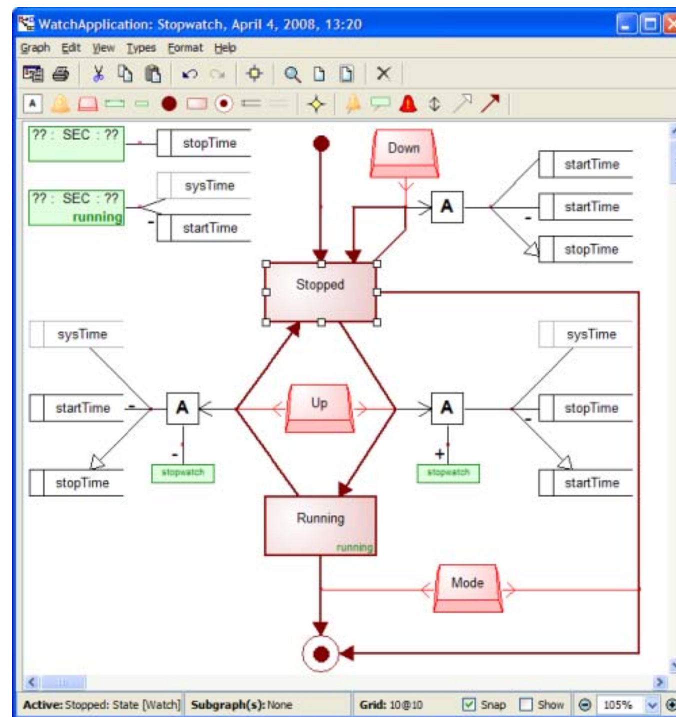


Figure 33: MetaEdit+ Diagram Editor reproduced from (MetaCase, 2008)

The Matrix Editor provides a matrix approach to allow the manipulation of objects and their relationships. It contains two axis containing objects and corresponding cells showing relationships as shown below in Figure 34.

Process/Entity Matrix: Sales and inventory, August 28, 2006, 9:32

Graph Edit View Types Axis Cell Format Analysis Help

	Receiving	Acquisition	Priority check	Unpack
Buy-order	read	create		
Supplier	read	read	update	
Prices		read	read	
Product	read	read	read	read
Inventory info		update		update
Delivery info	create	read	read	update
Reminder order			read	

Figure 34: MetaEdit+ Matrix Editor reproduced from (MetaCase, 2008)

The Table Editor is shown below in Figure 35, it allows a tabular or form based interface to the design information. The Table Editor is described to easily allow for data input into objects.

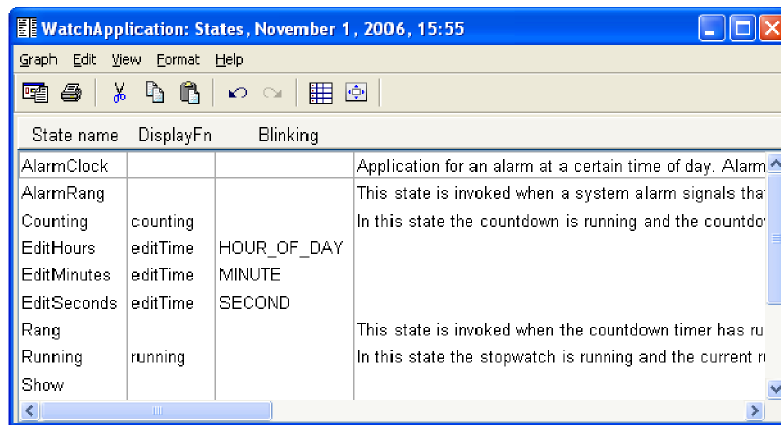


Figure 35: MetaEdit+ Table Editor reproduced from (MetaCase, 2008)

The MetaEdit+ Browsers allows developers to view DSL families and allows for easy editing via the Type Browser, Graph Browser and the Metamodel Browser shown below in Figure 36.

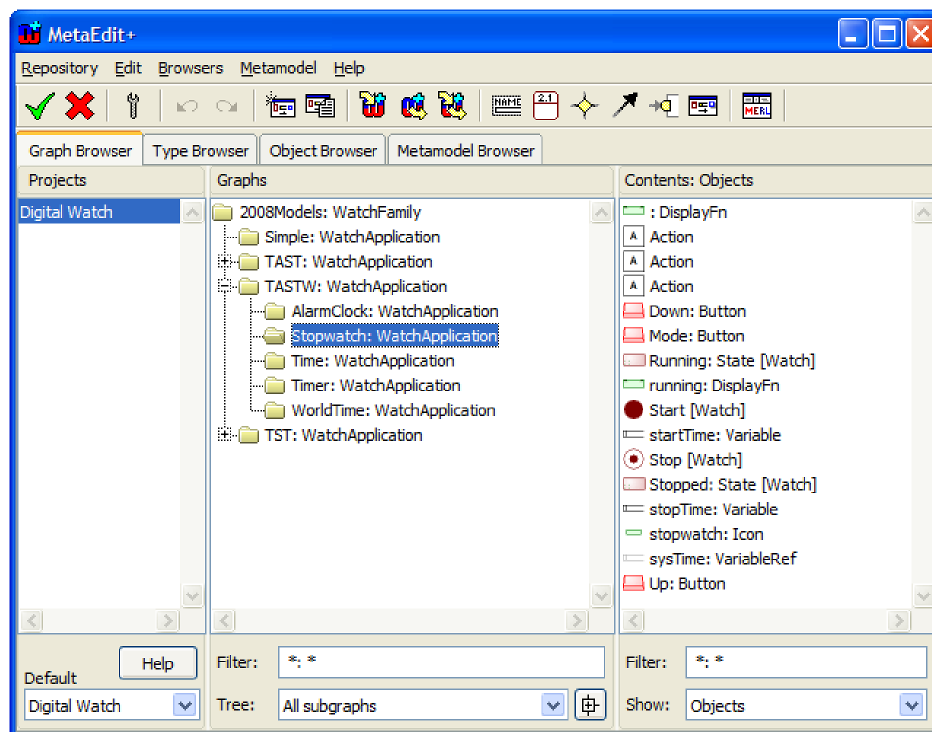


Figure 36: MetaEdit+ Browsers reproduced from (MetaCase, 2008)

2.5.4.3 GMF

Eclipse Graphical Modeling Framework (GMF) is a framework comprised of the Eclipse Modeling Framework (EMF) and Graphical Editor Framework (EMF) to build graphical editors. It allows developers to create DSLs using their meta-tools to specify the meta-model, graphical shapes and connectors and their mappings.

The figure below is an example reproduced from (Eclipse.org, 2008), which shows the capability of GMF to create a graphical editor to be used with the BPMN (Business Process Modeling Notation) specification.

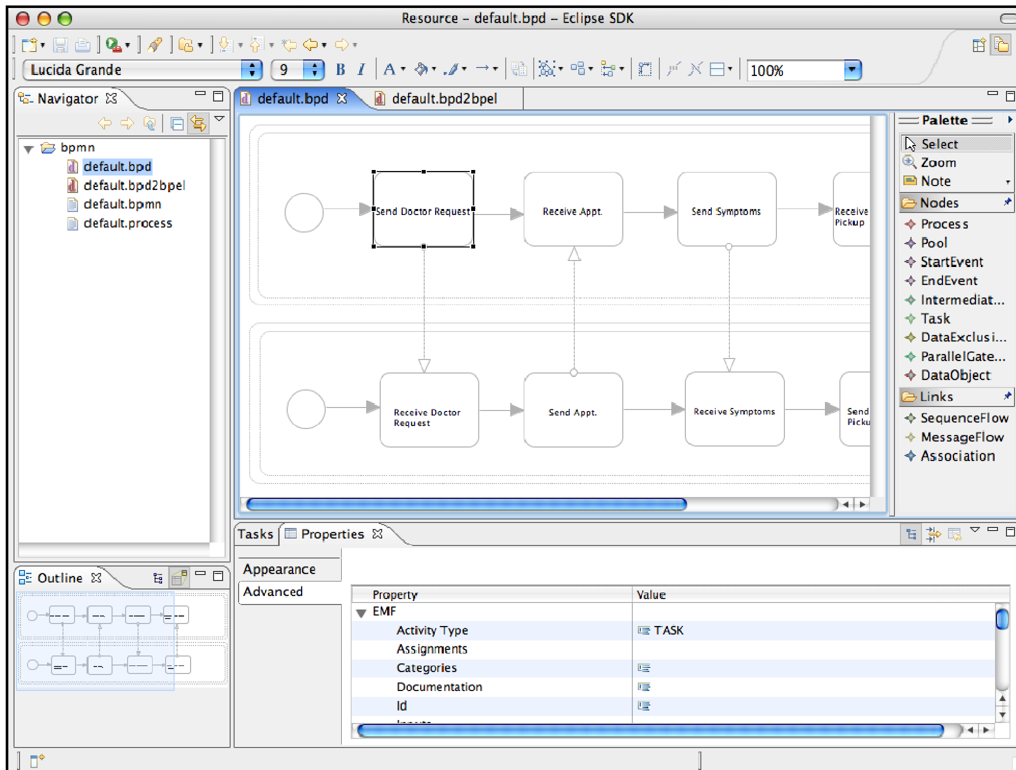


Figure 37: BPMN editor designed using GMF reproduced from (Eclipse.org, 2008)

(Pelechano, Albert, Muñoz, & Cetina, 2006) shows us an empirical comparison between the Eclipse GMF framework and the Microsoft DSL Tools and concludes by suggesting that the Eclipse modeling tools are more promising than its Microsoft equivalent. Although the study was done in early 2006, Microsoft DSL Tools have matured since then and were used to implement the solution described in this thesis (further justification in Section 3.3.1.1). (Pelechano, Albert, Muñoz, & Cetina, 2006) describes a table in which it compares DSL Tools to the Eclipse GMF tools, this is shown below.

Table 2: Microsoft DSL Tools vs. Eclipse GMF

	DSL Tools	Eclipse GMF
Metamodeling	Proprietary Notation	EMF (eCore)
Repository	XML file	XML, XMI
Graphical Notation	Direct Manipulation of XML files	GMF
Model to Model	Lack of Explicit technique	ATL, MTF, Viatra2, etc
Model to Text	Proprietary Template Language	MOFScript, FreeMaker, Velocity, JET, etc

2.6 Prism WIN Scribe IDE

Prism WIN Scribe is a tool designed to assist users while designing Prism reports. It is basically a text editor with some helpful features which are highlighted below. We also look at its current limitations and drawbacks.

2.6.1 Features

2.6.1.1 Report Wizard

Scribe offers wizards, an easy way for user to create new reports and insert header information into them. These wizards are shown below in Figure 38. The left hand side of the figure shows a wizard which provides the user to create a new report using commonly established templates and the right hand side of the figure shows a wizard which allows the user to enter header information for a newly created report.

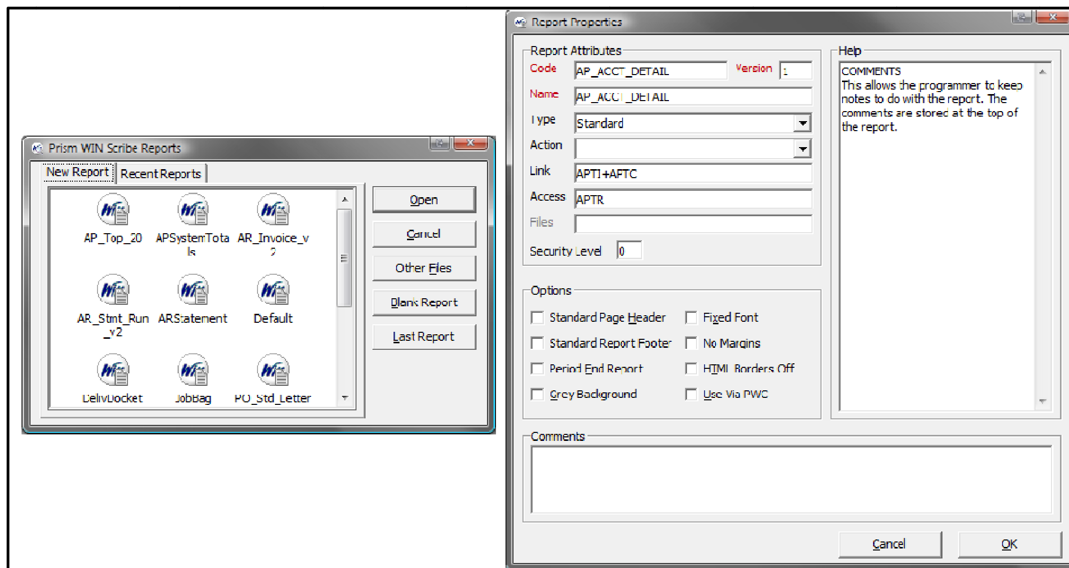


Figure 38: Scribe wizards

2.6.1.2 Report Partitioning

Scribe allows its users to modularly work on a report script by grouping different aspects of the script into different partitions. For e.g. all variables will be grouped together in the variable section. This is done via tabs, shown below in Figure 39.

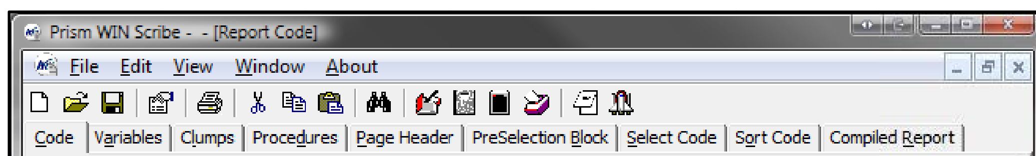


Figure 39: Scribe report partitioning

2.6.1.3 Libraries

Scribe provides with the users with three core libraries.

A meta-data library of the Prism WIN database, this contains a list of views, columns, relationships and indexes contained within the database. This informs the user of the potential information that can be extracted from the database. The meta-data library is shown below in Figure 40.

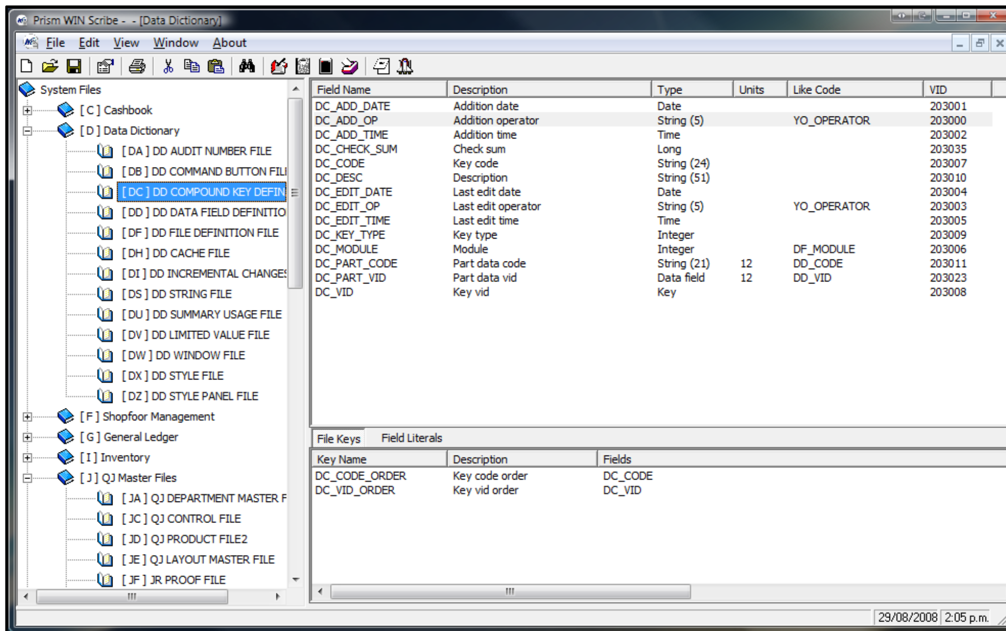


Figure 40: Scribe meta-data library

The function library lists the functions that exist within the Prism RWL along with a short description of what the function does. This is shown below in Figure 41.

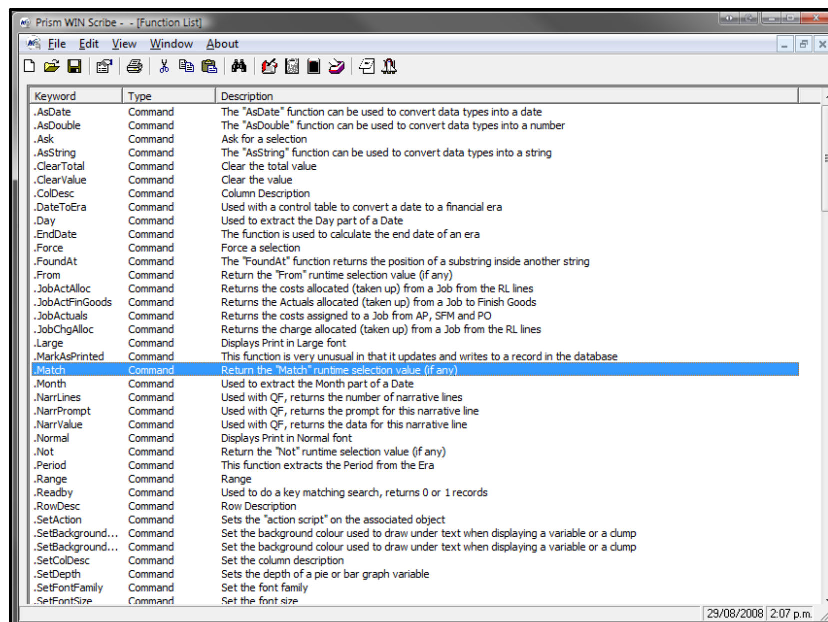


Figure 41: Scribe function library

The code library lists a set of commonly used code snippets. The user can also store custom code snippets here. The code library is shown below in Figure 42.

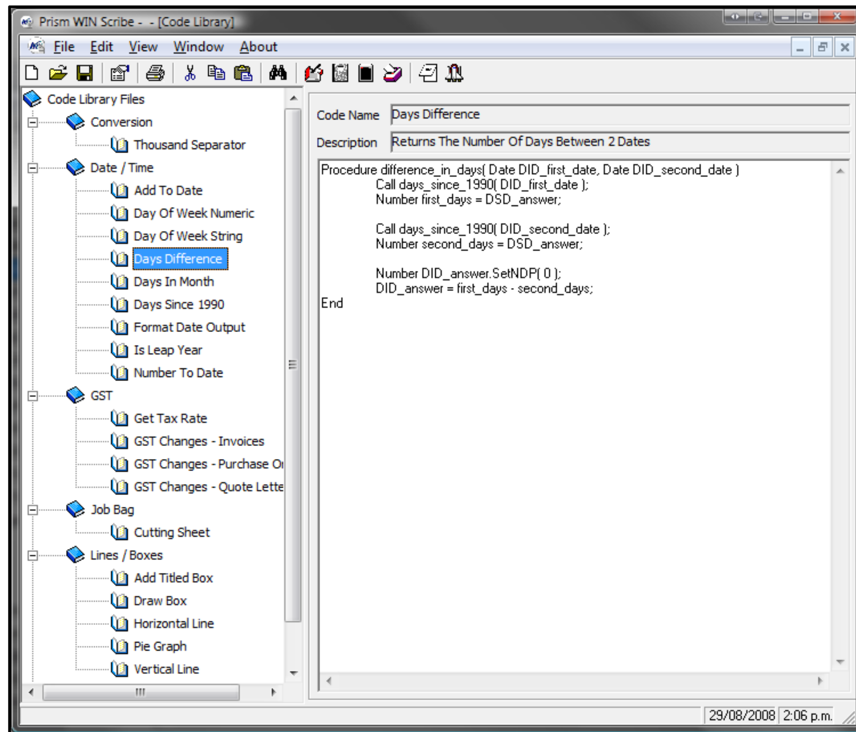


Figure 42: Scribe code library

2.6.1.4 Syntax colouring/completion

Scribe provides syntax colouring to increase readability of the Prism RWL. It also provides a drop down list of context sensitive information to the user while typing, helping the user to make common decisions, such as what functions are available, what column names exist in the database, what sort key to use on a particular table and so on. An example of this is shown below in Figure 43.

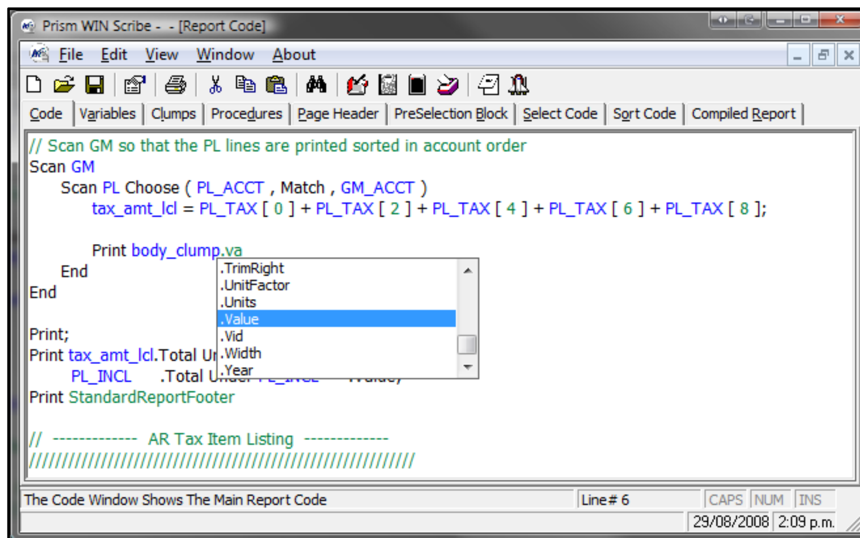


Figure 43: Scribe syntax colouring/completion

2.6.1.5 Import/Execute

Scribe can import text files and convert them to report scripts separating out individual partitions which were described in Section 2.6.1.2. Scribe can also execute the current RWL script the user is working on via the Prism WIN MIS executable.

2.6.2 Limitations

2.6.2.1 Technology

Scribe has been written in native C++ and is currently lagging behind new technology. Due to this reason Scribe has trouble running on Microsoft Windows Vista. Also bringing about a change in Scribe to match new RWL semantics and syntax is difficult due to this technological barrier.

2.6.2.2 Non-context sensitive

Even though Scribe provides syntax completion, it is not context sensitive. For e.g. the RWL specification states that the function *Year* can only be applied to a variable of type *ERA*, but from Figure 43 it can be seen that *Year* appears as a function for any variable type. Moreover, Scribe does not offer any database context sensitive help, for e.g. if a *Scan* (introduced in Section 1.4) is done on view *RM*, the user can only print information from that view, although Scribe does not recognise this and allows the user to print any column. This flaw is shown below in Figure 44.

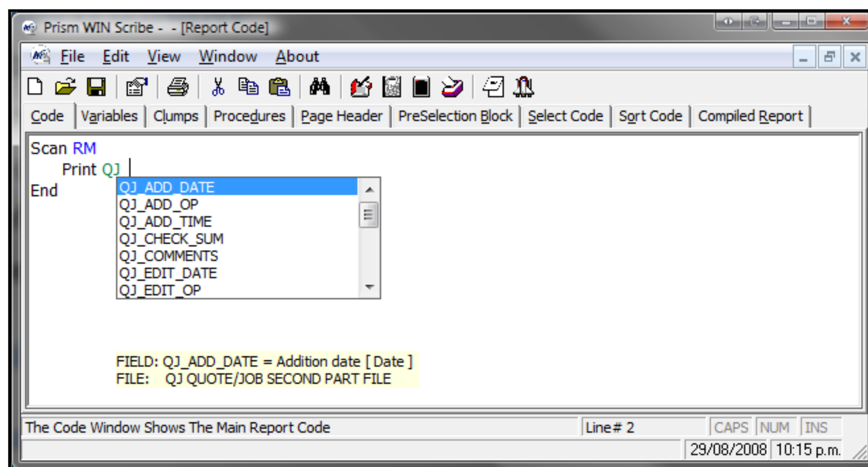


Figure 44: Scribe non-context sensitive auto completion

2.6.2.3 Executing reports

Section 2.6.1.5 described how Scribe can execute reports, although Scribe can only execute reports in a specific version of the Prism WIN system. The current version of Prism WIN is not compatible with Scribe. This limitation is important, as for users of the latest Prism WIN system there is no possible way of directly testing reports written via Scribe. Note that this is not entirely true as a user can still manually import a report in via Prism WIN and then execute it.

2.6.2.4 Text based

Scribe is entirely text based which makes it difficult to use for novice and intermediate users. Users have to learn the RWL syntax and also the semantics of how to put it together (Section 1.5.2). Moreover, the Prism database structure is complicated as explained in Section 1.5.1 thus, coupled with the learning curve of the RWL it makes it very cumbersome for a novice user to write a simple Prism WIN report.

2.7 Summary

This chapter gave a detailed insight into the research involved and the state of the art of some of the technologies involved in this thesis. We looked at software development methodologies and also the core areas involved in this thesis like modeling approaches, model-driven software development,

Metamodeling and other MDSD related areas. Finally, we concluded the chapter by giving an introduction to two current tools involved in DSL and MDSD development and an introduction to Prism WIN scribe, the in-house reporting IDE.

Chapter 3 - Our Approach

3.1 Introduction

This chapter provides a summary of this research programme which is elaborated upon in the following chapters.

We start by giving a very high level overview of our approach and then further explain the details of each process involved in our approach.

It also describes the methodologies and standards used which provides a framework for developing a prototype solution and outlines the technologies used to achieve it. It also describes how the requirements were drawn together for the solution.

3.2 High Level Approach View

This section gives the four core steps involved in our approach:

1. Design the RWL meta-model (using Microsoft DSL Tools)
2. Specify RWL constraints (using the meta-model or custom code)
3. Design our code generators (using text templates)
4. Expose the meta-model to the end-user using a UI (WPF or the Visual Studio Shell)

3.3 Software Technologies Used

3.3.1 Microsoft DSL Tools

3.3.1.1 Justification

For the thesis we chose to use the tools provided by Microsoft in its Visual Studio SDK. The core reason for this is related to Prism's business partnership with Microsoft and its strong existing usage of Microsoft technologies. Using Microsoft DSL Tools would make the integration process easier and would also reduce the learning curve if the solution suggested by this thesis had to be developed further by Prism personnel. On a more personal note, I am familiar with Microsoft products and its software development IDE, Visual Studio 2008, as it closely matches my skill set.

Note that even though we have used Microsoft DSL Tools to implement the solution it is completely possible to switch to a different technology such as the EMF/GMF Eclipse frameworks as suggested by (Bézivin, Hillairet, Jouault, Kurtev, & Piers).

3.3.1.2 Model Design

Throughout the course of this thesis the Microsoft DSL Tools model designer was the central point for starting the implementation phase. Our approach included designing a meta-model of the RWL language which included model fragments of the Prism WIN database. This meta-model was represented by domain concepts such as domain classes and domain relationships along with their corresponding shapes using Microsoft DSL Tools. We systematically and progressively started depicting each RWL construct within this meta-model. Although it was impossible to represent all the constructs of the RWL in this meta-model, a proof of concept was designed which showed that this is possible.

Note the usage of the term *meta-model* in the above paragraph. Let us take this opportunity to introduce the concept of the meta-model of the RWL and thus the difference between the meta-model and the model itself. As the RWL is a programming language, like any programming language, it has a set syntax and a set of constraints; these are represented by the meta-model of the RWL. A model of this particular meta-model will be instantiated by the end-user. Therefore, an instantiated model of the meta-model will essentially be a RWL script which is visually represented using the newly created meta-model.

3.3.1.3 Constraint Specification

A constraint is basically a condition which should be enforced by a given environment which in this case is a reporting language. Constraints within the RWL are identified with the use of (Prism New Zealand, 2005). These constraints were then sub-divided into three sub-groups highlighted below:

- *Hard Constraints:* Hard constraints are constraints which can be enforced by the meta-model of the RWL. For e.g. a report has to have one *ControlLine* structure. The *ControlLine* structure was introduced in Figure 4 and further elaborated on in Section 1.4.1. Figure 45 below shows how the meta-model can enforce this constraint. Note the multiplicity on the *ControlLine* (1..1) denoting a mandatory element.

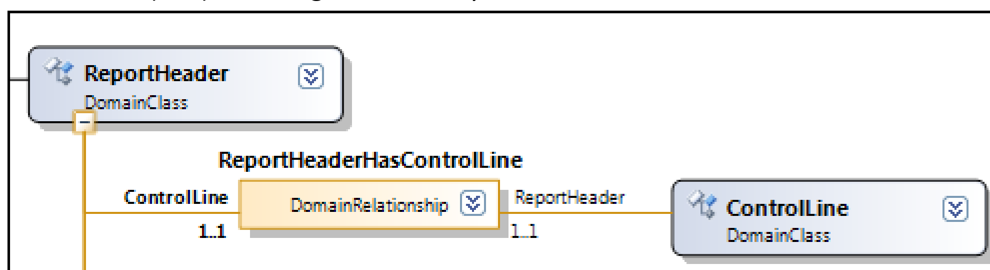


Figure 45: RWL hard constraint

- *Soft Constraints:* Soft constraints are defined as restrictions which cannot be enforced by the meta-model of the RWL. Constraints such as mandatory fields or other parameter values. For e.g. The parameter *Code* within the *ControlLine* structure is a mandatory value, thus the only way to enforce this by writing custom code. Microsoft DSL Tools exposes extension points within its generated code which allows DSL designers to design extensive constraint checking rules.
- *Dynamic Constraints:* Dynamic constraints are similar to soft constraints and can only be enforced by custom code. Although they are defined at runtime usually depending on user input. For e.g. if a *Scan* is executed on a view called RM, the user can only print information from the view within the scope of the scan. Therefore, only information contained in columns like RM_* can be printed (refer to Section 1.5.1).

3.3.1.4 The Experimental Hive

The Experimental Hive was briefly introduced in Section 2.5.1.4 and shown in Figure 25. We use the Experimental Hive continuously during this thesis to execute our solution. The Hive is essentially an instance of Visual Studio with our DSL templates pre-loaded within it. It works as a temporary host for our newly created meta-model and allows us to design RWL models against it. The Hive also allows us to debug our meta-model and its constraints.

3.3.1.5 Text Templates

Introduced in Section 2.5.1.5, the text templating engine is complementary to the DSL designer. Microsoft DSL Tools generate the domain model code from the notation using text templates (Section 2.5.1.3) and use text templates to output text from a model created by the end-user. Our solution includes a set of text templates, which when run over a created model, outputs RWL script corresponding to that model.

3.3.2 Windows Presentation Framework

Windows Presentation Framework (WPF) is the next generation of the Windows Forms toolkit provided by Microsoft. It allows for the creation of rich user interfaces which can run seamlessly on Microsoft's new operating system, Windows Vista. We chose to design our solution using the WPF framework as it fits perfectly with Microsoft DSL Tools and is XML based which gives us the future capability of generating designers using the DSL Tools.

WPF was used in one of our approaches to provide a UI which exposed the RWL meta-model to the end-user.

3.3.3 Language Integrated Querying

Language Integrated Querying (LINQ) is a new technology introduced by Microsoft which allows developers to write SQL like syntax using C# or VB to query any "LINQ supported" objects. A LINQ supported object is any object from SQL databases to XML files to even an array.

As introduced in Section 1.5.1, the Prism database stores its own meta-model. LINQ lends itself to our solution and allows us to query Prism's database meta-model without writing complex SQL queries. Moreover, it makes our code easily readable increasing its quality.

3.4 Methodologies and Standards

This thesis builds a solution using a combination of two methodologies, the MDSD approach coupled together with an agile development approach. To achieve the solution we intend we initially designed potential models using UML, this is also described in this section.

3.4.1 Unified Modeling Language (UML)

Our central goal was to design a meta-model of the RWL using Microsoft DSL Tools. Prior to that, we decided to statically model various constructs of the RWL using UML. It allowed us to depict and analyse potential relationships, hierarchies, types and constraints. Our approach involved analysing RWL examples from (Prism New Zealand, 2005) and identifying various relationships. We then progressively modeled these examples using UML. We continued to do this until we had a number of RWL constructs within the UML model. After modeling all the RWL examples from (Prism New Zealand, 2005) we noticed that what we potentially have is a meta-model of the RWL as it could potentially represent any instance (example) of RWL.

3.4.2 Model-Driven Software Development

The core of this thesis aims at increasing the level of abstraction for end-users to give them the ability to design Prism WIN reports with ease. The way the abstraction level can be elevated is with the use of models and using MDSD was the obvious choice.

This thesis tries to provide a model-centric approach to solve the problems described in Section 1.5. Although we do not explicitly deal with MDSD in its true essence as we do not have a CIM or PIM as

these details are abstracted away from us by the Microsoft DSL Tools (described in detail in Section 3.3.1). Two approaches were taken during the course of designing the models for this thesis. Although the next chapters of the thesis concentrate on the best approach (RWM Shell approach) let us look at all the approaches in some detail in this section.

3.4.2.1 Class Diagram Approach

It was decided in the earlier phases that the solution would be a five stage process:

1. Design a meta-meta-model structure using Microsoft DSL Tools
2. Design our RWL meta-model using the newly created meta-meta-model
3. Generate code representing our meta-model using text templates
4. Design the end-user UI using WPF which uses the generated code representing the meta-model as its back-end
5. The WPF UI generates the required RWL script

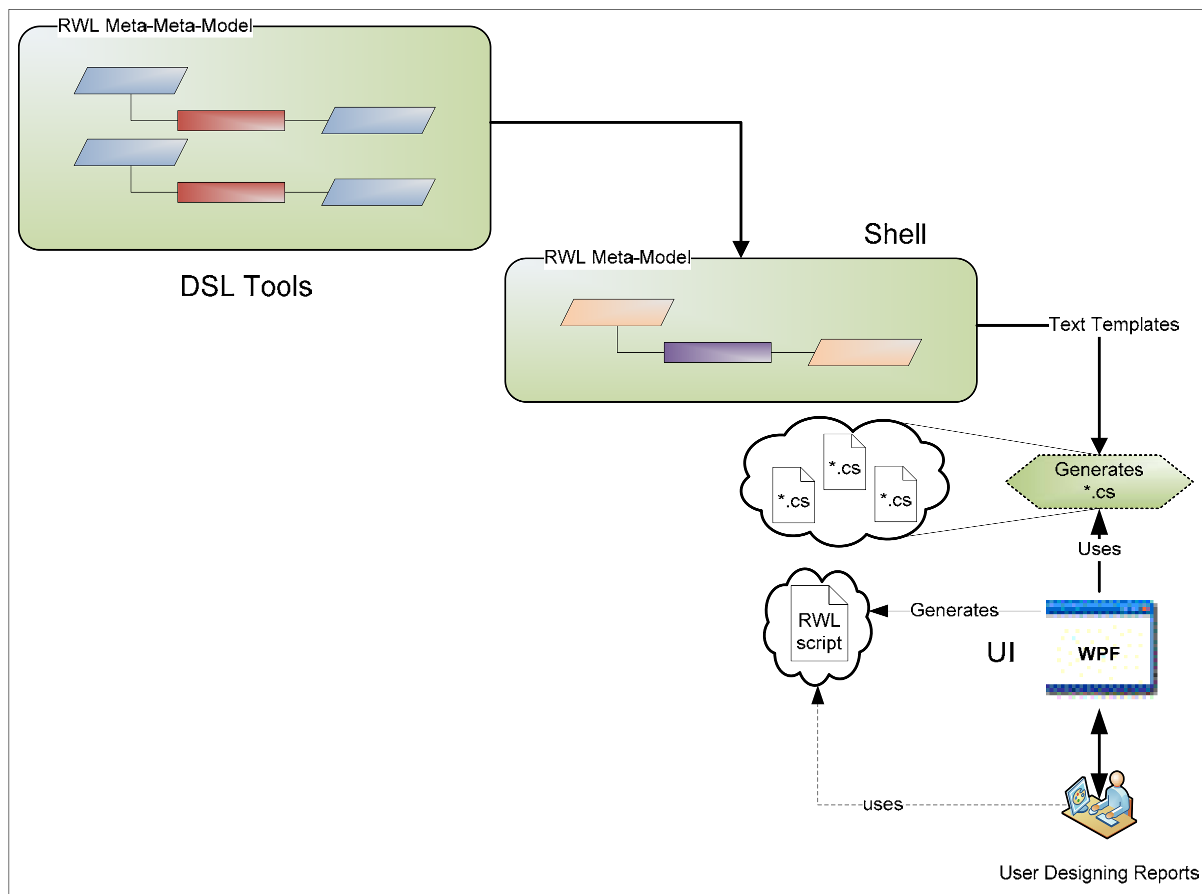


Figure 46: Class Diagram Approach

The figure above shows the five stages of this approach. The essence of this approach is the WPF UI layer which gives us customizability and a rich user experience.

The meta-meta-model represented in the Microsoft DSL Tools is a highly customised model very similar to that of the meta-model of UML. This enabled us to design a UML like model in the shell which could generate code representing the meta-model of the RWL eventually used by the user via

the WPF UI. The meta-model of the RWL will then generate the RWL script as per the users request against the RWL model which they create using the UI.

3.4.2.2 RWM Shell Approach

After spending some time on the class diagram approach mentioned in the previous sub-section it became apparent that it would potentially require a lot of resources in terms of programming and time to create a demonstrable prototype. Therefore it was decided that we cut down on one phase of the development. The approach was similar to the one shown in Figure 46 although the WPF layer was omitted. An alternative UI layer was used instead of the WPF layer. This UI layer is called the Visual Studio Isolated Shell, thus the name of the approach.

The Visual Studio Shell is a streamlined version of the Visual Studio IDE which can host Visual Studio Extensions such as models created using the DSL Tools (Microsoft, 2008). The shell behaves exactly like the Experimental Hive (Section 2.5.1.4) but the only difference is that end-users do not need Visual Studio installed. Also, the Visual Studio Shell can be freely redistributed and can be used royalty free for applications designed to be run on the shell.

Not using the WPF layer essentially meant that all other stages had to be pushed up one level. This meant we could use the text templates to generate actual RWL script from the models which the user designed. This is shown below in Figure 47.

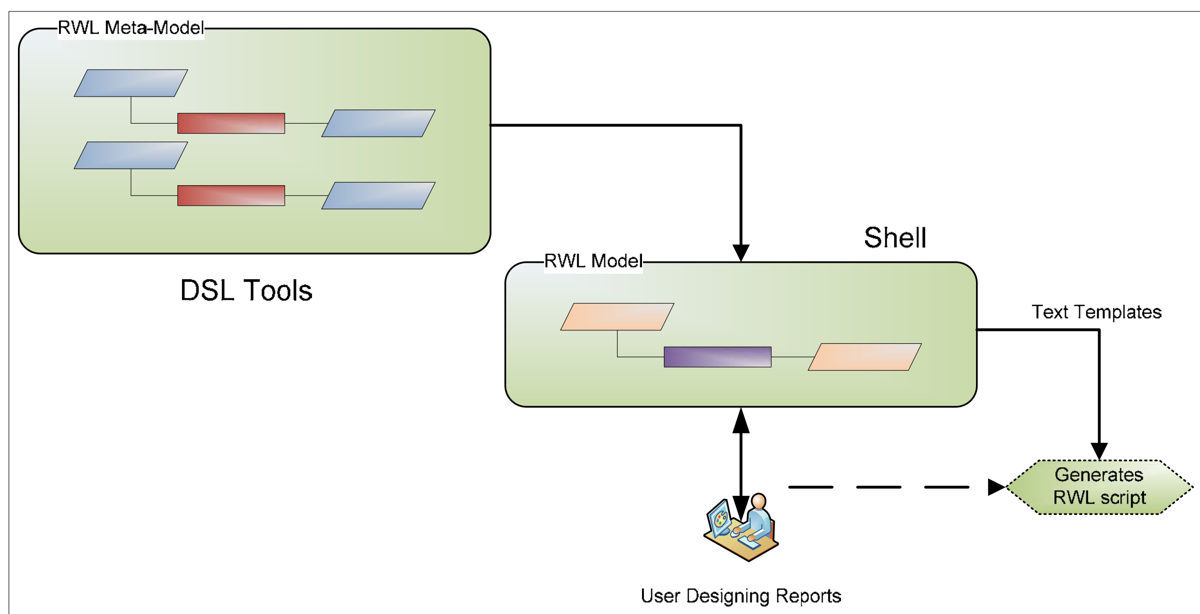


Figure 47: RWM Shell Approach

The RWM Shell Approach can be looked at as a three stage process:

1. Design the RWL meta-model structure using Microsoft DSL Tools
2. End-user designs the RWL model using the meta-model using the shell
3. Use text templates to generate the corresponding RWL script from the RWL model

3.4.3 Agile development

We use an agile development approach throughout the course of the thesis. We design the meta-model and the models progressively working upwards in terms of complexity. Continuous user feedback is received from both, the developers who are the potential designers and users of the meta-models and the end-users who will be designing models of the RWL using the meta-model. This development processes allows to use a highly iterative development style which involves designing new meta-model elements, prototyping those into a UI, evaluating them and refining them.

3.5 Gathering Requirements

Gathering the requirements for the solution is an ongoing process and as any software system, the requirements for this thesis may change as the solution is adopted by Prism for further development. The requirements may also deviate by a trivial amount as we collect and analyze more information about the RWL. Note that as we adopt MDSD along with an agile development approach the thesis strives to keep the requirements flexible to allow for slight deviations and other user requests as the solution grows. The following two sub-sections outline the approach taken to gather the requirements for the model and the end-user tool.

We have made a distinction in the two different ways of requirement gathering as we concentrate on two distinct goals in this thesis. The first goal is the obvious one, a tool for end-users to easily design Prism reports and the second goal is slightly subtle although an important aspect of the thesis. As we are designing a meta-model for a programming language (RWL) it is important that we make this meta-model flexible so it can be expanded if need be, this is the our second goal.

3.5.1 Tool Requirements

The requirements for the tool were gathered with the end-user in mind. Even though the primary users of the tool would be Prism customers, it is safe to assume that the “end-user” set also includes Prism employees who can potentially design reports. Due to time constraints imposed by the thesis it was not feasible to involve live Prism customers during the requirements gathering process, thus the requirements were congregated together with the help of Prism employees.

3.5.2 Meta-Model Requirements

The requirements for the meta-model were gathered from the perspective of future development. By this we mean that the meta-model should be able to withstand future modifications by new developers as the RWL expands and changes. Various aspects of the RWL meta-model were analyzed and requirements were formatted on how to cater for them.

3.6 Summary

We looked at an overview at the approach taken by the thesis to solve an enterprise level problem like report writing. We established four core steps in our approach which indirectly involved MDSD and agile software development processes.

We used Microsoft DSL Tools to design the RWL meta-model and the text templating engine to generate code. The solution also used other secondary technologies such as WPF and LINQ.

The chapter eventually described how the requirements were gathered by sub-dividing them with respect to the meta-model and the end-user tool.

Chapter 4 - Requirements

4.1 Introduction

This chapter contains a detailed list of requirements for this thesis. How these requirements were gathered was described in brief in the preceding chapter (Chapter 3). The chapter divides the requirements up into two major sections: Functional and Non-Functional. Functional requirements are defined as actions the system needs to perform in order to meet user needs as opposed to non-functional requirements which support a user while performing those actions.

The requirements are further grouped into two areas: Meta-Model requirements and Shell requirements. As introduced briefly in Chapter 3 meta-model requirements are developer-centric whereas the shell requirements are end-user-centric.

4.2 Functional Requirements

The core functional requirement for the thesis is to provide end-users with a tool which gives them the capability to design Prism reports with ease as opposed to the current practise of textually designing reports. This sub-section divides this core requirement into subordinate requirements which give us the ability to examine and cater for them independently.

4.2.1 Report Writer Meta-Model

4.2.1.1 Correctly represent RWL

The meta-model of the RWL should correctly represent the language and all its semantics. It is mandatory that the meta-model of the RWL is correctly represented by our visual language so that the correct RWL script can be generated by the tool. The meta-model would also represent constraints and should give users assistance while using the RWL models. Correctly representing the RWL via the meta-model would also mean that Microsoft DSL Tools generate the correct domain model code and developer intervention is minimized. The correct meta-model would also allow us to generate the RWL script from a given RWL model without any end-user input.

Example:

```
1.  Code      RW_EXAMPLE
2.  Name      “Report Writer Example”
3.  Type      Standard
4.  Access    STSR
```

The code above shows the ControlLine construct of the RWL. This was examined in Section 1.4. (Prism New Zealand, 2005) states that a given report should have one *ControlLine* construct at all times. Therefore, if a meta-model had to enforce this constraint we would have the following figure:

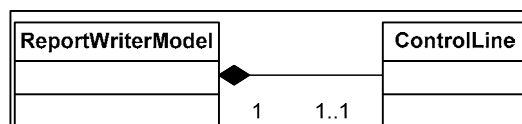


Figure 48: Meta-model representing RWL

The above figure shows us the *ReportWriterModel* which represents the root of our RWL model and the *ControlLine* which represents the control line construct in our RWL. The relationship between

them shows us that the *ReportWriterModel* contains a ControlLine construct. Moreover it also states that that the *ReportWriterModel* should contain one and only one *ControlLine* construct.

4.2.1.2 Constraint Validation

Constraints required by the semantics of the RWL should be validated by the meta-model. RWL imposes many constraints on its user; these constraints should be mapped within the meta-model. Any potential model designed against this meta-model should respect these constraints. If it is not possible for the meta-model to represent these constraints then custom code needs to be written by the DSL designer which does the necessary validation.

Constraints can be of three types as explained in Section 3.3.1.3. Hard constraints should be represented by the meta-model and soft and dynamic constraints should be represented by custom code. The required constraint validator should be executed on the meta-model when a validation request is made by the user via the shell.

Section 3.3.1.3 explains why soft and dynamic constraints cannot be represented by the meta-model. Even if we use custom code to represent these constraints, these constraints are essentially part of the meta-model we design.

Example:

An example of a hard constraint was already represented by Figure 48. An example of a soft constraint is shown below in Figure 49.

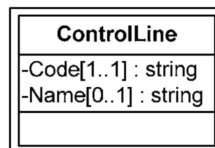


Figure 49: RWL soft constraint

(Prism New Zealand, 2005) states that the attribute *Code* is a mandatory field, thus denoted by the multiplicity of *1..1*. It can be seen that the attribute *Name* is not mandatory and is denoted by *0..1*.

An example of a dynamic constraint is represented by the following RWL code snippet:

```
1.  Scan RM
2.      Print RM_NAME;
3.  End
```

A scan is conducted on view *RM*, thus only columns contained within that view can be printed within its scope, and this is a dynamic constraint as the view on which the scan is executed on is not known till runtime (user entered value).

4.2.1.3 Code generator

Any code generator written should output correct code. This is especially important for the RWL code generator. It is the responsibility of the code-generator to correctly utilize the meta-model to generate required code which represents the instantiated models.

The code generator which outputs RWL should output correct RWL which can be interpreted by the Prism WIN MIS system. Any RWL generated should be readable and formatted in a way which promotes usability.

Example:

Generated code should be the correct and ideally also be formatted in a way which promotes easy reading by end-user.

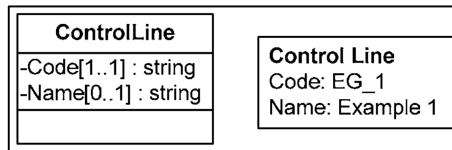


Figure 50: RWL meta-model and its instantiated model

Figure 50 above shows the meta-model representation of the *ControlLine* construct and an instantiated model with some sample values.

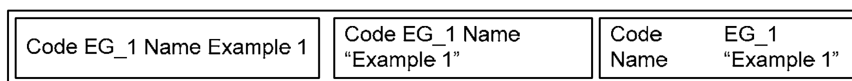


Figure 51: Difference in generated RWL

Figure 51 shows the generated RWL code for the instantiated model shown in Figure 50. The left hand side of this figure shows the wrong generated code, note the missing quotations mark for the field *Name*. The centre shows the correct RWL although formatted in a way which makes reading difficult and the right hand side of the figure shows correct and well-formatted RWL.

4.2.1.4 Mechanism to specify field editors and meta-data access

The RWL is a complex programming language and not all of its fields are trivial user input values. Some values need to match values within the Prism database meta-data. As this meta-data can contain a huge amount of data most of it being superfluous to the user it is necessary for us to provide smart editors for these kind of fields.

It is also required that these smart editors give the user some notion of what the meta-data represents. For e.g. simply showing the user a list of columns within a table would not be sufficient, although this list coupled with the description of each column would suffice.

The Prism WIN MIS database maintains its own meta-data as outlined in Section 1.5.1. This meta-data needs to be accessed by the meta-model of the RWL in order to give users access to information about the Prism database. Another requirement arises that this data is accessed atomically and is always kept in a consistent state.

Example:

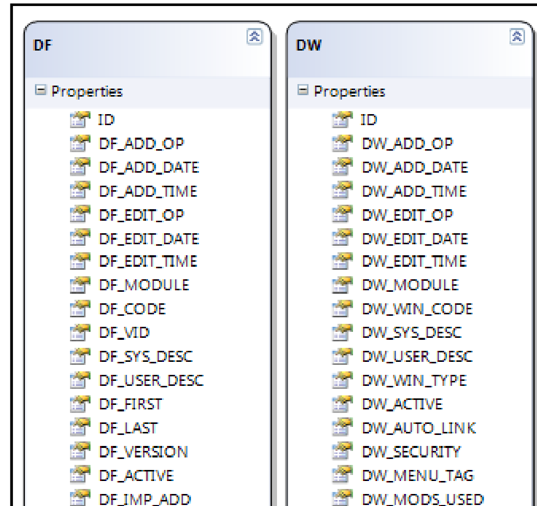


Figure 52: Prism meta-data section

Views DF and DW holds all the tables and the windows[2] that exist in the Prism database respectively. These views are shown in Figure 52 along with some of the columns they contain.

Getting information from DF would inform RWL users about the various views that can be scanned. For e.g. the *Scan* statement shown in the RWL snippet below shows a scan being executed on view RM.

```
1.   Scan RM
2.       Print RM_NAME;
3.   End
```

Information from the DW is important for the RWL user as it could be used to populate fields like *Access* as shown in the code RWL snippet below.

```
1.   Code        RW_EXAMPLE
2.   Name        “Report Writer Example”
3.   Type        Standard
4.   Access      STSR
```

Note that the field *Access* represents the windows[2] via which this particular report could be executed from.

Also, simply showing the user a list of the views or windows that exist within the Prism database will not suffice. A user interface needs to be developed which provides users with more detailed information. An interface example is shown below in Figure 53.

[2] Windows in the Prism WIN MIS system represent various menu combinations which open sub-interfaces where each of them encapsulates a different printing function. For e.g. window STSR represents the sub-interface System→Toolbox→Scripts→Reports accessed via ALT+S+T+S+R.

DF_CODE	DF_SYS_DESC
CA	CB Statement file
CB	CB Batch file
CC	CB Control file
CD	CB Dissection file
CL	CB Line file
CM	CB Bank masterfile
CS	CB Summary figure file
CT	CB Transaction file
DA	DD Audit number file
DB	DD Command button file
DC	DD Compound key definition file
DD	DD Data field definition file
DF	DD File definition file

Figure 53: UI example showing Prism meta-data information

In the above figure we see that the meta-data of the Prism database often contains a code and a description. The code tends to be a system specified combination of letters which often have little to no meaning to end-users, although if this is coupled with the description, it makes it the code more meaningful. Therefore, showing the code along with its description is ideal for improving the usability of the RWL tool.

4.2.1.5 Mechanism to specify mandatory fields

Mandatory fields are fields which always need an input from the user. This is essentially a constraint although the requirement is that we have a simple and elegant way of specifying which fields in a given construct are mandatory. The developer working on the meta-model should be able to add this constraint to any field for any construct.

Some degree of automation should be provided for the developer in terms of validation, for e.g. it would be best if the developer could mark a given field on a construct as *Mandatory* and the meta-model could automatically check if the user had entered a value for this field in the instantiated model via the shell.

Example:
The *ControlLine* construct contains three mandatory fields; these are shown in the code snippet below:

```
1.  Code      RW_EXAMPLE
2.  Type      Standard
3.  Access    STSR
```

Note that *Code*, *Type* and *Access* are three mandatory fields within the *ControlLine* construct. Therefore, the meta-model could use a similar approach to that shown in Figure 49.

4.2.1.6 Extension Points

The RWL is constantly evolving as end-user requirements change. Due to this reason the meta-model should be flexible enough to cater for these changes. Therefore, it is the responsibility of the DSL designer to include extension points within the meta-model which would assist future development. It is not required that the DSL designer caters for all future enhancements as it is impossible to do so, although a framework or a guideline on making these future enhancements is needed.

Example:
An example of an extension point could be a valid hierarchical architecture of the meta-model which would allow easy addition or modification of new and old RWL constructs.

4.2.1.7 Versioning

Versioning is a direct consequence of the requirement outlined in 4.2.1.6. If extension points allow further meta-model development it is required that a versioning scheme is also deployed. The versioning scheme should guarantee that a model designed with an older version of the meta-model will function as required with a newer modified version of the meta-model. This may always not be possible to guarantee, at that juncture the user should be explicitly informed about the changes within the meta-model and given directions on how to rectify their model.

We could go one step further and provide transformation services at times when the meta-model changes drastically and it becomes difficult for the end-user to perform the required changes within their model. The transformation service should be able take as input the old and new meta-model and the end-user model and transform it to meet the specifications of the new meta-model.

This case may arise when a developer changes the structure of the meta-model by adding new RWL constructs or modifying existing ones and also when adding or modifying relationships between meta-model elements.

Example:
An example of a possible transformation service is shown below in Figure 54.

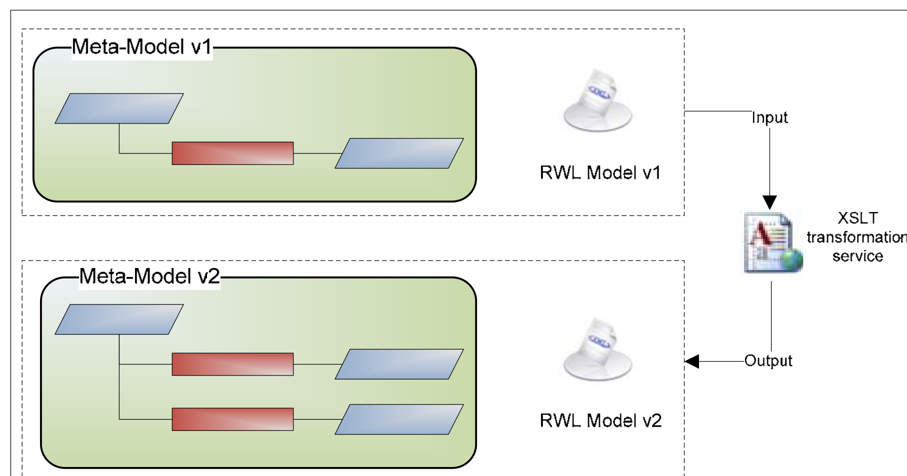


Figure 54: Example transformation service

4.2.1.8 Explorer view

As the RWL is naturally hierarchical it can be easily visualized using a tree-like structure, especially if it is represented using visual elements. The explorer view should aid the user in seeing an overview of the report they have designed. The explorer view should be interactive and let the user add information to the modeled report if required. What information can be added or modified within the explorer view is dictated by the meta-model. Also, the explorer view should display meaningful information about each RWL construct which would help the user uniquely identify them on the design surface.

Example:
Consider the RWL script shown below:

```
1.  Code      RW_EXAMPLE
2.  Type      Standard
3.  Access    STSR
4.
5.  PageHeader
6.      Print StandardPageHeader;
7.  End
8.
9.  Scan RM
10.     Print RM_CUST;
11. End
12.
13. Print StandardReportFooter;
```

The above code snippet can also be visualized in a tree-form shown below in Figure 55.

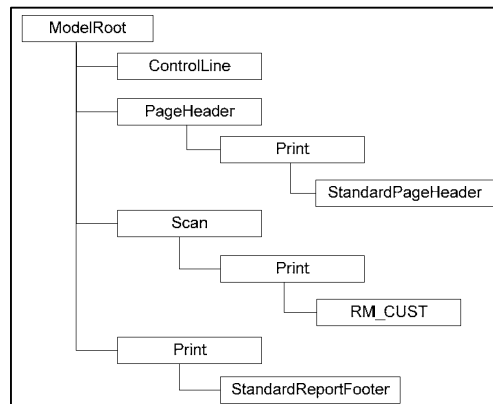


Figure 55: Tree view representation of RWL snippet

4.2.2 RWM Shell Host

4.2.2.1 Visually design reports

RWL is a language which describes dependencies between RWL constructs and may also contain nested structures. A visual representation would be able to explicitly show these properties which is not entirely possible textually. Therefore, the shell should provide users with a set of shapes and connectors with which they can design RWL scripts. The visual elements provided by the shell should have the appropriate RWL meta-model elements in the background. Each shape and connector dragged onto the shell canvas by the end-user should instantiate the appropriate meta-model element.

Example:
Consider the RWL script shown below:

```
1.  // Some comment
2.  Code      RW_EXAMPLE_03
3.  Type      Standard
4.  Access    STSR
5.
6.  Print StandardPageHeader;
7.  Scan RM
8.      Print RM_CUST + RM_NAME;
9.  End
10. Print StandardReportFooter;
```

This script could be represented visually using the notation shown below in Figure 56.

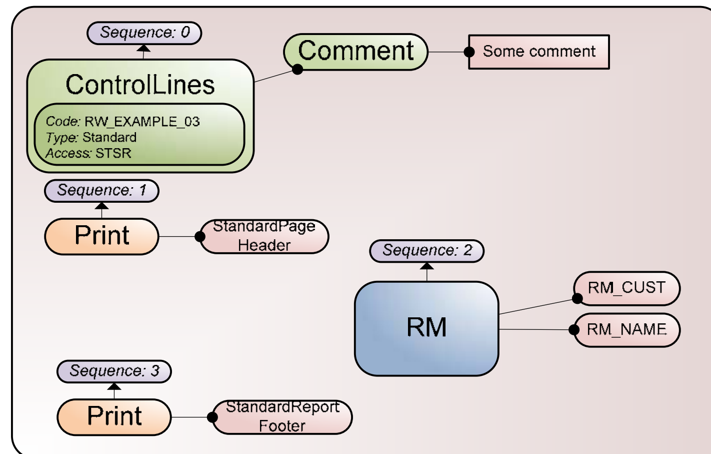


Figure 56: Example of a visualized RWL script

The sequence number in the above figure maps to the order in which each element occurs in the textual representation of the script. Note that *RM* represents a scan construct implicitly. This figure is simply an example of the various possibilities of representing a RWL script visually.

4.2.2.2 Visual notation

Selecting the appropriate visual notation is a formal requirement. The visual notation should be consistent and should give users a sense of the underlying RWL construct. Relationships between constructs should be made clear and if any hidden dependency exists, users should be given an opportunity to explicitly examine them if need be.

Designing the visual notation is more of an art and usually takes numerous iterations of design and implementation to develop. We took two approaches to this problem which is explained in the next chapter.

Example:

Figure 56 shows an example of a consistent visual notation for the RWL model. Note that all items that are printed are encapsulated within the same shape with the same size and colour. Although this example does not provide an intuitive notation as the shapes used do not necessarily depict the underlying RWL construct.

Figure 57 below shows a *Scan* construct on view *RM*. Note that a scan is nothing but a select statement (Section 1.4.1). Therefore a possibly more intuitive way of representing the *Scan* construct could be with the use of layering. Layering would suggest that this construct can potential yield more than one result.

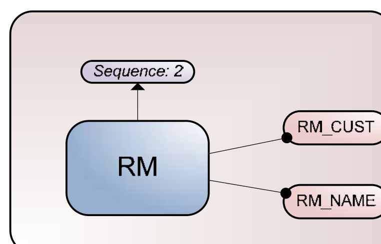


Figure 57: Example of an intuitive RWL concept

4.2.2.3 Constraint validation

Any constraints enforced by the RWL semantics should be enforced by the shell via the meta-model. Users should be informed while dragging and dropping shapes and connectors whether they are violating these constraints. At any point if this is not possible then a "lazy" constraint validation should be done. A "lazy" constraint validation may allow a potentially illegal RWL model although it will inform the users when an explicit validation request is made.

Users should be informed either with an error or a warning depending on the strictness of the constraint. A validation request should always be executed when the user saves or opens the RWL model as saving and opening an invalid model may cause system errors if the meta-model cannot read this file. Constraint validation errors should be meaningful and give users appropriate detail to rectify them.

Example:

A RWL model can only contain one *ControlLine* construct. If the user tries to add another *ControlineLine* construct to the model, this should be disallowed by the meta-model via the user interface. A potential way of enforcing this constraint and its corresponding feedback message is shown below in Figure 58.

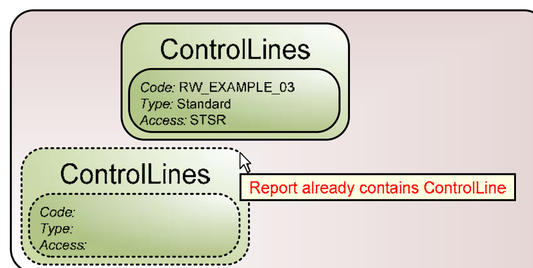


Figure 58: Shell constraint validation

The user should be informed if a *ControlLine* construct is missing any of its three mandatory fields, namely *Code*, *Type* and *Access* as introduced in Section 4.2.1.5. Although this falls under the "lazy" validation category as the user will only be informed about this if a save is attempted or an explicit validation request made. This is shown below in Figure 59.

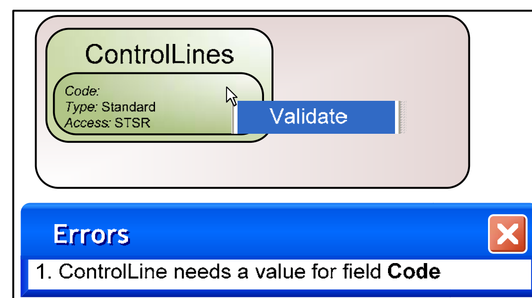


Figure 59: Shell "lazy" constraint validation

A set of RWL constraints is explained in detail in Section 5.4. Note that this section does not include every possible RWL constraint, only includes the common constraints which we have managed to implement in the given time period.

4.2.2.4 Show program flow

As RWL is a sequential programming language it is required that any visual language depicting this behaviour visualizes this sequential program flow. The program flow is governed by the user designing the report therefore an appropriate mechanism needs to be provided which could allow them to define and visualize this program flow.

<u>Example:</u>

Consider the RWL script shown below:

```
1.  Code      RW_EXAMPLE_03
2.  Type      Standard
3.  Access    STSR
4.
5.  Print StandardPageHeader;
```

A possible visual representation of the above RWL script is shown below in Figure 60.

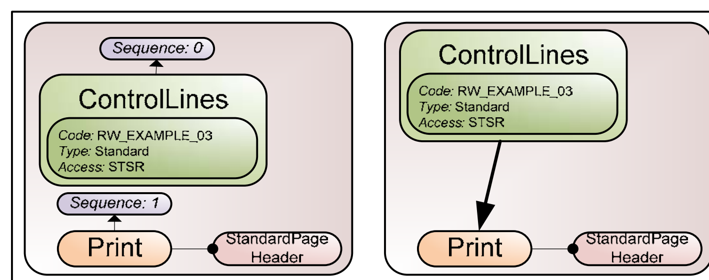


Figure 60: Showing RWL program flow

Shown in the above figure are two examples of representing program flow within a RWL model. The left hand side of the figure shows a visual-text numbering scheme to represent flow as opposed to the right hand side which shows an arrow head in the direction of the flow. The start state in both model examples is obvious. In the left side of the figure the element with smallest number will be the first executed element and in the right side of the figure the first element will be the one which does not have any incoming connections.

4.2.2.5 Containment of child elements

Some constructs in the RWL can contain other constructs. Such composite constructs should be represented by the visual language using a containment mechanism which will assist the readability of the models.

<u>Example:</u>

Consider the RWL code snippet shown below:

```
1.  Scan RM
2.      Print RM_CUST + RM_NAME;
3.  End
```

From the script it can be seen that the *Scan* does a select statement (Section 1.4.1) on the view *RM* and within that it *Prints* whatever data occurs in the column *RM_CUST* and *RM_NAME*. Therefore it can be seen that *Scan* is a composite construct, in this case containing a single child construct *Print*. An example of representing this containment using a visual language is shown below in Figure 61.

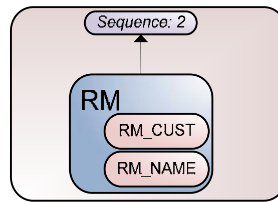


Figure 61: RWL composite construct notation example

4.2.2.6 Ordering of child elements

The requirement for containment was introduced in Section 4.2.2.5 which gives rise to the requirement for ordering child elements. A composite RWL construct can contain other RWL constructs, although these constructs are ordered due to the sequential nature of the RWL. Due to this fact, it is required that the sequential nature of the children of composite constructs should be visualized with some ordering mechanism. This would allow the end-user to visualize the particular order in which the composite construct will execute its children.

Example:
Consider the RWL code snippet shown below:

```
1.   Scan RM
2.         Print RM_CUST + RM_NAME;
3.   End
```

Looking at the above script we can notice how the print statement would initially print data from column *RM_CUST* followed by data from column *RM_NAME*. This ordering needs to be conveyed by the visual notation. Looking at Figure 61 it can be clearly seen that *RM_CUST* occurs before *RM_NAME*. A general assumption has been made here that the user would be reading from top going down.

4.2.2.7 Show/hide children (Elision)

This requirement arises as a direct repercussion of the requirements outlined in Section 4.2.2.5 and 4.2.2.6. A mechanism needs to be added to a composite construct which would allow users to show and hide its children. This would allow the RWL model to be concise and tidy and would also allow the end-user to concentrate on certain aspects of the report if need be. Showing and hiding children of a composite construct could potentially raise the level of abstraction for users who do not care about the details of its children.

Example:
Consider the RWL code snippet shown below:

```
1.   Scan RM
2.         Print RM_CUST + RM_NAME;
3.         Print "Hello World";
4.   End
```

It can be seen that the *Scan* construct now has two child *Print* constructs. This can be represented as shown below in Figure 62.

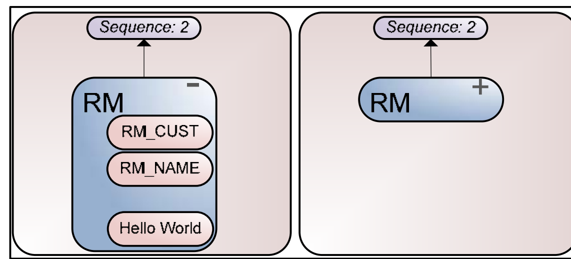


Figure 62: Show/Hide child constructs notation example

From the left hand side of the above figure we can see the *Scan* construct in its expanded form which exposes all its children. Note the "-" symbol on top-left side of the shape denoting that the shape can be collapsed. The right hand side of the figure shows the same *Scan* construct although in a collapsed position with its children hidden. Note the "+" symbol on the top-left side of the shape denoting the shape can be expanded.

4.2.2.8 Context sensitive assistance

As the RWL is a complex language it is required that the shell provides some degree of assistance to users. The assistance provided by the users should be context sensitive so that it does not add more complexity to the language. Assistance could be provided to the users via tooltips, pre-populated drop down fields, messages or by any other visually intuitive manner.

Example:

A *Scan* can essentially be executed on any valid view which exists within the Prism database. There are approximately 350 views within this database and sorting through them to find the desired view would be cumbersome and time consuming for a given user. This assumes that the user actually knows which view they want to execute the *Scan* on, if the user does not have a clear idea on the desired view then this task would be extremely difficult.

The above mentioned problem could be solved by simply allowing the user to select a view from a drop down field which contained all the views within the Prism database. This could be further refined by showing more detailed information about the view, such as its description. An example of this is shown in Figure 53.

Note that this example refers to the *Scan* construct; therefore the context of this view selector would be the *Scan* construct. Other constructs could offer similar assistance.

4.2.2.9 Evaluation

The visual language notation represented within the shell needs to be constantly evaluated against our user needs. This would assist the developer in providing the correct visual cue for various RWL constructs. Moreover, as new RWL elements are added to the meta-model, new notations will be added, therefore the evaluation of this notation is an ongoing requirement.

User evaluation could be informal which would encompass basic demonstration and user interaction or it could be formal using surveys and other evaluation techniques as discussed in Section 2.4.3.6.

An informal evaluation could be done between minor[3] releases and formal evaluations could be done between major releases.

We have included evaluation as a functional requirement because we are creating a visual language which will be used by end-users to design Prism reports. Due to this evaluation is one of the core aspects in making a visual language intuitive and user friendly, making it an important functional requirement.

4.2.2.10 Test cases

The visual language developed as part of the shell has to be run against some set test cases. For the purpose of this thesis we can extract those test cases from (Prism New Zealand, 2005). A test case could essentially be a test of basic report scripts which are created using the RWL model. Therefore, whenever a change is made to the meta-model or to the notation, these tests could be rerun and their generated RWL analyzed.

Test cases could be further refined and automated as need be although this is not a strict requirement. A test case could potentially be a RWL model which can be evaluated against the current meta-model and the generated RWL script could be automatically tested against the Prism WIN report writer engine as shown in Figure 3.

An infrastructure needs to be implemented which allows for such test cases to be run against the meta-model to test its validity. This infrastructure should be able to execute single or multiple test cases and display a result indicating failure or success.

4.3 Non-functional Requirements

The core non-functional requirements for both, the meta-model and the shell are to provide an intuitive interface to the developers and the end-users respectively. This core non-functional requirement, like functional requirements, is divided up into two subordinate requirements allowing us to analyze them independently.

4.3.1 Report Writer Meta-Model

4.3.1.1 Intuitive Extension Points

Extension points provided within the RWL meta-model need to be intuitive. They need to expose a clear architecture and intent so that future developers can expand the meta-model with newer RWL constructs if need be.

<u>*Example:*</u>

Consider the RWL code snippet shown below:

```
1.  Scan RM
2.      Print RM_CUST + RM_NAME;
3.      Print “Hello World”;
4.  End
```

It can be seen that a *Print* construct can print column values as well as literals. So a potential hierarchal architecture would be similar to that shown in below in Figure 63.

---

[3] A minor release, as opposed to a major release, would essentially mean a meta-model change where a transformation service (Section 4.2.1.7) is not required. Therefore, the user can safely transfer their models across different meta-model versions.

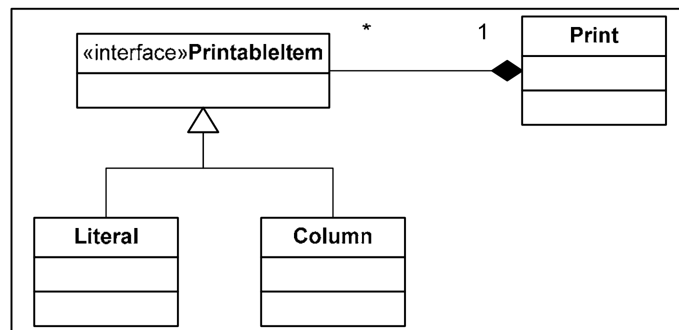


Figure 63: Intuitive RWL meta-model extension point

This architecture would make it clear that a *Print* construct can contain a number of *PrintableItems*. Therefore in the future if another construct gets added to the RWL which can potentially be printed then simply inheriting from the *PrintableItem* interface would suffice.

4.3.1.2 Modularity

The meta-model of the RWL needs to be modular and loosely coupled with other parts of the system so that changes in one part of the system will not affect other parts of the system. This requirement arises as our RWL meta-model is constantly evolving and constant change is required. Also, each RWL construct and each relationship between these constructs need to be represented by a single meta-model element. This would further add modularity to the meta-model and allow any further development to be isolated to a particular meta-model section of interest.

Example:

The meta-model exposes the Prism database and its meta-data to give users assistance while designing the RWL model. Therefore it is required that the meta-model and the data access layer be kept disparate from each other. Moreover, each meta-model element has to be isolated from its counterparts. This can be seen in Figure 63 where the *Print* constructs is isolated from the actual items that can be printed via the *PrintableItem* interface.

4.3.1.3 Scalable

The initial architecture of the meta-model should be designed with scalability in mind. As the RWL will keep growing, it is essential that the meta-model can sustain these changes.

Moreover the visual representation of the RWL also needs to be scalable. Complex RWL scripts span hundreds of lines so when these get translated to a visual notation, the meta-model should be able to clearly represent these and avoid clutter for easy readability. This requirement is detailed further in Section 4.2.2.7.

Example:

Designing the RWL meta-model like shown in Figure 63 will scale appropriately as the system grows as explained in Section 4.3.1.2. Any other sub-system, like the data access layer, also needs to be scalable, as more and more data gets exposed to the user via the meta-model. The meta-model and all its subordinate sub-systems should work in sync in aiding the user design the RWL model and should perform at a level where the user is not distracted away from this task.

4.3.1.4 Maintainable

This thesis essentially provides a framework for a meta-model for the RWL; therefore it is mandatory that this framework is maintainable by other developers. The meta-model should have a consistent naming scheme which promotes maintainability. The relationships between RWL constructs should clearly represent the participating constructs and should have unique distinguishing names.

Example:

Looking at Figure 63 we can notice the naming scheme followed by the meta-model. The names of the model elements closely match the actual names of the RWL constructs. This allows future developers to easily notice relationships and other association nested within the meta-model. Moreover the figure clearly represents via UML the relationship between a *Print* construct and its corresponding *PrintableItems*.

4.3.1.5 Robust

The meta-model provides an initiation point for all RWL models created thus it has to be robust enough to cater for any possible model and also to gracefully reject any end-user error. The meta-model also provides all the core elements for code generation and RWL script generation thus it has to be robust enough to cater for variations in the way end-users design RWL models.

Example:

Consider the RWL script shown below:

```
1.  Code      RW_EXAMPLE_03
2.  Type      Standard
3.  Access    STSR
4.
5.  PageHeader
6.      Print StandardPageHeader;
7.  End
```

The above script can be represented visually in two distinct ways, both shown below Figure 64.

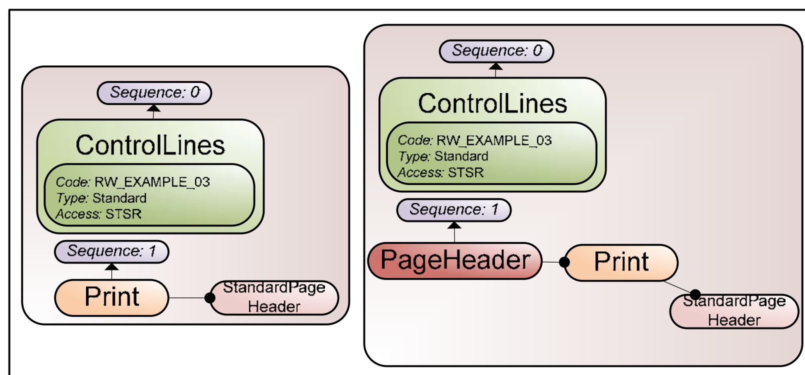


Figure 64: RWL model variations

Note that both these models ultimately represent the same report as the *PageHeader* construct is a grouping construct for displaying multiple elements on a report header. Although, in this particular example, the script only has one *Print* construct within the *PageHeader* therefore it can be pushed out of the grouping. Thus, the user can essentially use either approach and the meta-model should be able to cater for both and generate the appropriate RWL script.

4.3.1.6 Simplicity

The RWL meta-model has to be designed for simplicity for two main reasons. Firstly, it makes the code generator and the RWL script generator simple and secondly, it decreases the learning curve for new developers if the RWL meta-model needs to be expanded.

Example:

Looking at Figure 63 we can see that the way to go from a *Print* construct to its corresponding *PrintableItems* is simple and at most one level deep. Therefore, a simple implementation of this meta-model in an object oriented language is shown below in Figure 65.

Figure 65: Simple implementation of the RWL meta-model

Thus, if the RWL code generator needs to access all *PrintableItems* associated to a particular *Print* constructs, this is simply possible via the property *PrintableItem:List<PrintableItem>* shown in the figure above.

4.3.2 RWM Shell Host

4.3.2.1 Intuitive

The shell has to be intuitive to allow users to concentrate on the task at hand which is design a RWL model. The notation exposed by the meta-model also needs to be intuitive and should be consistent to allow seamless RWL model design.

Example:

A potential intuitive shell interface is shown below in Figure 66.

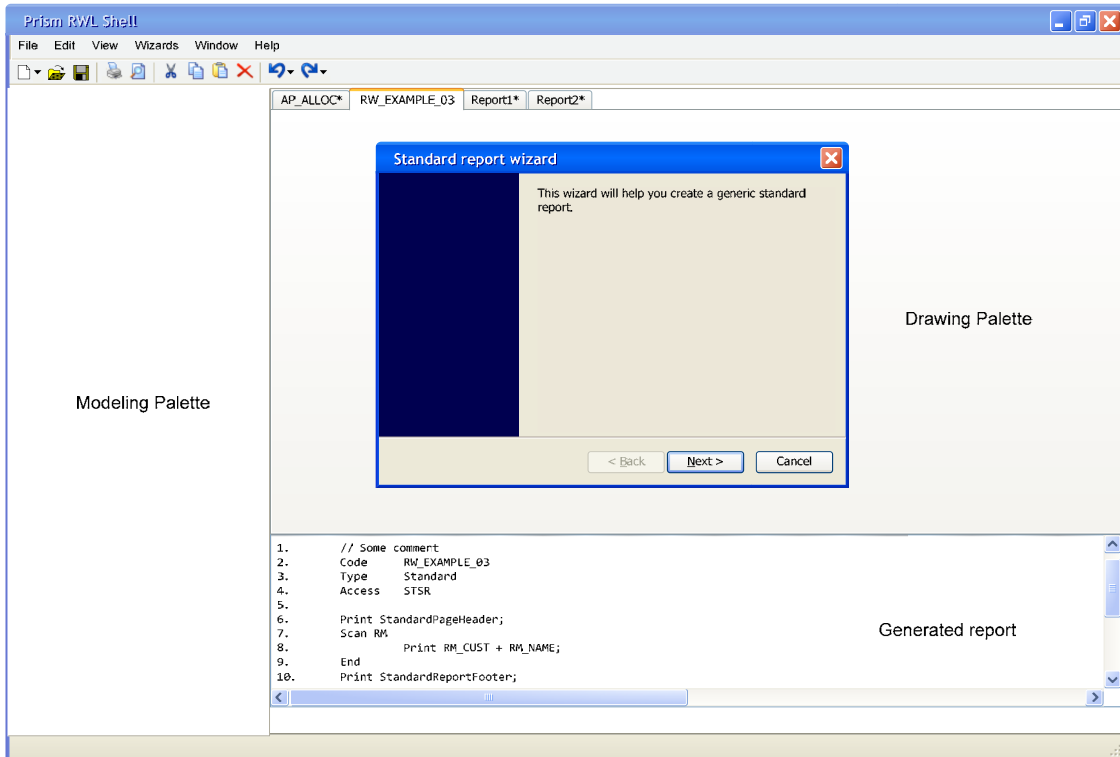


Figure 66: Sample shell interface

Note in the above figure how we have separate areas where the user can find the models, a drawing area (RWL model design surface) and an area where the generated report is shown. This interface is simple to use as it contains a simple left to right flow where the user starts dragging the model and drops it on the draw palette. The window at the bottom of the palette shows the generated RWL script from the corresponding model. This generation can be triggered from the menu items or the toolbar. Therefore we can safely say that this simple user interface follows the same mental model as the user would have using modern day interfaces making this example UI intuitive.

4.3.2.2 Simplicity

Any user interface exposing the meta-model to the user needs to be simple. Because the shell will be defeating the complexity of the Prism database and that of the RWL it is essential that it does this with simplicity to lower the learning curve for the end-user as much as possible. Moreover the shell will have minimal to no documentation so it is vital that each aspect is designed with simplicity in mind.

Example:

The sample interface in Figure 66 shows a simple approach to designing the shell. It does not have any extra additions which would be confusing thus it making it simple for the user to design the RWL model. The interface has a simple left-to-right flow, finding the models on the left hand side and the RWL modeling surface to its right.

4.3.2.3 Robust

The shell has to be made robust enough to allow for user manipulation, even if it is erroneous. The shell should allow users to make any possible moves and if an error occurs, either at the meta-model

or at the UI level, the shell should inform the users of the error and fail gracefully. If possible and applicable the shell should also inform the user on how to rectify that error.

Example:
Consider the RWL script shown below:

```
1.  Code      RW_EXAMPLE_03
2.  Type      Standard
3.  Access    STSR
4.
5.  Print StandardPageHeader;
```

The above script is shown visually in the interface introduced in Figure 66. A variation of this interface is shown below in Figure 67.

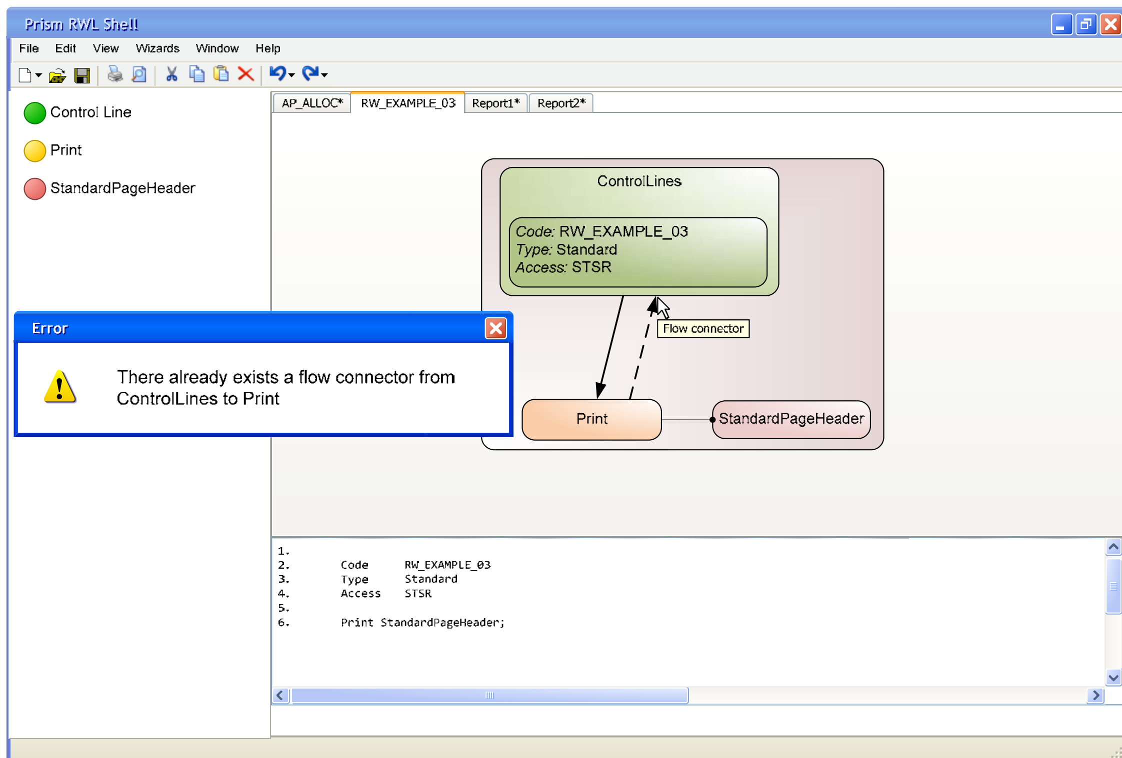


Figure 67: Sample shell interface with error detection

The figure above shows the user adding a flow connector from the *Print* construct to the *ControlLine* construct. This potentially violates the sequential nature of the RWL as there is already a flow connector existing in the other direction. The interface should gracefully accept this and inform the user about the violated constraint.

4.3.2.4 Self-explanatory icons (Closeness of Mapping)

Each icon which represents a given RWL construct has to be self-explanatory. This would allow users to realize what the icon represents and use it effectively and also decreasing the learning curve. Icons should also be consistent in terms of size and look and feel.

Others icons representing various features on the shell should also be intuitive and follow the user mental model.

Example:
Looking at Figure 67 we can see that we have the model toolbox on the left hand side of the interface, although at this stage each RWL construct is only represented by a coloured dot which is not ideal. These should be replaced with more intuitive icons, for e.g. a potential icon for the *Print* construct could be a printer icon.

Figure 67 shows other icons representing secondary shell features such as save, open, undo, redo, etc. Note how these icons are consistent with an already existing user mental model matching a modern operating system interface such as that of Microsoft Windows®.

4.3.2.5 Modular

The shell should be modular in terms of the meta-model components it exposes. It should have separate sections for the toolbox, canvas, property editors, toolbars and menus. Modularising the shell would allow users to direct their concentration on the task at hand.

Example:
Figure 67 shows an example of such a modular interface where each section or aspect of designing a RWL model are disparate. The toolbox is on the left hand side of the interface and the canvas to its right. Toolbars and menus are grouped together on the top of the interface.

4.3.2.6 Metaphors used

Visual notation exposed by the shell should potentially use appealing metaphors where possible. Metaphors aid users in thinking about the notation in terms of the RWL constructs. It allows the user to simply look at a given RWL model and know what it represents rather than looking at a mapping file which maps the notation to the RWL constructs. Care has to be taken that the metaphors used are not contradictory and do not hamper the users' perception of what the RWL model represents.

Example:
An example of a metaphor is given in Figure 57. If a metaphor is not feasible for a given construct we can use an UML like approach where a stereotype is used. So we could potentially have a notation like shown below in Figure 68.

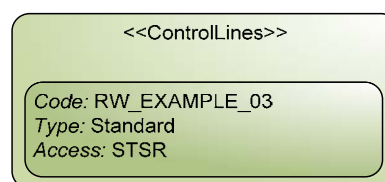


Figure 68: Notation using stereotyping

4.4 Summary

This chapter details the requirements for this thesis in terms of the meta-model which needs to be developed to represent the RWL and the shell user interface which will potentially host this meta-model. Therefore, the core requirement is to give end-users the ability to visually design RWL scripts. Other secondary requirements also highlighted within this chapter are the need to make the meta-model and the shell user friendly and intuitive for developers and end-users respectively.

Various examples are given to complement the requirements and help elaborate on the requirements.

Chapter 5 - Design

5.1 Introduction

This chapter details the design and analysis phase of the thesis research. It analyzes the requirements established in the previous chapter and details the design steps taken to cater for each of them. Similar to the requirements division, we divide this chapter into two main parts, design of the meta-model and the design of the shell which hosts this meta-model. The chapter also describes the architecture of the system along with details about the user interface design.

We use UML and a use-case modeling approach to highlight and justify the key design decisions made during the course of the meta-model and the RWL shell host design. Also note that we used two approaches during the course of the thesis which were introduced in Section 3.4.2, namely the Class Diagram Approach and the RWM Shell Approach. Therefore, in the following sections we will be analyzing the design decisions for each of these approaches independently.

5.2 Stakeholders

We have identified two main stakeholders which we have to consider while catering for the requirements outlined in Chapter 4. Each stakeholder has an aspect of the solution which they are primarily concerned about, therefore we have divided up the stakeholders with respect to the meta-model of the system and the shell host which will eventually host the meta-model.

5.2.1 Meta-Model

The key stakeholder with respect to the meta-model is the developer. The meta-model essentially builds a framework for a Prism developer to build future models and evidently improve the already existing RWL meta-model.

5.2.2 Shell Host

The key stakeholder with respect to the RWL shell host is anyone who designs Prism reports, essentially Prism customers. The key responsibility of the shell is to expose the meta-model of the RWL to the end-user.

5.3 Use Cases

Use cases were used to highlight our core functional requirements in terms of its stakeholders using a graphical notation. We will be using these use cases in Section 5.4 and Section 5.8 to highlight our design decisions.

5.3.1 Developer

The developer of the meta-model has the obvious function of designing the meta-model, although other core requirements were also identified. These are showing below in Figure 69.

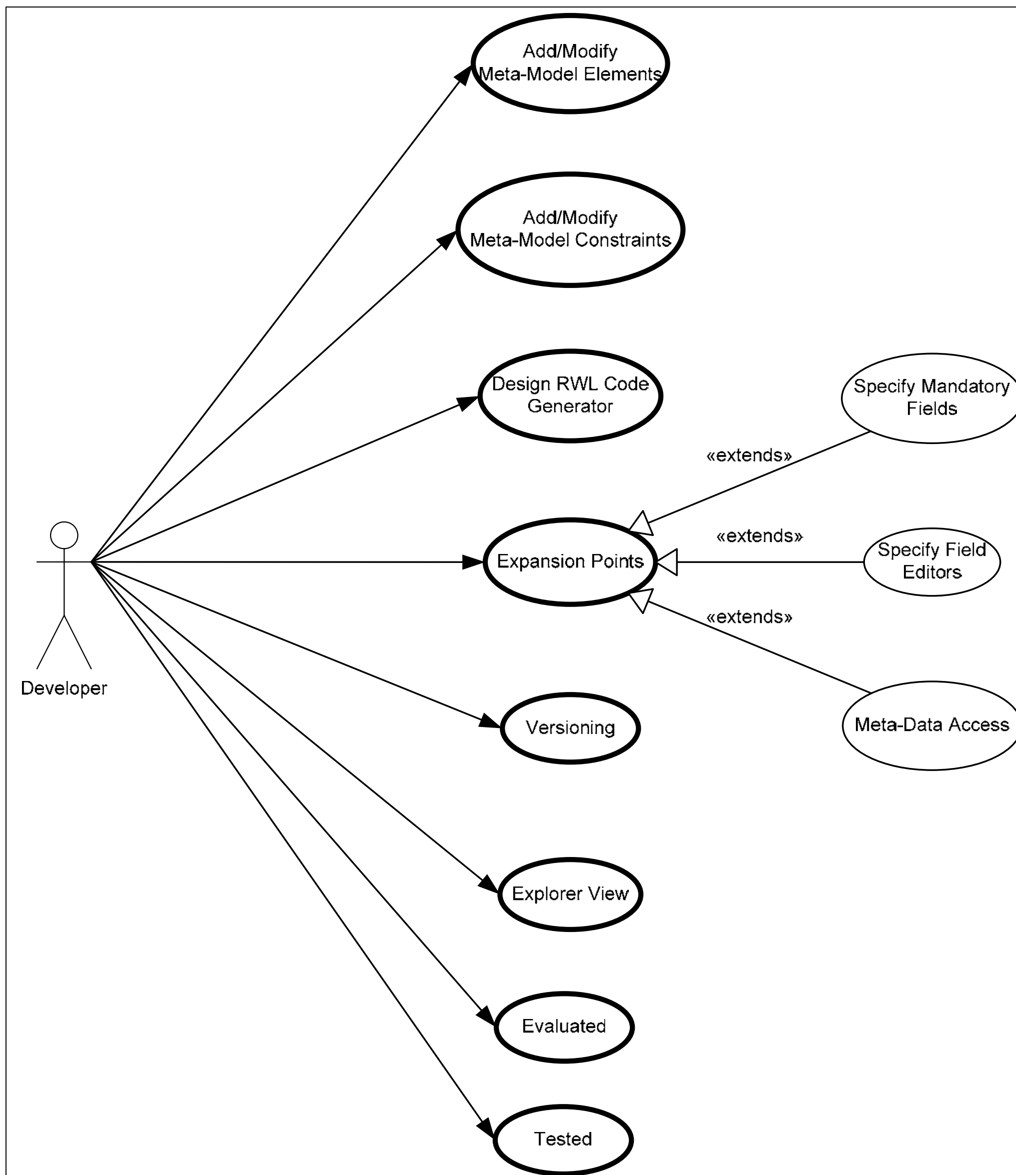


Figure 69: Use case developer perspective

Figure 69 identifies eight core functionalities (identified by bold ellipses) we have to cater for in terms of the developer. Other use cases identify secondary requirements which specializes a given primary requirement.

5.3.2 Report Designer

Any individual that designs a report will potentially be the end-user of the meta-model as they will use it to design RWL models using a standalone UI. Reports will be designed in essentially two ways depending on our approach. In the first approach the end-user uses a WPF UI to design RWL models and in the second approach they use the Visual Studio Shell. The core functions a report designer will be performing are encapsulated in the use case diagram shown below in Figure 70.

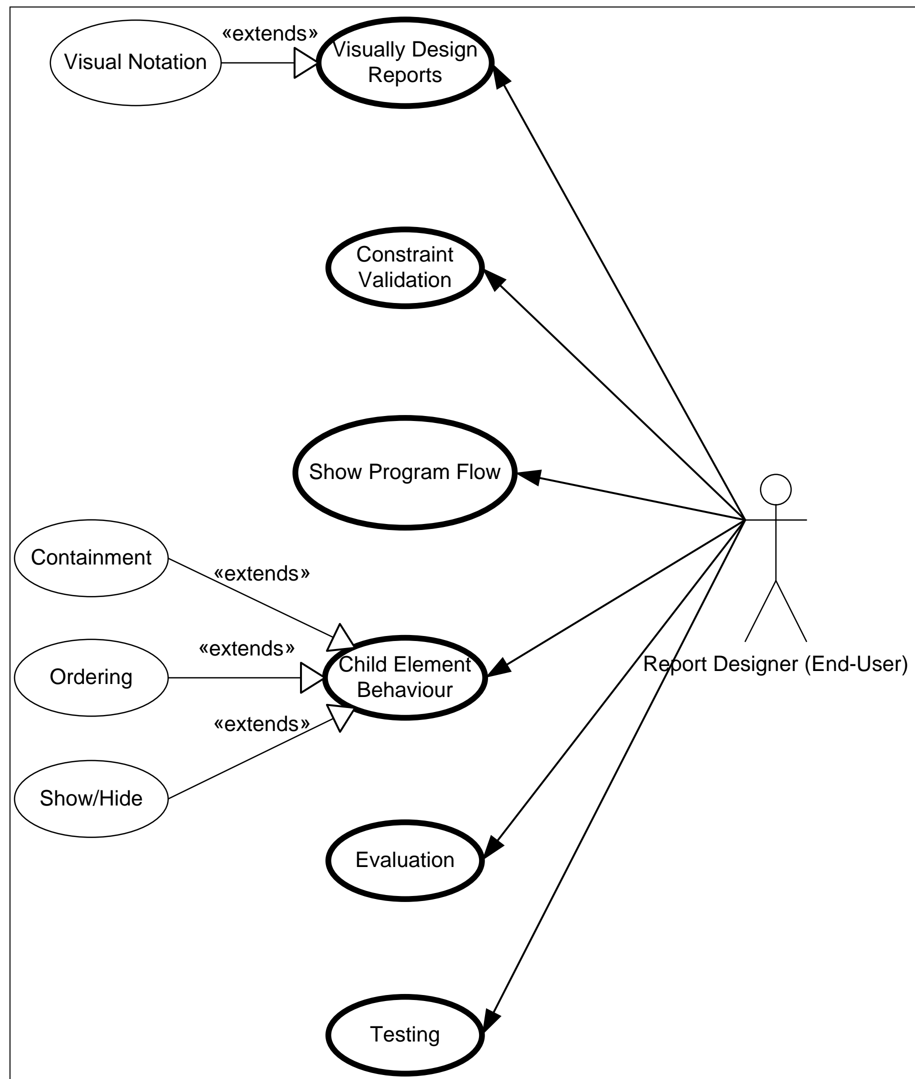


Figure 70: Use case end-user perspective

Figure 70 identifies six core functionalities (identified by bold ellipses) in terms of the end-user. Other uses cases shown in the figure are specializations of the core use cases.

In the following sections we examine the design aspects of the approaches taken during the course of this thesis. The sections are analyzed according to the requirements highlighted in the use cases shown in Figure 69 and Figure 70.

5.4 Object oriented Design/Analysis

Our main hypothesis was to prove that an enterprise task such as designing a report is possible using a visual language. The visual language should potentially raise the abstraction level of the reporting language and other systems which are used in conjunction with it such as an enterprise database. It is also intended to make it easier to understand and maintain existing reports as well as create new reports.

One potential way of raising the level of abstraction is with the aid of a model as outlined in Section 2.3. Therefore, it was realized that a possible solution to our hypothesis was to use MDSD and let users create high level models of reports which abstract irrelevant details of the language and its

conjunctive systems. To give the user capabilities to design models, we needed to base these models on some semantics and constraints, a valid meta-model (Section 2.3.4).

This section details the steps on how the meta-model evolved from simple Prism reporting scripts from (Prism New Zealand, 2005) and also details some of the core constructs. It also illustrates the process via which the meta-model was developed by successively considering more complex and new RWL constructs and appending them to the meta-model. Along with adding new constructs we also continuously refactored our existing meta-model as we were using an agile software development process. This meant that we used an iterative approach to our solution design.

Each step is divided up into three main parts: Script, Meta-Model and Analysis. The script represents RWL code examples which we look at to develop the meta-model. The meta-model, just as the name suggests, is the meta-model which caters to the script introduced in the "Script" part and finally we analyze our meta-model. In our analysis we look at how new structures have been incorporated into the meta-model and their significance. We also look at any new relationships formed between new or existing meta-model RWL constructs.

Step 1:

Script:

```
1.  Code      RW_EXAMPLE_00
2.  Type      Standard
3.  Access    STSR + ARCR
```

Meta-Model:

ControlLineModel
-Code : string -Type : ListTypeEnum -Access : string

Analysis:
This is an example of a simplest valid Prism report script. It contains three fields:

1. Code: Uniquely identifies the report in the Prism WIN MIS system.
2. Type: The type of the report. Can be any of these six values {List, Summary, Period, Standard, General, ActionScript}.
3. Access: The windows[2] within the Prism WIN MIS system where this report can be accessed from.

Step 2:

Script:

```
1.  Code      RW_EXAMPLE_01
2.  Type      Standard
3.  Access    STSR + ARCR
4.  Name      "Report Writer Example 01"
```

Meta-Model:

ControlLineModel
-Code : string -Type : ListTypeEnum -Access : string -Name : string

Analysis:

Similar report to that shown in Step 1, although an extra attribute Name, which can be a small description of the report.

Step 3:

Script:

```
1.  Code      RW_EXAMPLE_02
2.  Type      Standard
3.  Access    STSR
4.  Name      "Report Writer Example 02"
5.
6.  Print     StandardPageHeader;
7.  Print     "Hello World";
8.  Print     StandardReportFooter;
```

Meta-Model:

Analysis:

The above meta-model is more complicated than the reporting script it models, although the reason for this is that we have identified some patterns and denoted them in the meta-model shown above.

- *ControlLineModel*
  Contains other fields (attributes). What each of these attributes mean is irrelevant as the important attributes were identified in Step 1.
  - *WindowSelectionModel*
    We identified that the field *Access* is only populated by the user with valid Prism WIN window[2] codes. Therefore, we anticipated a pattern and provided the user with the WindowSelectionModel via which this can be made possible.
- *IReportStatementModel*
  A generic interface which all models inherit from to give them the capability of being used in a more generic way if need be.
- *IGeneratedCodeModel*
  Any model which potentially generates RWL script has to inherit from this model. We added this model as we identified the need that the meta-model will be used to generate RWL scripts.
- *IPrintableModel*
  Any model that can be associated to a potential *Print* statement should inherit from this model. As seen in the script and in the meta-model, both StandardPageHeaderModel and StandardReportFooterModel can be used in a Print statement thus inheriting from this model. Also note the LiteralModel which inherits from this model as it can be used in a Print statement.
- *IControlStructureModel*
  Introduced this model to provide grouping to child models such as the PrintConstructModel, StandardPageHeaderModel and StandardReportFooterModel.
- *SequenceUtility*
  Any model can use this utility to provide a sequential ordering. This came about due to sequential nature of the Prism RWL.
- *Relationships (Contains and BelongsTo)*
  - *PrintConstructModel-IPrintableModel*
    The meta-model depicts the relationship between a *Print* statement (PrintConstructModel) and any printable item (IPrintableModel) via this relationship. Therefore, it shows it as a PrintConstructModel can contain 0..* IPrintableModel items and an IPrintableModel item has to belongs to a PrintConstructModel.
  - *ControlLineModel-CommentConstructModel*
    Depicting the fact that the ControlLineModel can contain 0..* CommentConstructModel items and that a CommentConstructModel item has to belong to a ControlLineModel.

Step 4:

Script:

```
1.  Code        RW_EXAMPLE_03
2.  Type        Standard
```

```
3.   Access    STSR
4.   Name      "Report Writer Example 03"
5.
6.   Print     StandardPageHeader;
7.   Scan RM
8.             Print RM_CUST + RM_NAME;
9.   End
10.  Print     StandardReportFooter;
```

Meta-Model:

Database

ControlLineModel
-Code : string
-Name : string
-Type : ListTypeEnum
-Link : string
-Access : string
-File : string
-PeriodEndReport : bool
-Action : string
-Sort : string
-Select : string
-Preselection : PreselectionBlockModel
-NoMargins : bool
-FixedFont : bool

Access:string
WindowSelectionModel

«interface»
IDBModel

«interface»
IReportStatementModel

-BelongsTo
1

«uses»

«interface»
IGeneratedCodeModel

TableModel
-TableName : string

1
-BelongsTo

«uses»
«uses»

1
-Has

-Contains 0..*

ColumnModel
-ColumnName : string

*
-Contains

LiteralModel
-Value : string

CommentConstructModel
-Value : string

«uses»

«utility»
SequenceUtility
-SequenceNumber : int

IControlStructureModel

0..*
«interface»
IPrintableModel

-Contains

StandardReportFooterModel

-Contains 0..*

-BelongsTo
PrintConstructModel
1

StandardPageHeaderModel

ScanConstructModel
1
-BelongsTo

-BelongsTo 0..1

Analysis:

Only the new constructs added in the above meta-model are analyzed from this point forward.

- *ScanConstructModel*
  A ScanConstructModel represents an SQL select statement as highlighted in Section 1.4. Therfore, it has a TableModel associated with it.
- *TableModel*
  A TableModel represents a given Prism database table (view) which inherits from a IDBModel as it is a database concept.
- *ColumnModel*
  Encapsulates a column with a Prism database table (view). It inherits from the IPrintableModel interface as it can be used in conjunction with a *Print* statement.
- *Relatsionships (Contains and BelongsTo)*
  - *ScanConstructModel-ScanConstructModel*
    We have anticipated the necessity of this relationship as a *Scan* can potentially have another nested scan within it. This relationship is recursive in nature, thus represented by a reference to itself.

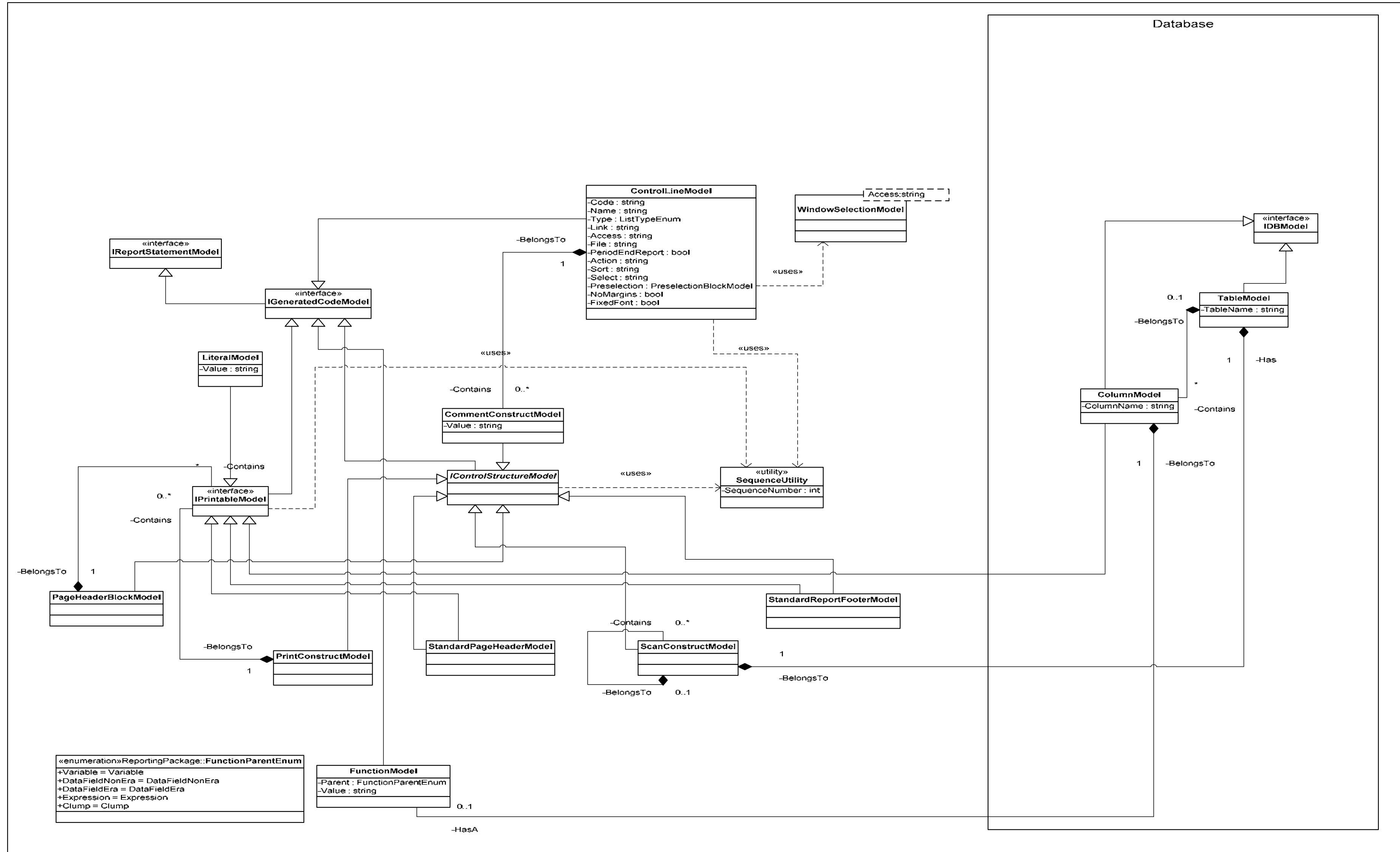
Also note the swimlane *Database* on the left hand side of the meta-model. This is introduced so we can keep the models which are database oriented disparate from the RWL models.

Step 5:

Script:

```
1.  Code      RW_EXAMPLE_04
2.  Type      Standard
3.  Access    STSR
4.  Name      “Report Writer Example 04”
5.
6.  PageHeader
7.       Print     StandardPageHeader;
8.       Print     RM_CUST.ColDesc + RM_NAME.ColDesc;
9.       Print;
10. End
11.
12. Scan RM
13.           Print RM_CUST + RM_NAME;
14. End
15.
16. Print     StandardReportFooter;
```

MetaModel:



Analysis:

New constructs added:

- *FunctionModel*
  Represents a function, for e.g. *ColDesc* seen in the RWL script. Note the relationship between the *FunctionModel* and the *ColumnModel*.
  - *FunctionParentEnum*
    An enumerated object contain five possible values representing the various constructs on which a function can be performed. This was a pattern established by looking further into the RWL semantics using (Prism New Zealand, 2005).
- *PageHeaderBlockModel*
  A grouping statement which allows us to have multiple constructs within the page header of a report. Also note the relationship between the *PageHeaderBlockModel* and *IPrintableModel* interface. This is due to the fact that a *PageHeaderBlockModel* can contain 0..* number of *IPrinableModel* items.

Step 6:

Script:

```
1.  Code      RW_EXAMPLE_05
2.  Type      Standard
3.  Access    STSR
4.  Name      “Report Writer Example 04”
5.  Select    RM_CUST.From + RM_CUST.To + RM_BAL_TYPE.Match
6.
7.  Clump cust_clump = RM_CUST + RM_NAME + RM_BAL_TYPE + RM_BAL_OWE;
8.
9.  PageHeader
10.     Print     StandardPageHeader;
11.     Print     cust_clump.ColDesc;
12.     Print;
13. End
14.
15. Scan RM
16.           Print cust_clump;
17. End
18.
19. Print     StandardReportFooter;
```

MetaModel:

ReportWriter

Database

ControlLineModel
-Code : string
-Name : string
-Type : ListTypeEnum
-Link : string
-Access : string
-File : string
-PeriodEndReport : bool
-Action : string
-Sort : string
-Select : PreselectionBlockModel
-NoMargins : bool
-FixedFont : bool

Access:string
WindowSelectionModel

PreselectionBlockModel

«interface»
IReportStatementModel

«interface»
IGeneratedCodeModel

LiteralModel
-Value : string

CommentConstructModel
-Value : string

ClumpModel
-Name : string

«interface»
IPrintableModel

IControlStructureModel

«utility»
SequenceUtility
-SequenceNumber : int

«interface»
IDBModel

TableModel
-TableName : string

ColumnModel
-ColumnName : string

PageHeaderBlockModel

StandardReportFooterModel

PrintConstructModel

StandardPageHeaderModel

ScanConstructModel

«enumeration»ReportingPackage::**FunctionParentEnum**
+Variable = Variable
+DataFieldNonEra = DataFieldNonEra
+DataFieldEra = DataFieldEra
+Expression = Expression
+Clump = Clump

FunctionModel
-Parent : FunctionParentEnum
-Value : string

-BelongsTo 1
«uses»
-Contains 0..1
1 -BelongsTo
«uses»
«uses»
-BelongsTo
0..* -Contains
0..1
-Contains 0..*
0..1 -BelongsTo
* -Contains
0..* *
-Contains -Contains
«uses»
0..1 -BelongsTo
-BelongsTo 1
-BelongsTo 1
-Contains 0..*
1
-BelongsTo
-BelongsTo 0..1
-BelongsTo 1..*
0..1 -BelongsTo
* -Contains
1 -Has
0..1 -HasA
0..1 -HasA

<u>Analysis:</u>
New constructs added:

- *ClumpModel*
  A *Clump* represents a variable which enables grouping of various values. For e.g. in this particular step it groups a set of database columns.
- *PreselectionBlockModel*
  Represents the *Select* field withing the *ControlLineModel*. It potentially allows the report designer to input runtime values which can be used by the report.
- *Relationships*
  - *ClumpModel-ColumnModel*
    A *Clump* can potentially contain 0..* *Columns* and a *Column* can belong to *Clump*.
  - *ClumpModel-FunctionModel*
    A *Clump* can have a *Function* and a *Function* can belong to a *Clump*. Also note that this changes the multiplicity of the relationship *ColumnModel-FunctionModel*. Now a given *FunctionModel* can belong to either a *ColumnModel* or a *ClumpModel*.

Note the colour coding of the swimlanes to enable us to distinguish between constructs which belong to the database side as opposed to the constructs which belong to the report writer.

<u>Step 7:</u>

After these steps it became evident that as new RWL constructs got added the meta-model did not have to sustain major changes. Therefore we could successfully say that we had designed a partial meta-model of the RWL which could now be converted from a static UML structure to a DSL.

<u>Step N:</u>

Shows the final (although partial) UML meta-model after which we started the implementation phase of the thesis. We can continue to refine and add more RWL constructs to this meta-model although we do not expect many changes to the meta-model as the core constructs and relationships have already been captured.

Note the new RWL constructs shown in the meta-model, for e.g. *IfPredicate*, *ChooseModel*, etc. We do not analyze these newly added constructs as their details are irrelevant in terms of this thesis because all we are trying to do is represent these in a DSL. Moreover, as the RWL is always evolving this thesis focuses on providing a proof of concept prototype rather than catering for “all” possible RWL constructs.

Meta-Model:

ReportWriter

Database

«utility» SequenceUtility
-SequenceNumber : int

«uses»

«interface» IReportStatementModel

Select:string

SelectModel

ControlLineModel
-Code : string
-Name : string
-Type : ListTypeEnum
-Link : string
-Access : string
-File : string
-PeriodEndReport : boo
-Action : string
-Sort : string
-Select : PreselectionBlockModel
-NoMargins : bool
-FixedFont : bool

Access:string, Link:string

WindowSelectionModel

PreselectionBlockModel

«interface» IGeneratedCodeModel

ClumpModel
-Name : string
-Down : bool

VariableModel
-Name : string
-Type : object

CommentConstructModel
-Value : string

NewPageModel

IControlStructureModel

IHeaderFooterModel

PageHeaderBlockModel

IGeneralControlStructureModel

PrintConstructModel

«interface» IPrintableModel

StandardPageHeaderModel

StandardReportFooterModel

ForConstructModel

«enumeration» ReportingPackage::IfConcatenator
+And = AND
+Or = OR

IfPredicate
-IsNegated : boo
-Concatenate : IfConcatenator
-EquateTo : object

ConditionConstructModel

IfElseConstructModel
-Type : IfElseEnum

SetIfConstructModel
-Type : IfElseEnum = IF

«enumeration»ReportingPackage::IfElseEnum
+If = IF
+Else = ELSE
+ElseIf = ELSEIF

LiteralModel
-Value : string

ScanConstructModel

PrintableFunctionModel
-Colour : int
-Format : int
-Font : string

ChooseModel
-ChooseType : ChooseEnum

«enumeration» ReportingPackage::ChooseEnum
+Match = Match
+From = From
+To = To
+Not = Not

«enumeration» ReportingPackage::FunctionParentEnum
+Variable = Variable
+DataFieldNonEra = DataFieldNonEra
+DataFieldEra = DataFieldEra
+Expression = Expression

FunctionModel
-Parent : FunctionParentEnum
-Value : string

«interface» IDBModel

TableModel
-TableName : string

ColumnModel
-ColumnName : string

-BelongsTo -Contains -Has -HasA «uses» 1 0..1 0..* * 2

5.5 Overall architecture

5.5.1 Class Diagram Approach

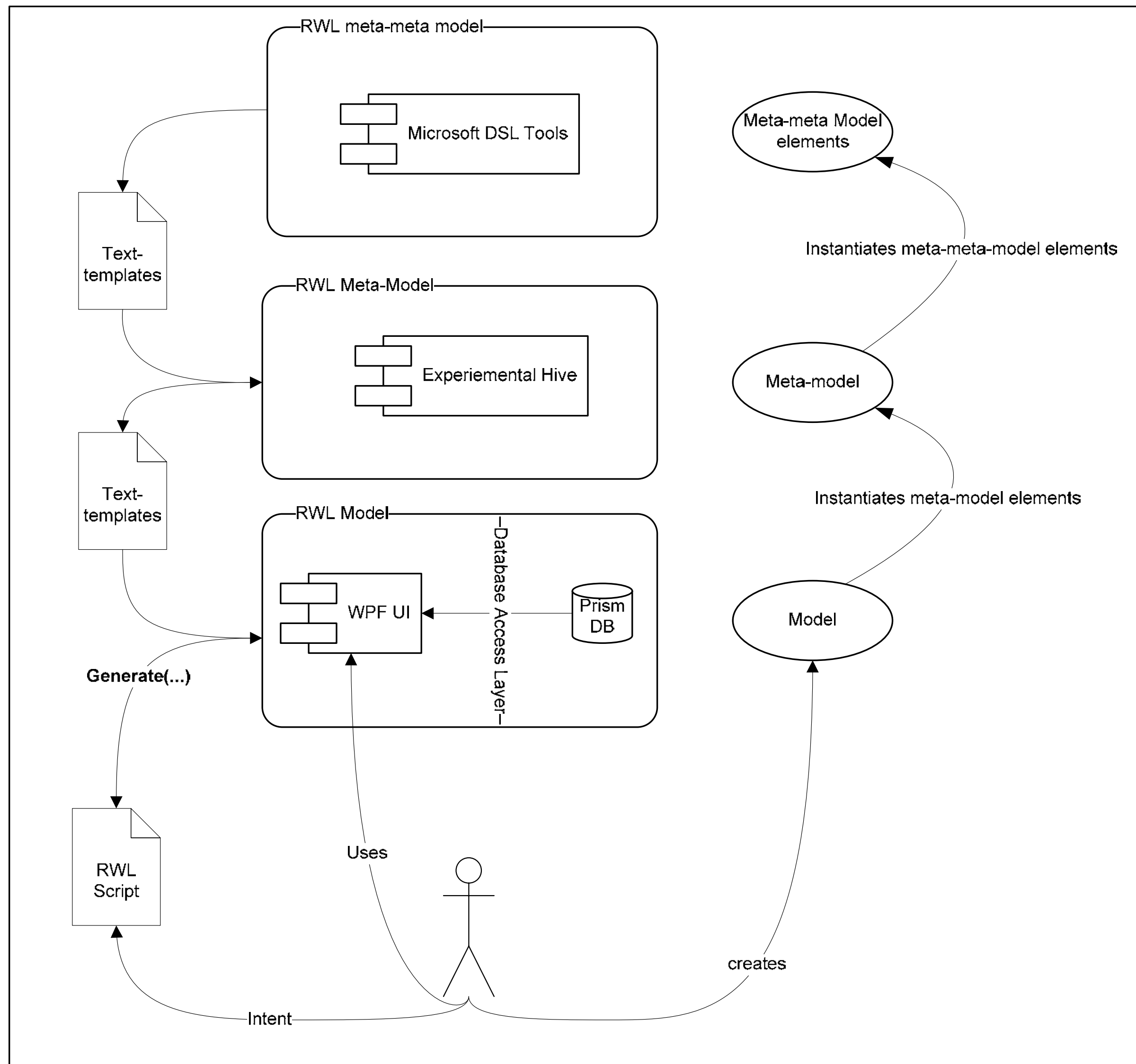


Figure 71: Class Diagram Approach: Architecture

Figure 71 above shows a high level architecture diagram of the Class Diagram approach. We see in the centre of the figure the essence of this approach where we create the meta-meta model from which we create the meta-model and eventually the model. If we look from a different perspective we can see that the end-user creates a RWL model which instantiates RWL meta-model elements which in turn instantiates the meta-meta model elements. The architecture diagram also shows at what level we access the Prism database and the user interaction with the UI, the model and the expected RWL script. The figure also depicts how the code is generated from the models with the final RWL script being generated by the UI by calling individual Generate(...) methods on model elements.

5.5.2 RWM Shell Approach

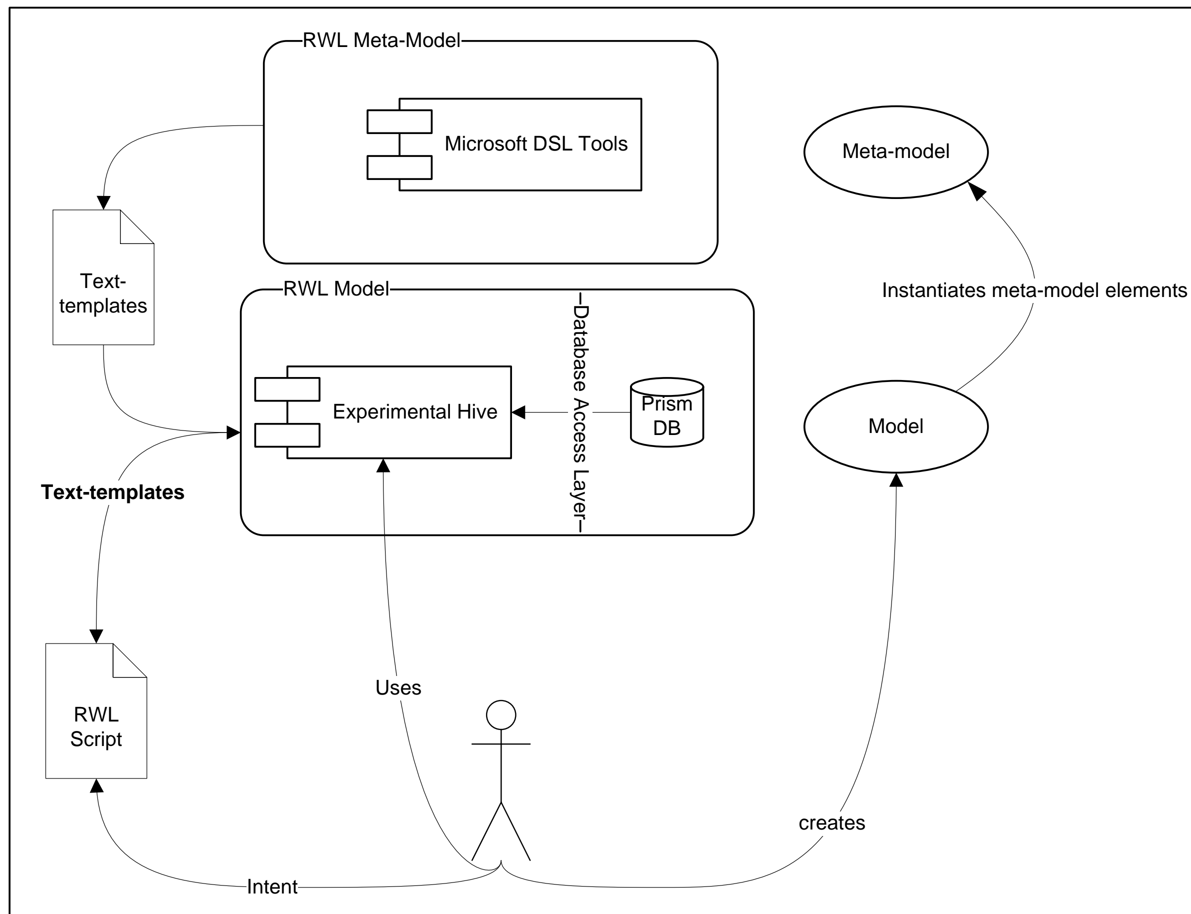


Figure 72: RWM Shell Approach: Architecture

Figure 72 above shows a high level architecture diagram of the RWM Shell approach. Although similar to our Class Diagram approach, we potentially have one less layer in this approach. Our WPF UI layer is subtracted and our RWL meta-model is represented in the DSL Tools itself. Also note that we use text templates to generate our RWL script as opposed to a individual Generate(...) method like in the Class Diagram approach.

5.6 Approach Overview

This section gives us a very brief overview of the two approaches tried to achieve the goal to modelise the RWL and provide users with a visual mechanism to design Prism reporting scripts.

Our first approach, Class Diagram Approach, involved the use of UML-like class diagrams. We decided on this approach as class diagrams provide a generic yet efficient way of modeling systems. It was also decided that we would append domain specific information into this class diagramming tool which would allow us to create a potential meta-model of the RWL where each construct would be represented as a class and relationships represented as associations. Therefore we essentially used the Microsoft DSL Tools to design this highly customizable Class Diagram Tool which was then utilized to create a RWL meta-model. At the end of the process we designed a standalone UI using WPF which exposed this RWL meta-model to end-users via which they could design RWL models and generate RWL script from them. It was also realized in the early stages of this approach that representing RWL constructs and relationships using shapes and lines would involve a heavy

overhead in terms of programming and time. Therefore we decided to discard this approach and use a simpler alternative allowing us to quickly develop a prototype for a visual RWL tool. This led us to our second approach, RWM Shell Approach.

Our second approach was similar to our first approach although we decided to remove the WPF UI layer. Therefore what we decided to do was to represent the RWL meta-model using the Microsoft DSL Tools as opposed to representing it using a highly customized class diagramming tool. This allowed us to rapidly design the meta-model and expose it using the Visual Studio Shell via which the end-users could design RWL models. We could then write templates against these models to simply generate the RWL script. This approach gave us the ability to quickly develop the RWL meta-model and the prototype needed to allow end-users design Prism reports. One drawback of this approach was the lack of customization which a WPF UI would offer. Therefore a trade-off was made and we chose the path which allowed us to create a functioning prototype within the time period allowed for our thesis.

5.7 Class Diagram-Based Design

5.7.1 Overview

The Class Diagram Approach was introduced in Section 3.4.2. We look at this approach and the design aspects in detail in this sub-section.

The Class Diagram approach was primarily taken to provide end-users with a rich UI to design Prism reports. This was possible using WPF (introduced in Section 3.3.2). The paradigm behind the approach was to design a rich UML-like DSL which could be flexible enough to let developers design a functionally rich meta-model of the RWL. This meta-model could then be exposed by a sophisticated front-end to users assisting them to design Prism reports. This front-end UI would also generate the required RWL script from the model which the end-users created.

A major flaw in this approach was that it would essentially take a lot of time and resources to design a partially functional front-end which could effectively demonstrate the meta-model. The front-end had to visualize shapes, relationships and other constraints which were not feasible in the time period allowed for this thesis. Nonetheless this was realized part way through the design and implementation stage therefore we include this approach in this thesis.

5.7.2 Meta-Model Design

The meta-model of the RWL in this approach was divided up into two steps. The first step involved designing a meta-meta-model using the DSL Tools and the second stop involved using this meta-meta-model to create the RWL meta-model.

Our meta-meta-model as described in previous sub-section is essentially a customized UML meta-model. The following sub-sections describe the design decisions taken in terms of the use-cases illustrated in Figure 69.

5.7.2.1 Add/Modify meta-model elements

The DSL Tools provide a Class Diagram model wizard which we used to create an initial model of a tool which allows us to draw highly customized domain specific class diagrams. The figure below shows what we initially started with while using this approach.

Figure 73: Class Diagram approach: Initial class diagram (meta-meta-model)

Figure 73 essentially shows a meta-model of a UML-like language which we can customize and configure according to our domain. After refining this model to fit our domain it will became our meta-meta-model. Via this meta-meta-model we will be able to design a highly domain specific visual representation of the RWL constructs, i.e. the RWL meta-model. For e.g. if we need a RWL construct called ControlLine model, we would be able to instantiate a meta-meta-model element called *ModelClass* (shown in Figure 73) and call is *ControlLine*. This *ControlLine* RWL meta-model construct represented by a *ModelClass* meta-meta-model construct will eventually be instantiated by the end-user via the standalone UI.

Figure 74 below shows the built-in templates within the DSL Tools which generate the executable code for the model (executable code for the meta-meta-model) shown in Figure 73.

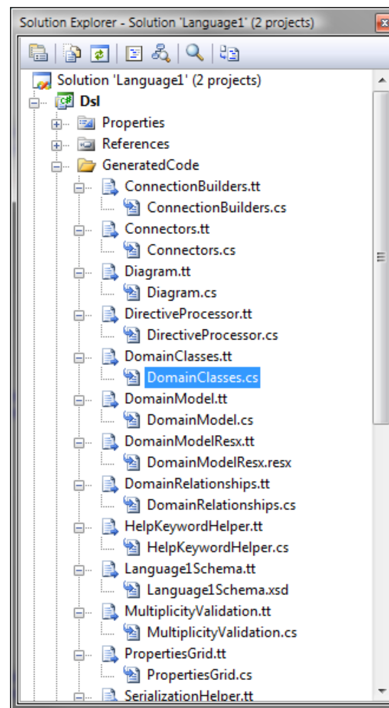


Figure 74: Built-in templates

5.7.2.2 Add/Modify meta-model constraints

Microsoft DSL Tools expose various mechanisms to design model constraints. The DSL Tools provided us with *relationships, rules* and *overrides* to cater for hard and soft/dynamic constraints. We look at what each of these concepts mean and then we analyze them with respect to our approach to detail how we used these concepts to meet our requirements.

Microsoft DSL Constraint Overview

Relationships

The DSL Tools exposes two types of relationships which we can use to create associations between model elements and therefore enforce hard constraints. We can also force this constraints to allow us to propagate deletes and updates if need be.

1. Embedding Relationship (Part-of or aggregation relationship)
   An embedding relationship between two model elements means that the target model element is embedded within the source model element. Every model element has to be embedded within one source model, this is necessary for the successful serialization of the model. A target model element can only be in one embedding relationship, this rule ensures that a model element has only one parent at all times. An example of an embedding relationship *FamilyHasPeople* is shown below in Figure 75.

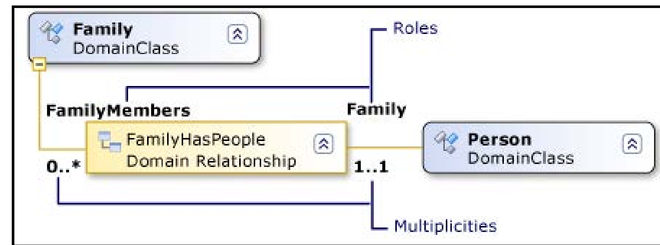


Figure 75: Embedding relationship reproduced from (Microsoft, 2007)

Note in Figure 75 the opposite roles, to read this relationship we state that a *Person* has to belong to one and only one *Family* and a *Family* can have more none or infinite *Person*. The role accessors appear so that to access all the *Person* objects in a *Family* we have to go through the property *FamilyMembers* and to access the *Family* which a *Person* belongs to, we have to go through the property *Family*.

Also note that the target model of an embedding relationship appears as a child node of the source model element in the Explorer View (essentially a tree-view of all the model elements).

2. Reference Relationship (Has-a/Uses or association relationship)
   A reference relationship, as the name states, is when a source model element references a target model element. The target model element appears as a property within its source model element. Reference relationships have no strict rules on the multiplicities its source and target model elements and are read the same way as an embedding relationship. The only difference is that they are represented using a dashed line as shown below in Figure 76.

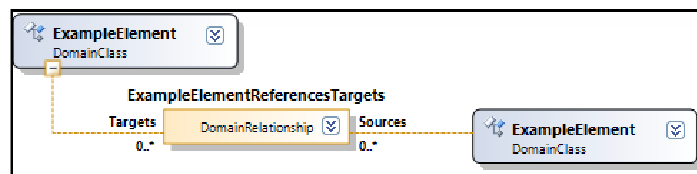


Figure 76: Reference relationship

Rules

A rule is created when the DSL designer needs to propagate changes to other parts of the model when a particular model element is affected (Cook, Jones, Kent, & Wills, 2007). Rules are similar to events which get triggered on various changes. For e.g. a rule can be triggered when a model element is added (inherit from AddRule) or changed (inherit from ChangeRule). *AddRule* and *ChangeRule* are built-in classes provided by the DSL Tools framework.

The framework mandates that an attribute is used to specify which model element a particular rule applies to. This is done by using the *RuleOn* attribute: `[RuleOn(typeof(Person), FireTime = TimeToFire.TopLevelCommit)]`. This adds a rule on the model element called *Person*. Other details about the *RuleOn* attribute are irrelevant.

For e.g. let us have a look at the rule shown below:

```
[RuleOn(typeof(Person), FireTime = TimeToFire.TopLevelCommit)]
internal sealed class PersonDeleteRule : DeleteRule
{
        public override void ElementDeleted(ElementDeletedEventArgs e)
```

```
        {
                if (MessageBoxResult.Yes ==
                        MessageBox.Show("All children of this person will be deleted, continue?")
                {
                        base.ElementDeleted(e);
                }
        }
}
```

The above rule applies whenever a *Person* is deleted from the model. It simply allows us to ask the end-user whether they do want to delete this person and all children it may have. *TimeToFire.TopLevelCommit* simply states that this rule will be triggered when the top level transaction commits its changes. We can also fire rules when either a nested transaction commits (*TimeToFire.LocalCommit*) or directly after a change is made (*TimeToFire.Inline*).

Overrides

The code generated by the templates shipped with the DSL Tools to represent the model is in C#. The code is structured in a modular and flexible way which allows the user to easily override its default behaviour. This is very useful when custom logic is needed to do validation and other constraint checks for a target DSL tool.

For the Class Diagram approach we are essentially writing UML like constraints which force the DSL user to create a valid class diagram of our RWL meta-model. An example of such a constraint is demonstrated by a reference relationship shown below in Figure 77.

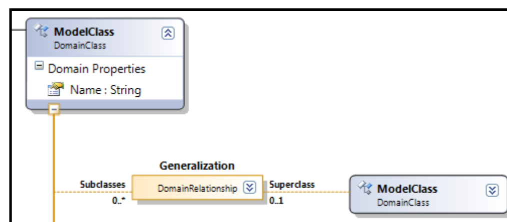


Figure 77: Class Diagram Approach: Class Diagram relationship constraint

Note in the above figure, model element *ModelClass*. A constraint seen in Figure 77 is imposed by the reference relationship *Generalization*. This shows that a *ModelClass* may only have one *ModelClass* as its *SuperClass*.

Another possible constraint is on the *ModelClass* model element which states that its *Name* property cannot be empty. This constraint can be enforced by an override shown in Figure 78.

```
internal sealed partial class NamePropertyHandler
{
    protected override void OnValueChanging(ModelClass element, string oldValue, string newValue)
    {
        if (String.IsNullOrEmpty(newValue))
            throw new ArgumentException("Name has to be populated");
        base.OnValueChanging(element, oldValue, newValue);
    }
}
```

Figure 78: Class Diagram Approach: Class Diagram override

5.7.2.3 Design code generator

The code generator for the Class Diagram approach is essentially a three phase process as described in Section 3.4.2.1 and shown in Figure 79.

1. Automatically[4] generate code representing the meta-meta-model of our UML-like DSL. Create a UML like model representing the RWL meta-model.
2. Generate code representing the meta-model using text templates. End-user creates RWL models via the WPF UI which uses this code.
3. Generate RWL script representing the RWL model which the end-user created.

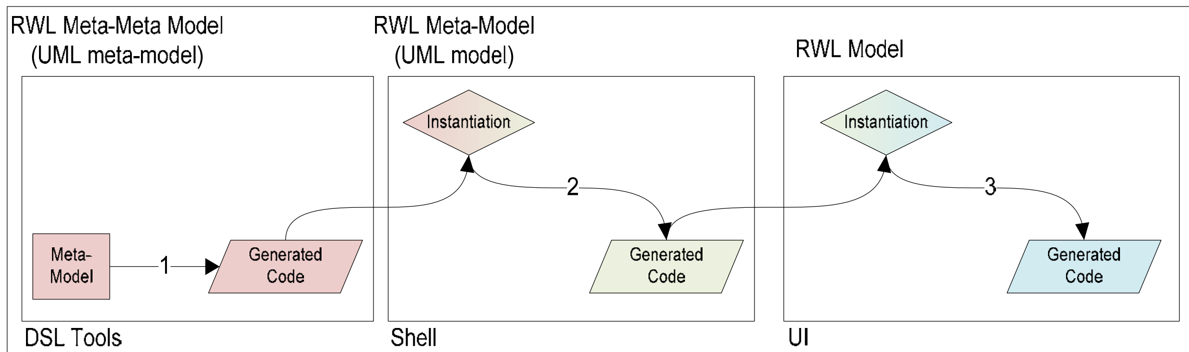


Figure 79: Class Diagram Approach: Code generator

The above process is further explained in Chapter 6 (Section 6.3.2.2) where we look at the implementation details of the code generator.

5.7.2.4 Extension Points (Mandatory Fields)

Mandatory fields are nothing else except attributes or properties which need a user value present to make it valid. Therefore, in our class diagram approach we envision each RWL construct being represented as a *Class* and each field being represented as an *Attribute* within that class. Figure 80 represents this *Class* contains *Attribute* relationship.

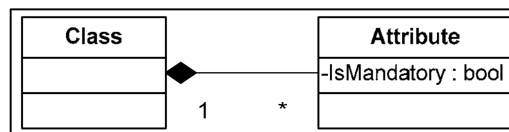


Figure 80: Class Diagram Approach: Mandatory fields design

Note from the above figure that the *Attribute* object contains an attribute called *IsMandatory* which allows developers to denote either that particular *Attribute* is a mandatory *(true)* or not *(false)*. This approach makes it flexible for future developers to add/modify mandatory attributes when designing the RWL meta-model using this meta-meta-model.

For example, the *ControlLine* construct within the RWL would be represented as a *Class* and one of its attributes, *Code*, will be represented by an *Attribute*. Moreover as previously stated, *Code* is a mandatory attribute therefore it will have the *IsMandatory* property set to true.

5.7.2.5 Extension Points (Field Editors)

The idea of field editors is to allow developers select a specialized window via which a field of a RWL construct can be edited. We cater for this requirement in a similar fashion to that described in Section 5.7.2.4. We add another attribute to our *Attribute* object, introduced in Figure 80, called *EditorType*. This is shown below in Figure 81.

[4] Microsoft DSL tools generate this code using built in text templates.

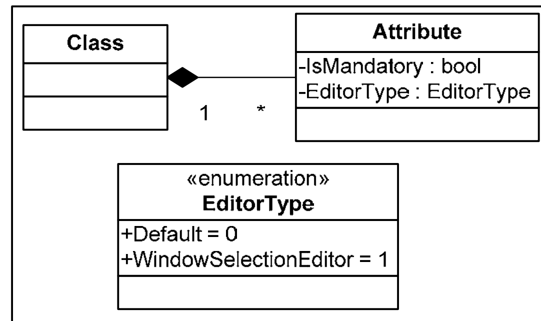


Figure 81: Class diagram approach: Field editors design

Note the enumeration type *EditorType*, this currently contains only two values, namely *Default* and *WindowSelectionEditor*. This enumeration value can be expanded by other values as our meta-model grows although currently we have only anticipated two values.

Let us look the significance of each enumerand value.

1. *Default*
   The default field editor provided by the system will be used.
2. *WindowSelectionEditor*
   An editor which displays all the Windows[2] that exist with the Prism WIN MIS system (illustrated in 4.2.1.4).

5.7.2.6 Extension Points (Prism meta-data access)

For our Class Diagram Approach the Prism meta-data does not need to be accessed unless when needed by the end-user while designing the RWL model. Meta-data access allows the end-user to lookup information about the Prism database while using the visual language as opposed to remember the information. Therefore the design of our meta-data access is important so we can give users as much information about the structure of the Prism database. This is done by using LINQ (introduced in Section 3.3.3).

LINQ provides developers to query databases using programming language syntax. LINQ also provides an ORM (object relational mapping) tool[5] for database tables and views. We use this mapping to give us readymade mapping classes for tables and views which contain Prism meta-data information.

[5] This tool itself is designed using Microsoft DSL tools.

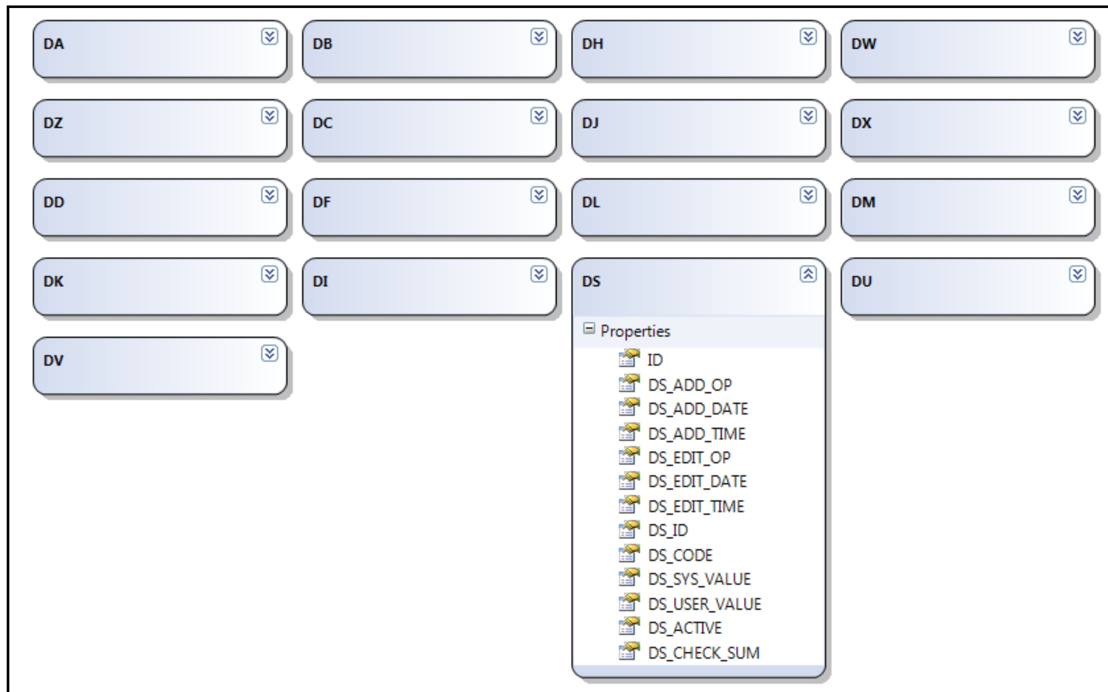


Figure 82: Prism meta-data

Figure 82 above shows all the views within the Prism database which contain meta-data information. We have shown an expanded version of one of these views to display its properties, note these properties match the columns within the view in the database. Also note the two part naming convention. This convention does follow certain business rules (for e.g. all system tables as the ones shown in the above figure are like D*) although it is not user friendly and requires the end-user to have intimate knowledge about the Prism database.

```
[Table(Name="dbo.DS")]
public partial class DS
{
    Fields

    public DS()...

    [Column(Storage="_ID", AutoSync=AutoSync.Always, DbType="Decimal(18,0) NOT NULL IDENTITY", IsDbGenerated=true)]
    public decimal ID...

    [Column(Storage="_DS_ADD_OP", DbType="Char(6) NOT NULL", CanBeNull=false)]
    public string DS_ADD_OP...

    [Column(Storage="_DS_ADD_DATE", DbType="SmallDateTime NOT NULL")]
    public System.DateTime DS_ADD_DATE...

    [Column(Storage="_DS_ADD_TIME", DbType="Int NOT NULL")]
    public int DS_ADD_TIME...

    [Column(Storage="_DS_EDIT_OP", DbType="Char(6) NOT NULL", CanBeNull=false)]
    public string DS_EDIT_OP...

    [Column(Storage="_DS_EDIT_DATE", DbType="SmallDateTime NOT NULL")]
    public System.DateTime DS_EDIT_DATE...

    [Column(Storage="_DS_EDIT_TIME", DbType="Int NOT NULL")]
    public int DS_EDIT_TIME...

    [Column(Storage="_DS_ID", DbType="Int NOT NULL")]
    public int DS_ID...

    [Column(Storage="_DS_CODE", DbType="Char(22) NOT NULL", CanBeNull=false)]
    public string DS_CODE...

    [Column(Storage="_DS_SYS_VALUE", DbType="Char(82) NOT NULL", CanBeNull=false)]
    public string DS_SYS_VALUE...

    [Column(Storage="_DS_USER_VALUE", DbType="Char(82) NOT NULL", CanBeNull=false)]
    public string DS_USER_VALUE...

    [Column(Storage="_DS_ACTIVE", DbType="SmallInt NOT NULL")]
    public short DS_ACTIVE...

    [Column(Storage="_DS_CHECK_SUM", DbType="Int NOT NULL")]
    public int DS_CHECK_SUM...
}
```

Figure 83: Generated mapping from LINQ tool

An example of a corresponding mapping file generated from one of the views show in Figure 82 is shown above in Figure 83. This mapping file allows the developer to write code like that shown in Figure 84 to query information from the view DS which is part of the Prism database which contains meta-data information (as described in Section 1.5.1, note the naming scheme, all tables and views in the Prism database which contain meta-data information are in the format D* where * can be another alphabet).

```
IEnumerable<DS> allTuples = from dsTuple in DataContext.DSs select dsTuple;
```

Figure 84: Example code to query a given Prism meta-data view or table

Therefore, it can be seen that using LINQ and a dedicated Data Access Layer (DAL) makes the process of extracting meta-data information about the Prism database via simpler and straight forward for the developers. This meta-data can then be displayed to the end-user via an interface similar to that shown in Figure 169 and Figure 179.

5.7.2.7 Versioning

We cater for version changes by allowing the developer to enter an attribute value corresponding to the root of the meta-model which represents the current version of that meta-model. This attribute is shown below in Figure 85.

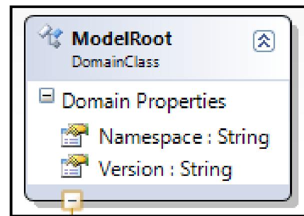


Figure 85: Class Diagram Approach: Versioning

A simple check could be done while opening an existing RWL model file, if the version numbers do not match the user could be informed or an automatic transformation process can be triggered. This process can allow the user to transform their RWL model corresponding to an older meta-model to a RWL model which corresponds to the current meta-model.

This attribute (Version) can be changed by the developer whenever a change is made to the RWL meta-model. A change may range from a simple attribute change to a complex relationship/entity change.

5.7.2.8 Explorer View

This approach has to cater for two distinct explorer views, one to allow quick access for developers to the meta-model they create using our meta-meta model tool and the second to allow end-users to see the RWL model they create using the meta-model.

It was decided that the best way to represent these was to provide a tree-based approach to the view of the model. DSL Tools already offered an explorer view (Figure 86) for the developers to see the meta-model they created using our meta-meta model tool so all we had to do was to provide a tree-view for the RWL model created by the end-user.

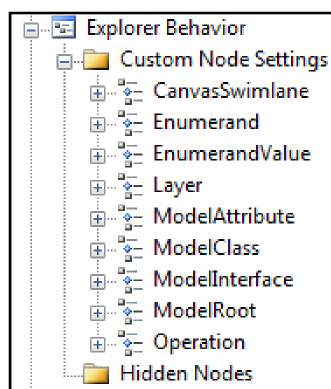


Figure 86: Class Diagram Approach: Meta-model explorer

5.7.2.9 Evaluation and Testing

The meta-meta-model tool caters for the developer therefore we have to design an easy way of evaluating it. It was decided that we evaluate and test the meta-meta-model tool by determining its effectiveness while creating the meta-model, determining the degree of automation and the correctness of the code generated which depicts the RWL meta-model.

Although this approach was discarded in its early stages, no formal evaluation or testing was carried out.

5.7.3 Shell Host Design

The Shell Host in our Class Diagram approach is essentially a WPF UI hosting the generated RWL meta-model classes.

As we moved on from this approach in the early design and implementation stages we did not cater for all the requirements via our user interface. The following sub-sections highlight the design decisions on the subset of requirements we catered for prior to moving on to our refined approach.

5.7.3.1 *Visually design reports (Visual Notation)*

The main purpose of our shell host is to allow users to utilize the RWL meta-model to visually create RWL models which will then generate the RWL script. We decided to statically allocate shapes to some common RWL constructs represented by the meta-model so that we can get our prototype displaying some shapes and generating code. Although our true intention was to allow developers to assign shape and notation values to RWL constructs while they design the meta-model.

Figure 87 below shows the relationship between our generated RWL meta-model code and shapes. Essentially each shape will have an instantiated RWL meta-model object behind it which allows the report designer to edit its properties and validate any constraints surrounding it.

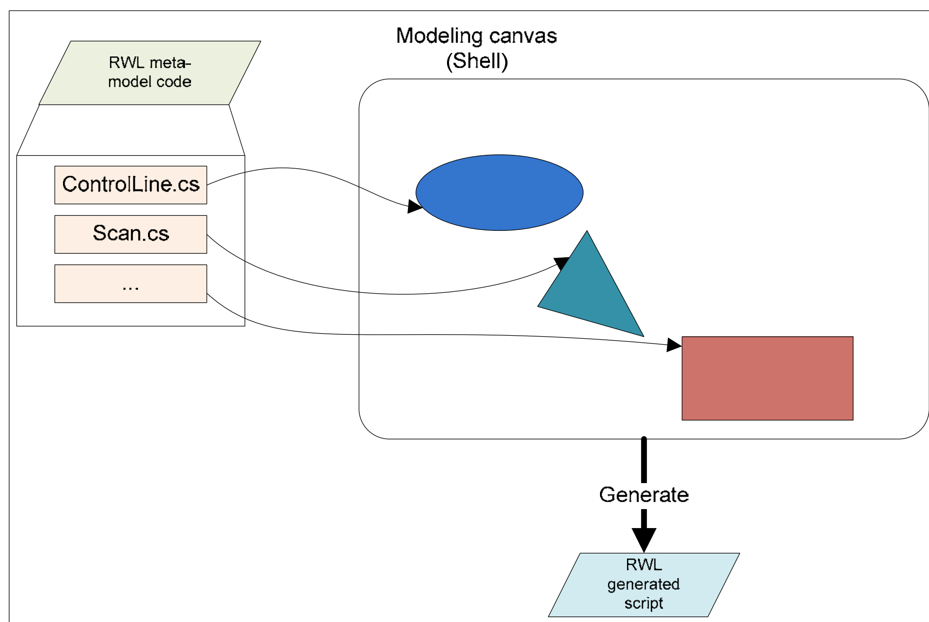


Figure 87: Class Diagram Approach: Visual designer design

5.7.3.2 *Constraint validation*

Due to the decommissioning of this approach (explained in detail in Section 5.7.1) in the early stages we only designed how we could implement validating our mandatory field constraints. It was decided that we allow the user to trigger a validation via a command, either through a menu or a sequence of keyboard keys. The validation was designed so that it triggers a series of events as shown in the sequence diagram in Figure 88.

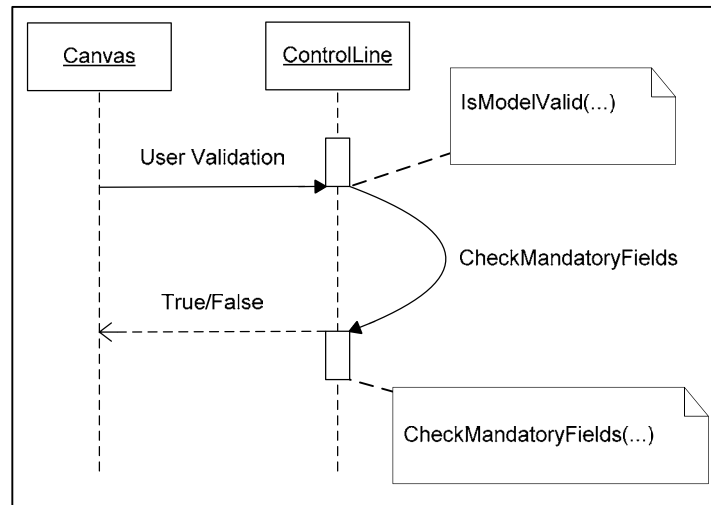


Figure 88: Class Diagram approach: Shell host validation method calls

The figure above shows an example validation request a user made on an instantiated meta-model element *ControlLine*. This request essentially calls a function on the model called *IsModelValid* which can call a series of constraint validators, in this case it only calls *CheckMandatoryFields* validator. The *CheckMandatoryFields* validator goes through all the fields marked with the *IsMandatory* property (outlined in Section 5.7.2.4) and returns either valid or invalid depending on whether a user has populated them.

5.7.3.3 Evaluation and Testing

Evaluation and testing of the WPF UI was done on the fly as we added more functionality to the interface. Formal evaluation and testing was scheduled to be done when the interface could be used to design an example report, although as we moved on to a different approach we had no further need to evaluate and test the interface. We explained why this approach was discarded in Section 5.7.1.

5.8 RWM Shell Approach Design

5.8.1 Overview

The Class Diagram approach was introduced in Section 3.4.2. We look at this approach and the design aspects in detail in this sub-section.

The RWL Shell approach was an evolution from our Class Diagram approach. It allowed us to create a rapid prototype based on the RWL meta-model we created using the Microsoft DSL Tools.

Note the difference between this approach and the Class Diagram Approach. As opposed to our Class Diagram approach where we offered a meta-meta-model to the developer which could then be used to create the RWL meta-model, in this approach we essentially design our RWL meta-model using the DSL Tools itself. Therefore the user interface which would expose this meta-model to the user would be very similar to the Experimental Hive (outlined in Section 2.5.1.4); this UI is called the Visual Studio Shell. This allowed us to concentrate on the RWL meta-model, its semantics and also allowed us to use the built-in shapes designer (built into the DSL Tools) which allowed us to create a demonstrable prototype.

Also note that the built-in shapes did not offer much variety, it is very easy to expand these to create sophisticated customized shapes. For the remainder of the thesis we have strived to create a consistent notation using built-in shapes and refrained from doing superfluous work on creating attractive shapes which could be easily done as a future enhancement.

Moreover the design a “good” visual language is an art and requires extensive usability evaluations and testing. We tried to keep the visual language as simple and closely mapped to the real world RWL constructs as possible although more design work is needed to make the visual language a marketable product.

5.8.2 Meta-Model Design

Creating the meta-model for this approach was essentially transforming our static OOD (detailed in Section 5.4) to the DSL Tools and representing the semantics using constraints and other tools provided by the DSL.

5.8.2.1 Add/Modify meta-model elements

In this approach we started off with an empty model as we need to add meta-model elements which match the RWL constructs. The initial model we started with is shown below in Figure 89.

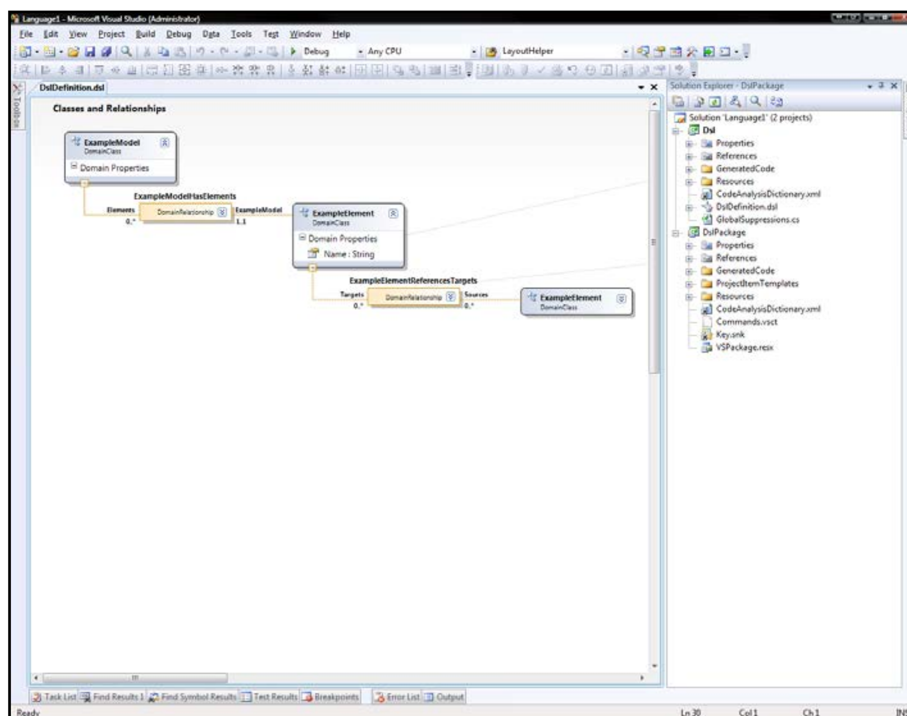


Figure 89: RWM Shell Approach: Initial meta-model

Note in the above diagram we have a few model elements pre-added for us by the DSL Tools, this is because the root model of any model cannot be empty therefore the DSL Tools anticipates this and adds the root for us. Therefore, when we start we have an executable model as opposed to an erroneous model. The code generator for the model shown in Figure 89 is the same as the one shown in Figure 74.

5.8.2.2 Add/Modify meta-model constraints

An overview on the different mechanisms provided by the Microsoft DSL Tools on how to add constraints was outlined in Section 5.7.2.2.

For this approach we are directly mapping our RWL semantic constraints within our model. Therefore, an example of a relationship constraint is shown below in Figure 90.

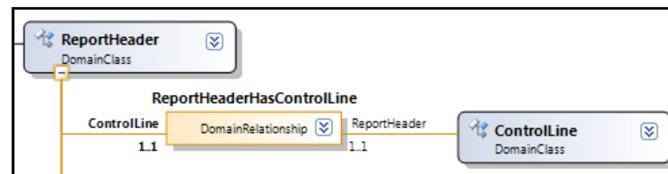


Figure 90: RWM Shell Approach: Relationship constraint

We can see from the above figure that a *ReportHeader* model element should have one and only one *ControlLine* model element. This is similar to the example shown in Figure 48.

Other constraints can be added in a similar fashion as shown in Figure 78. The core RWL constraints are attached as Appendix B.

5.8.2.3 Design code generator

This approach is essentially a two phase code generator as described in Section 3.4.2.2 and shown in Figure 91.

1. Automatically[4] generate code representing the RWL meta-model.
   Users create the RWL model which uses this code via the Shell.
2. Generate RWL script representing the RWL model using text templates.

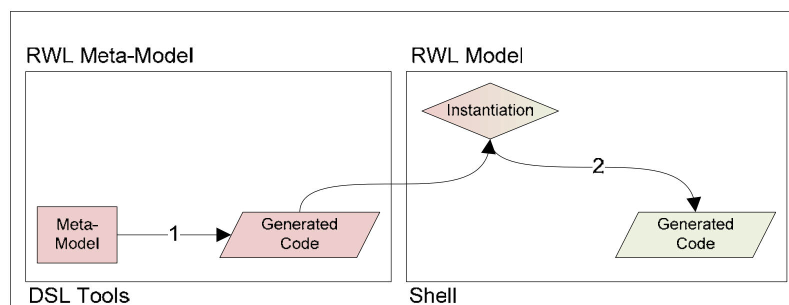


Figure 91: RWM Shell Approach: Two phase code generator

Note that in our RWM Shell Approach we essentially use the T4 Templating Engine to actually generate the RWL script as opposed to the code generation technique described for the Class Diagram Approach where we use the T4 Templating Engine to generate C# code representing the RWL meta-model. This allows us to reduce an entire step of generating code making the RWL Shell Approach ideal and easy to implement.

The above process is further explained in Chapter 6 (Section 6.4.2.1) where we look at the implementation details of the code generator.

5.8.2.4 Extension Points (Mandatory Fields)

In this approach we are essentially representing the RWL meta-model in the DSL Tools therefore we have to design and implement a mechanism via which we can represent mandatory fields. This mechanism has to be made flexible so that future developers can easily represent mandatory fields in new RWL meta-model constructs.

We decided to use .NET CLR attributes. Attributes give us an elegant way of marking code without changing the meaning of what the code represents. CLR attributes can be seen as meta-data for a particular code section. This meta-data can be queried at runtime and we as programmers can react to it accordingly. Moreover, DSL Tools have a flexible way for developers to mark fields with these attributes as shown in Figure 92.

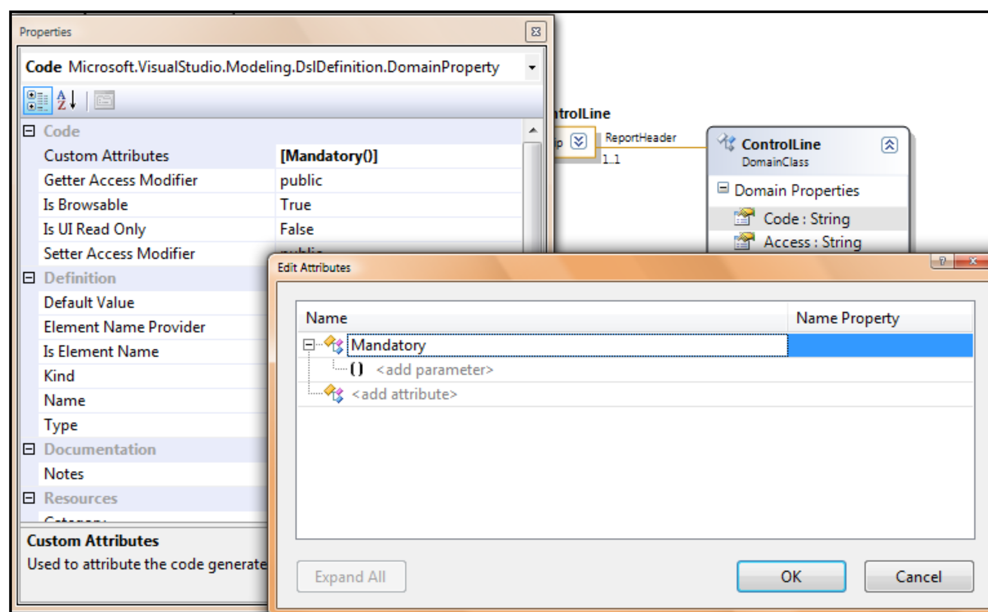


Figure 92: RWM Shell Approach: Mandatory attributes to mark mandatory fields

We added an attribute called *Mandatory* shown in the above figure. Therefore, any field annotated by the *Mandatory* attribute will have a constraint on it which requires the user to populate it to make that particular construct valid.

5.8.2.5 Extension Points (Field Editors)

Microsoft DSL Tools allow developers to specify customized field editors for any field. This is done by marking a field with the Microsoft .NET CLR *Editor* attribute. Details about this attribute are shown in Figure 93.

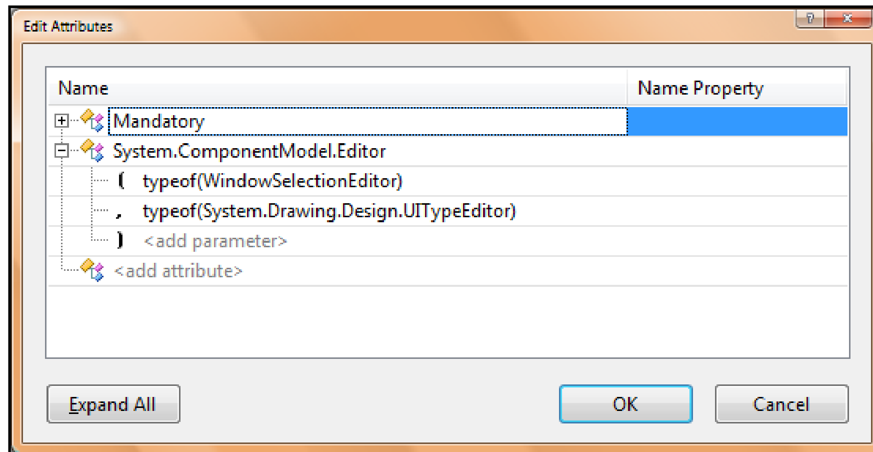


Figure 93: RWM Shell Approach: Specifying field editors

Note in the above figure the *Editor* attribute has two parameters, the first parameter specifies which type of editor handles this field (in this case a *WindowSelectionEditor*) and the second attribute specifies the base type (usually just `System.Drawing.Design.UITypeEditor`).

5.8.2.6 Extension Points (Prism meta-data access)

The meta-data access layer for both our approaches is identical. Refer to Section 5.7.2.6.

5.8.2.7 Versioning

This approach has no explicit versioning mechanism although as the meta-model grew we had to develop a manual process in which we modify existing RWL models to fit the newly developed meta-model. This manual approach is temporarily feasible as the DSL Tools allow us to serialize the RWL models into XML although as the RWL models grow in complexity this approach will soon become cumbersome and time consuming and an automated approach will have to designed.

Versioning is a vast research area and many solutions have been proposed especially in the databases and software tools area. A possible solution is described in (Roddick, 1995). We have also looked at a simple versioning and transformation process as a possible future enhancement which is explained further in Section 9.5.4.

5.8.2.8 Explorer View

We have to provide a tree-view to our RWL model for quick access by end-users. This capability can be easily catered for using the built-in explorer view provided by the DSL Tools. The figure below shows how the explorer view can be configured to correctly match the RWL model.

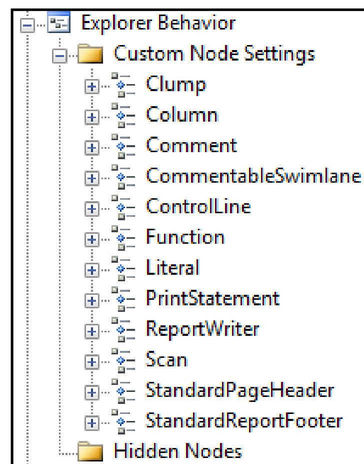


Figure 94: RWM Shell Approach: Model explorer

5.8.2.9 Evaluation and Testing

The evaluation and testing of the RWL meta-model can be done by designing RWL models against it. Various examples can be designed and evaluated against the meta-model to determine the effectiveness and correctness of it. Evaluation and testing has to be thorough enough to exercise all RWL semantics to determine whether the RWL meta-model correctly allows or restricts RWL construct creation and modification.

We have done a formal evaluation of this approach which is detailed in Chapter 8.

5.8.3 Shell Host Design

Our shell host is pre-designed for us as it is nothing except the Visual Studio Experimental Hive therefore it inherits most of the functionality from Visual Studio. Although we still have to cater for some of the extra functional requirements which the RWL model should exhibit. This functionality originates from the RWL meta-model but we include them in this section as it is functionality which is used by the end-user via the shell.

5.8.3.1 Visually design reports (Visual Notation)

The primary requirement of our shell host is to allow end-users to visually create RWL models. For this approach our visual notation is part of the meta-model as the DSL Tools allow us to represent each model construct with a corresponding shape. Therefore, it is possible that every time we add a RWL construct into the meta-model we can add a shape mapping which maps that construct to a shape. An example of a shape mapping is shown below in Figure 95.

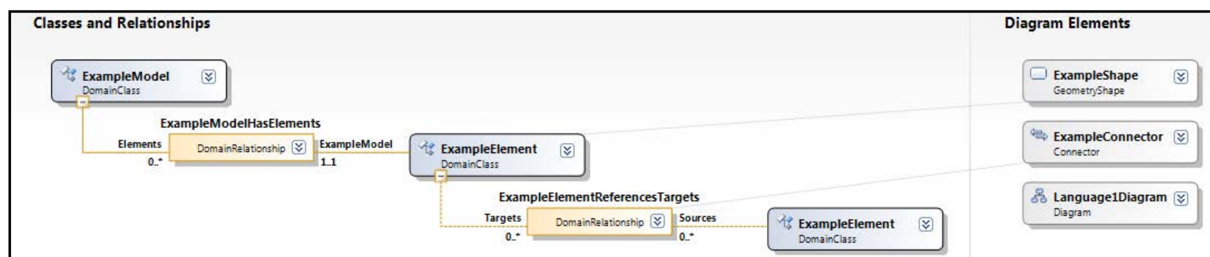


Figure 95: RWM Shell Approach: Visual notation via shape mapping

The above figure shows the same model as introduced in Figure 89; note the lines going from the model elements and relationships to shapes and connectors. These lines denote which shape and connector will represent which model and relationship on the shell canvas respectively.

5.8.3.2 Constraint validation

Validation is built-in into the shell and all we have to do is enable it for various triggers. Triggers in this respect are events such as save, load, menu etc. The enabling mechanism is shown in Figure 96.

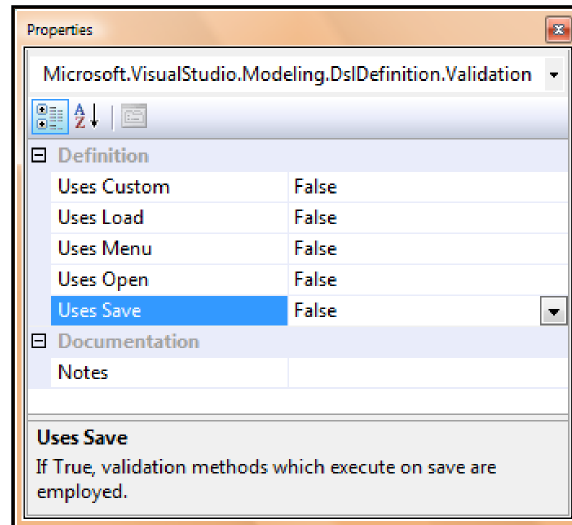


Figure 96: RWM Shell Approach: Enabling validation

The validation mechanism triggers rules on model elements which represent the RWL meta-model. Rules were introduced as part of the *Microsoft DSL Constraint Overview* in Section 5.7.2.2.

5.8.3.3 Show program flow

The RWL is inherently a sequential programming language where each RWL construct is followed by another construct and so on. Therefore it was mandatory to represent this behaviour visually to allow end-users to easily follow the flow of a given RWL model.

We decided to show this program flow with the use of directional arrows. Directional arrows make it easy for users to quickly determine the precedence of RWL elements. Arrows also fit well into our visual notation paradigm.

If a particular RWL model construct could possibly take part in a program flow relationship it could be represented in the meta-model by a generic *program flow* model element. Any RWL construct taking part in such a relationship could inherit from this genertic model element keeping the meta-model modular and cohesive.

Let us look at a quick example to demonstrate the *program flow* relationship.

```
1.  Code      RW_EXAMPLE
2.  Type      Standard
3.  Access    STSR
4.  Name      “Report Writer Example”
5.
6.  Print “Hello World”;
```

In the above RWL script we can see that the *Print* occurs after the *ControlLine* therefore the *Print* and the *ControlLine* statement can take part in a *program flow* relationship, although the literal

("Hello world") is part of the print statement and cannot take part in that relationship. Figure 97 below graphically represents this *program flow* relationship.

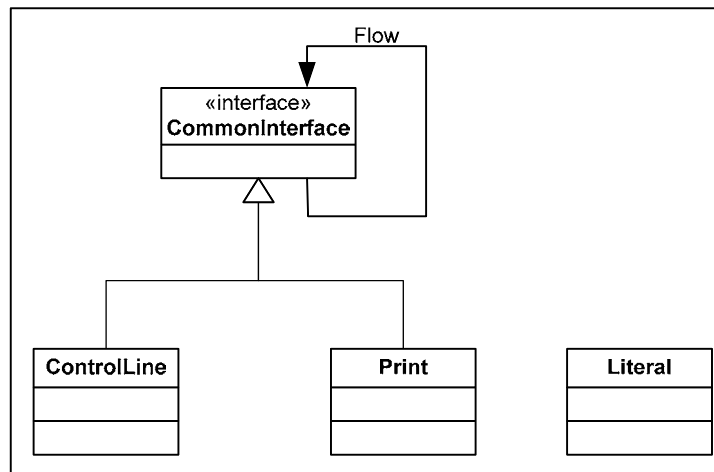


Figure 97: RWM Shell Approach: Program flow

5.8.3.4 Child element behaviour (Containment)

Providing the RWL meta-model via the shell has its advantages in terms of child element behaviour. The DSL Tools offer four core shapes: a geometrical shape, a compartment shape, an image shape and a port shape. Most RWL constructs have an inherent part-of (aggregation) structure and representing this via a containment mechanism is a very suitable visual metaphor. Therefore, to show the containment metaphor, the compartment shape does an excellent job. An example of a compartment shape is shown below in Figure 98.

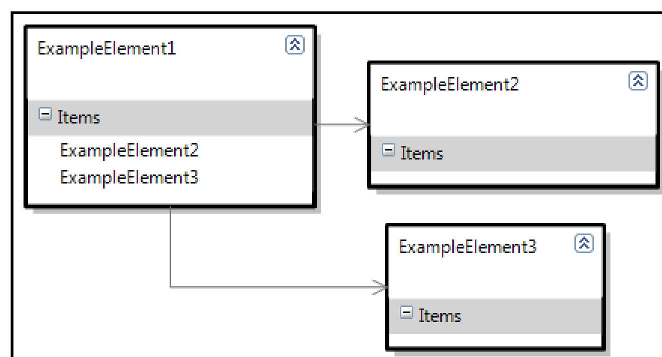


Figure 98: RWM Shell Approach: Containment

From the above figure we can see that each element is represented by a compartment shape. Note that the model element *ExampleElement1* contains two other elements, this can be shown easily within a given compartment (called *Items*) as demonstrated in Figure 98.

5.8.3.5 Child element behaviour (Ordering)

Let us look at a RWL code snippet to demonstrate the design aspect of ordering child elements.

```
1.  Print "Hello World" + "Hello World 2";
```

Looking at the above code snippet we see that a *Print* construct has two literals which are printed in a specific ordering. The requirement was to visually depict this ordering so that users can quickly determine the flow of a construct and therefore the flow of entire RWL script. We decided to depict

this ordering within our shell by placing each child element on the canvas in their expected order. Therefore a visual representation of the above script could look like Figure 99.

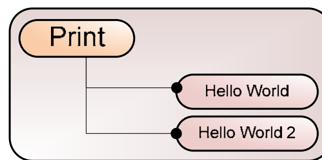


Figure 99: RWM Shell Approach: Ordering

Note that in most visual languages (e.g. UML) ordering of elements is not important as it does not have any semantic meaning although for the RWL the ordering of the elements represents the order in which they would be executed thus making it a vital part of the RWL as it represents semantics.

5.8.3.6 Child element behaviour (Show/Hide)

Due to the fact that some of the RWL construct are composite constructs and have children it was required that we can show/hide these children at command to keep the RWL model concise. The DSL Tools and the compartment shape lend themselves perfectly for this task. An example of a compartment shape was shown in Figure 98. Note the icon on the top-right hand side of the compartment shapes, this icon allows us to collapse and expand that shape. It was decided that if the user collapses a shape they are potentially asking us to hide its children and all outgoing relationships and that is what we do. Therefore when *ExampleElement1* in Figure 98 is collapsed, its children get hidden, as shown in Figure 100.

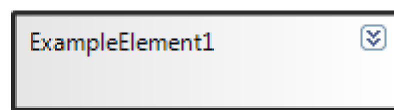


Figure 100: RWM Shell Approach: Show/Hide children

5.8.3.7 Evaluation and Testing

Evaluation of the shell was done with respect to the visual notation exposed by it which allows the end-user to create RWM models representing Prism reports. We constantly entertained feedback from the end-user and analyzed each requirement and incorporated it into our design if it was found appropriate. A formal survey was also designed which asked a range of questions and compared the visual designer to more traditional approaches of writing Prism reports. Metaphor analysis, visual paradigm analysis and cognitive dimensions were also used to evaluate our visual language.

Testing was done with respect to the generated RWL script. It was mandatory that the shell not only represents the report visually but also generates the correct RWL script corresponding to the model. We employed a manual process to test the generated RWL script although implementing an automated testing framework was considered and deemed as future work.

We have done a formal evaluation of this approach which is detailed in Chapter 8.

5.9 Design for users (non-functional perspective)

This section describes how our design caters for the non-functional requirements outlined in Chapter 4.

5.9.1 Developer

5.9.1.1 Class Diagram Approach

In this approach we have designed a highly customizable meta-meta-model which allows developers to create the RWL meta-model therefore it is inherently flexible enough to cater for newer RWL constructs. This approach is essentially designing a UML tool with some domain specific helper functions aimed to make the design the RWL meta-model easier. Moreover we have built-in a degree of automation mechanisms which allows users to easily create new meta-model elements which inherit from required base classes. We have automated method generation, event notifications, accessors and the mandatory field constraint checker. An example of this is shown in Figure 101 and Figure 102.

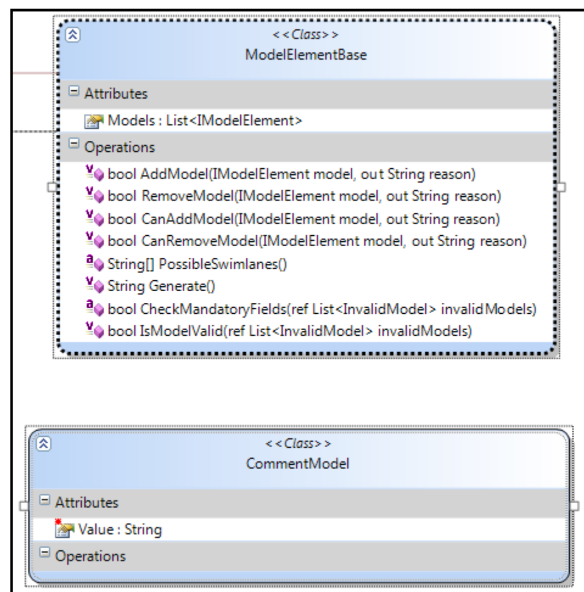


Figure 101: Class Diagram Approach: Automation for inherited elements before

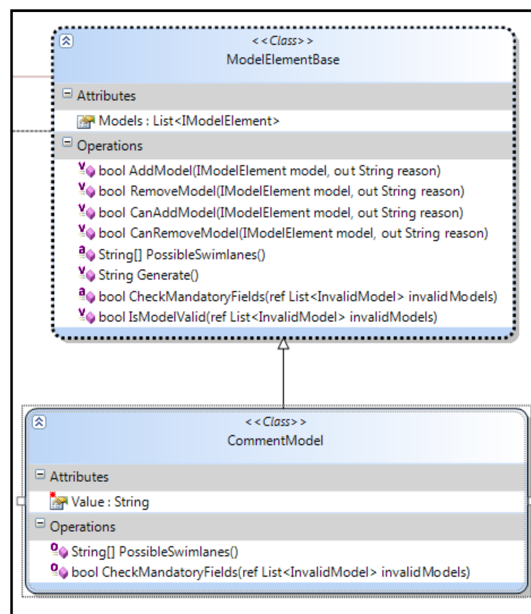


Figure 102: Class Diagram Approach: Automation for inherited elements after

The two figures above demonstrate the automation provided to developers when an inheritance relationship is added. Figure 101 shows that the developer added the *CommentModel* and to be able to demonstrate this as a RWL construct it needs to inherit from *ModelElementBase* (details of the inheritance hierarchy will be explained in Chapter 6). Therefore we provide a mechanism which extracts *abstract* methods from the parent and automatically implements them for the child, as shown in Figure 102.

We also have implemented a method generator which generates the *CheckMandatoryFields* method according to the mandatory fields indicated by the developer on the model element. For example we have a field on the *CommentModel* called *Value* which is mandatory. Therefore our generator generates the code shown below in Figure 103.

```
public override bool CheckMandatoryFields(ref List<InvalidModel> invalidModels)
{
    bool valid = true;
    List<InvalidModelReason> reasons = new List<InvalidModelReason>();
    if (String.IsNullOrEmpty(_value))
    {
        reasons.Add(new InvalidModelReason() { PropertyName = "Value", Cause = InvalidCause.EmptyValue });
        valid = false;
    }

    invalidModels.Add(new InvalidModel() { Model = this, Reasons = reasons });
    return valid;
}
```

Figure 103: Class Diagram Approach: Method generator

5.9.1.2 RWM Shell Approach

In this approach our RWL meta-model is developed within the Microsoft DSL Tools itself therefore we do not have to cater for non-functional requirements explicitly. The DSL Tools allow various hierarchical relationships between model elements which make it flexible enough to be maintained and scaled if need be. Although we do have to design our meta-model consistently so that future developers can easily understand the intent and cater for new RWL constructs if need be.

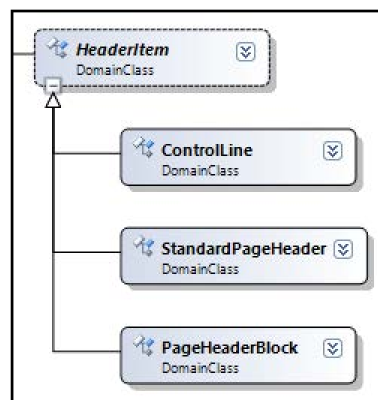


Figure 104: RWM Shell Approach: Model hierarchy

Figure 104 above shows a hierarchy within our meta-model which assists developers. If in the future another RWL construct needs to be added to the header section of a report it can simple inherit from the model element *HeaderItem*.

5.9.2 Report Designer

5.9.2.1 Class Diagram Approach

The report designer will be interacting with the RWL meta-model via a WPF UI. This UI is shown below in Figure 105.

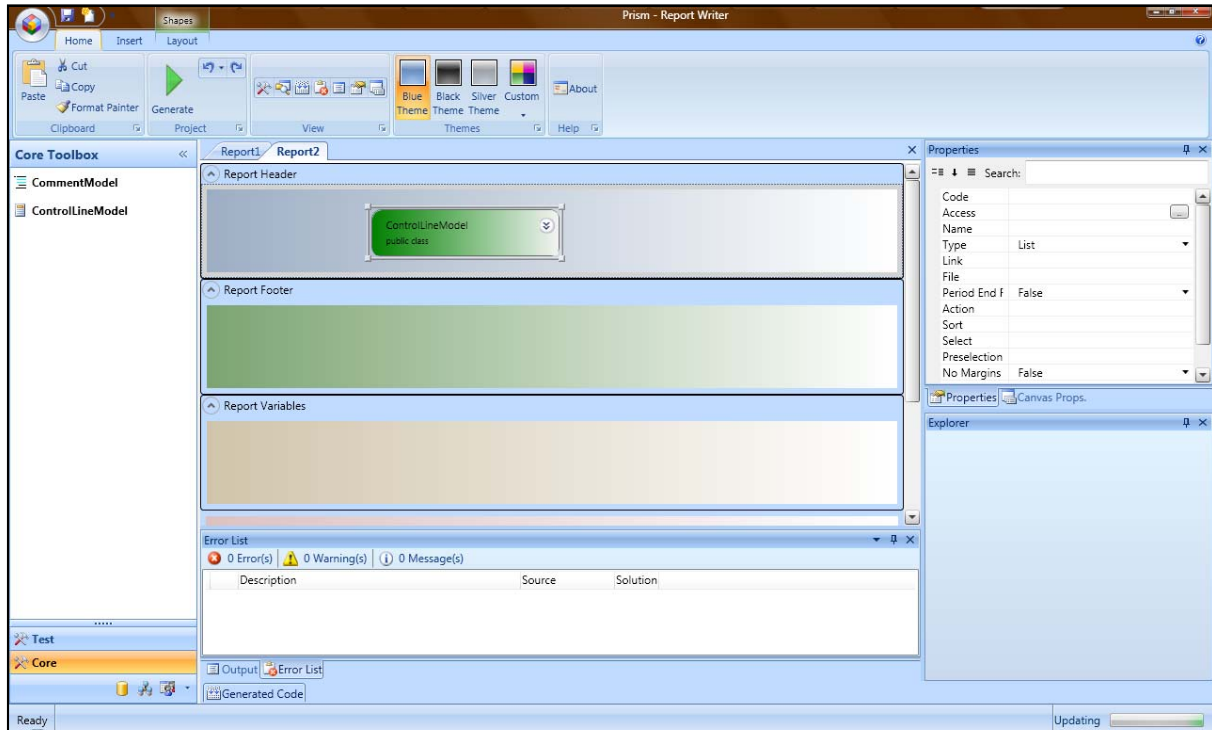


Figure 105: Class Diagram Approach: WPF UI

Note from the above figure how we have separated each section of the user interface: model toolbox on the left, canvas designer in the middle and the property editor on the right. Therefore this interface is simple and modular and is intuitive for a novice user to use without any prior training.

Each section is grouped within its own panel keeping the interface crisp and modular for cohesive user interaction. Each panel can be hidden/shown as per user needs to maximize any particular area of the interface. As per the current organisation we have a simple left to right flow (although this can be customized via the docked windows capability) where the user starts a drag from the toolbox on the left, drops it on the canvas in the centre and edits the properties of the model via the property editor on the right. Any validation errors and other output such as the generated RWL script is shown in the bottom section of the UI. We maintain consistency with the user-model for common interfaces by also providing a simple to use toolbar at the top via which end-users can perform common UI functions such as cut, copy, paste, save, etc.

5.9.2.2 RWM Shell Approach

The report designer will be interacting with the RWL meta-model via the Visual Studio Shell. This interface is shown below in Figure 106.

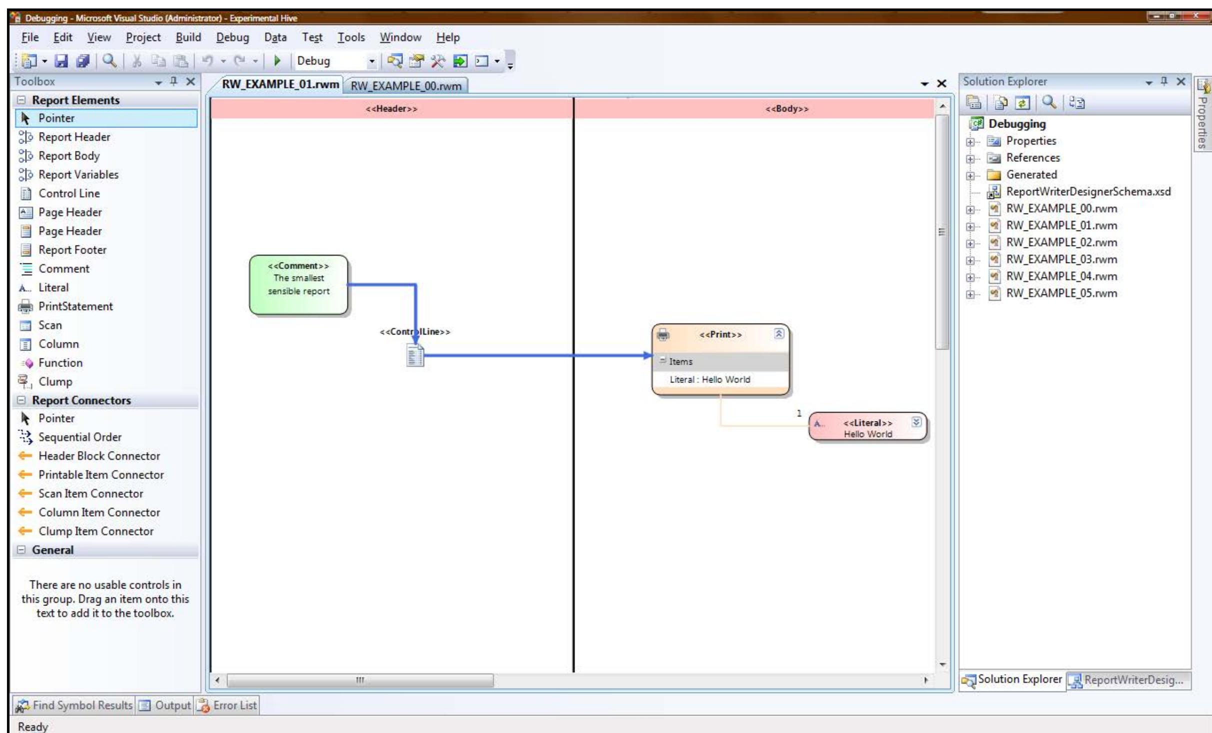


Figure 106: RWM Shell Approach: Shell UI

This interface is offered by Microsoft therefore it is intuitive, robust and modular to start with. Although we still have to design the toolbox, its icons and make the shapes intuitive.

This UI is nothing except a cut down version of the Visual Studio IDE provided by Microsoft. It offers users with a consistent user-model via which they can interact with our DSL. On top we have the menu and the toolbars to perform common functions such as cut, copy, paste, save, open, etc. The rest of the UI follows a simple left to right flow (although this can be customized via the docked windows capability) starting with the toolbox on the left which exposes the DSL meta-model elements, the canvas in the centre where the end-user creates the RWL model and the explorer/property editors to the right where the end-user can modified the instantiated model elements. Any validation errors or messages are shown in the “Error List” and “Output” windows shown in the bottom part (currently minimized) of the UI.

5.10 User Interface Design

This section mainly concentrates on the user interface design aspect of the WPF UI for our Class Diagram Approach. We briefly look at the design aspects of the UI for the RWM Shell Approach, this is because in this approach, the UI is basically provided to us by Microsoft in the form of the Visual Studio Shell (or Experimental Hive).

5.10.1 Class Diagram Approach UI

The following sub-sections describe the design decisions behind the WPF UI. Most of the decisions were taken with the help of (Spolsky, 2001).

5.10.1.1 Modern UI

The WPF UI, as the name suggests was based on WPF and used modern controls such as the ribbon toolbar and docked windows which our target user is familiar as they are exposed by the Microsoft Office Suite. A screenshot of the toolbar is shown below in Figure 107.

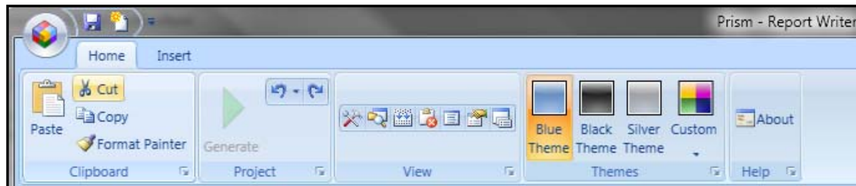


Figure 107: Modern UI toolbar

5.10.1.2 Flexible Screen Real Estate

The UI is used to design RWL models, these models can expand and become very large with time, due to this reason we needed a mechanism which allowed us to let the user concentrate on only the model if need be by hiding all the other irrelevant windows. This was done by using docking windows. A screenshot of this is shown below in Figure 108. Usage of this interface is described in the following chapter (Chapter 6) where we look at the implementation details.

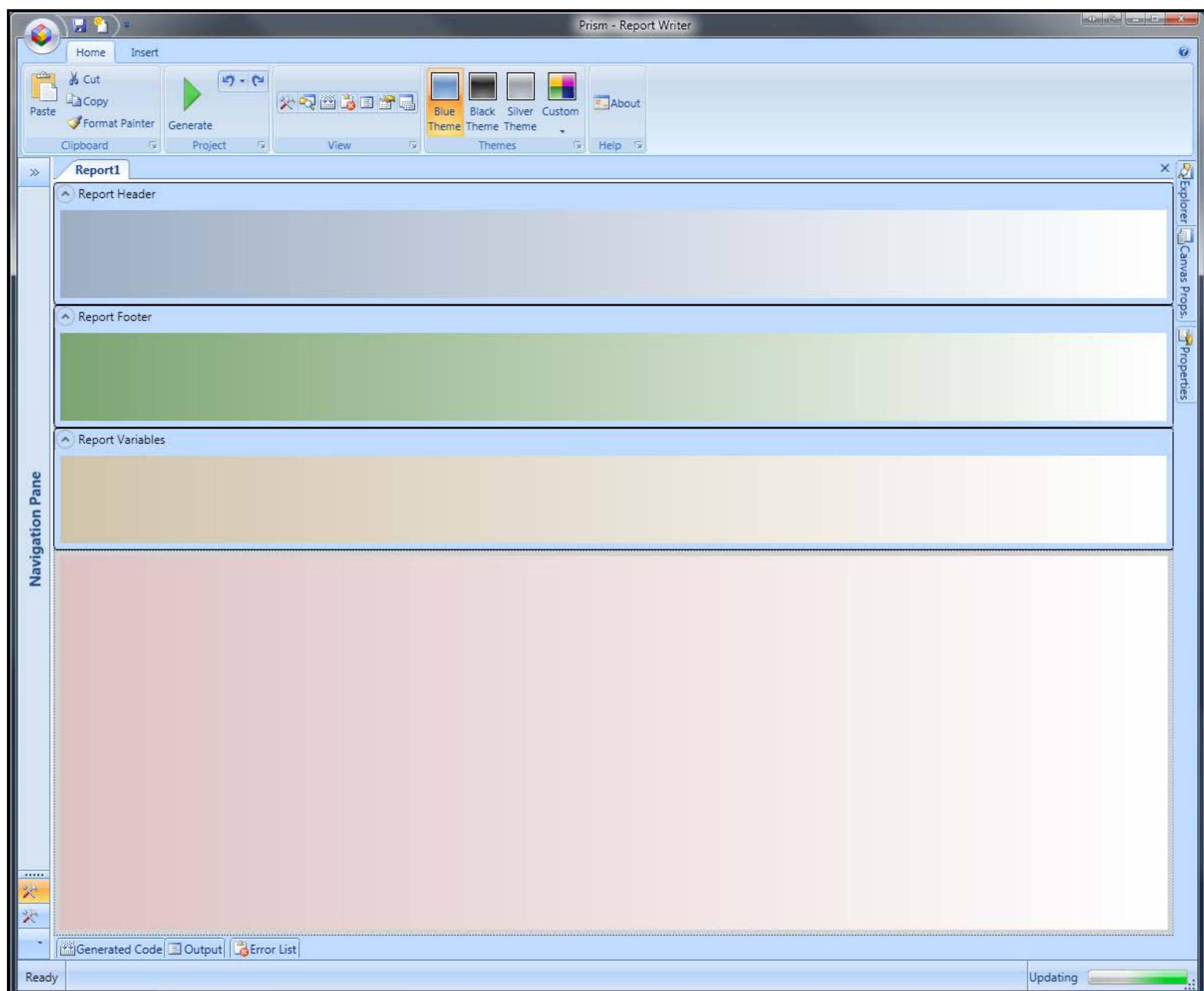


Figure 108: Docked windows

Note in the above screenshot how all the windows are hidden allowing our user to get a large view of the modeling canvas.

5.10.1.3 Help Available

The UI provides help at every instance in the form of tooltips and informative warnings and error message. An example of a tooltip is shown below in Figure 109.

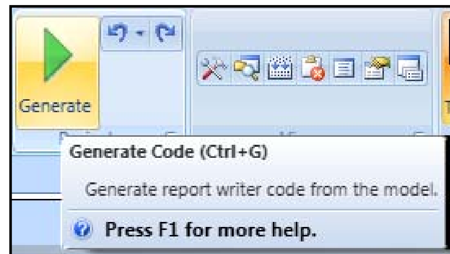


Figure 109: WPF UI tooltips (Help)

5.10.2 RWM Shell Approach User Interface

5.10.2.1 Usert Interface Overview

In the second approach (RWM Shell Approach) we use the Microsoft Visual Studio Shell to expose our DSL to the end-user. This potentially meant that we already have basic features such as open, save, cut, copy, paste, property editing, error windows, etcetera already implemented for us. Figure 106 shows the Visual Studio Shell which has a RWL model loaded into it. Note that it is very similar to our WPF UI (user interface from the Class Diagram Approach) where we have the toolbox on the left hand side where RWL model elements can be dragged from onto the canvas which is at the centre. To the right we have a property editor (currently hidden), model and file explorer. Note that the Visual Studio Shell is customizable and the end-user can drag and drop sub-windows and dock them anywhere within the parent window. Moreover we as developers can also mandate the appearance by disabling extra menu commands and toolbar buttons within the shell which may not have any relevance to our DSL.

5.10.2.2 Property Grid

The property grid provided by the Visual Studio Shell forms the integral part of our DSL and is the primary source of input as far as the end-user is concerned. The property grid always shows the properties (fields) of a selected model element. In Figure 110 below we see that the *ControlLine* model element is selected and therefore we see its properties in the property editor.

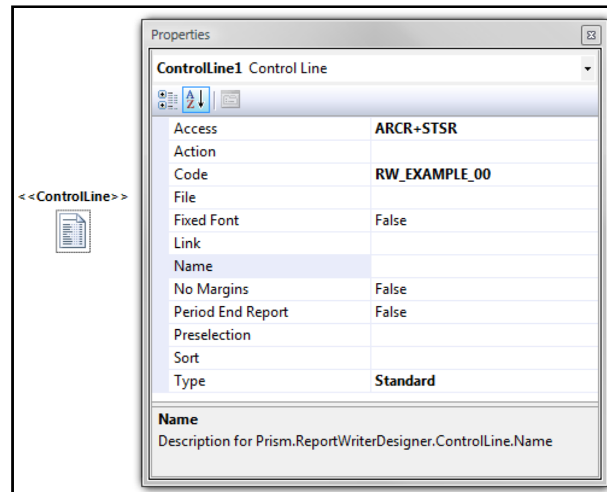


Figure 110: RWM Shell Approach: Property grid

The property grid exposes all allowable (some fields can be blocked by developer) fields of the model element and also any custom editors (Section 5.8.2.5) which a field may have.

5.10.2.3 Adding Model Elements and Connectors

The standard paradigm provided by the Visual Studio Shell to add model elements and connectors is to use a simple drag and drop operation. To add model elements the end-user is simply required to click on a model element in the toolbox and drag and drop it onto the canvas. If a drop if allowed (mandated by the constraints on the meta-model) the user sees an addition symbol along with the mouse cursor and if not allowed the user sees the not allowed symbol (both shown in Figure 111 below).

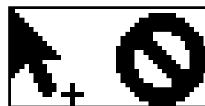


Figure 111: Allowed/Not allowed cursor icons

Creating connectors between two model elements requires a similar drag and drop operation where the end-user starts the drag at the source model element and ends the drag at the target. If a connector is allowed between these two model elements then an addition symbol is shown and if not then a not allowed symbol is shown (Figure 111).

5.10.2.4 Executing Text Templates

The Visual Studio Shell allows end-users to execute code templates to generate any textual output from the models they create using the DSL. In our case we want the end-users to be able to generate the RWL script from the RWL model. Currently we allow this via a manual process where the user needs to select the core text template file and execute a menu command (shown in Figure 112 below).

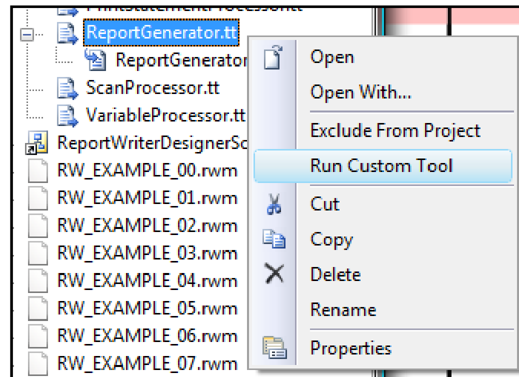


Figure 112: Executing text templates

In the above figure we see that for our prototype the core text template file is called "ReportGenerator.tt", so the end-user can right click on it and execute the menu command "Run Custom Tool" which executes the text templating engine and generates the required RWL script. As we can see that this is process is manual and in the future we envisage this process to be invoked via a custom menu item or a keyboard command. For e.g. the end-user can right click on the desired model and execute a menu command called "Generate Script".

5.11 Summary

This chapter detailed the design of the thesis in terms of our two approaches: The Class Diagram Approach and The RWM Shell Approach. We defined the core stakeholders as the developers who are mainly concerned with the meta-model of the RWL and the report designers who are mainly concerned with designing a Prism report using the UI provided.

The chapter also provided use case diagrams which gave us a graphical view at our core requirements which were then analyzed to detail the design decisions involved in the two approaches.

We also looked at how the meta-model came about from a static version of the RWL meta-model which was created with the use of simple reporting scripts. We concluded the chapter by analysing our non-functional requirements and followed by showing a high level architecture of the two approaches and eventually outlining the user interface design for our Class Diagram and RWM Shell Approach.

Chapter 6 - Implementation

6.1 Introduction

This chapter provides a detailed implementation view into the various aspects of the thesis. We look at the implementation details of each of our approaches and analyze code patterns, implementation class diagrams and event chains. We also look at details about how we implement business rules and access the Prism database meta-data.

The prototype visual report writing tools were implemented with C#, one of the .NET CLR languages and the other technologies involved were WPF: for user interface implementation and LINQ: for the database access layer.

6.2 DSL Terminology

The implementation of the thesis heavily relies on the terminology unique to the Microsoft DSL Tools. This section looks at all the relevant terms and tries to give a brief introduction into what each of these mean. The main rationale for this section is to give the readers enough information to allow them to recognize the details about the implementation. We also looked at why we used the Microsoft DSL Tools in Section 3.3.1.1 and also did an empirical comparison with Marama in Section 2.5.3. To reiterate the primary reason behind the use of Microsoft DSL Tools was to ease the integration of Prism WIN (MIS system) with our newly developed tool as both are implemented in Microsoft .NET languages.

6.2.1 The DSL Project

Creating a new Visual Studio DSL solution creates two aspects: The DSL Project and the DSL Package Project. We will be looking into the DSL Package Project in the following sub-section. Let us look at The DSL Project closely.

The DSL Project defines the actual domain specific language and its specification (Microsoft, 2007). It encapsulates the actual DSL file (*.dsl) which contains all the necessary information needed to create an executable domain specific language. Moreover this project provides the developer with an editable interface to this (*.dsl) file. The figure below shows the organisation of The DSL Project.

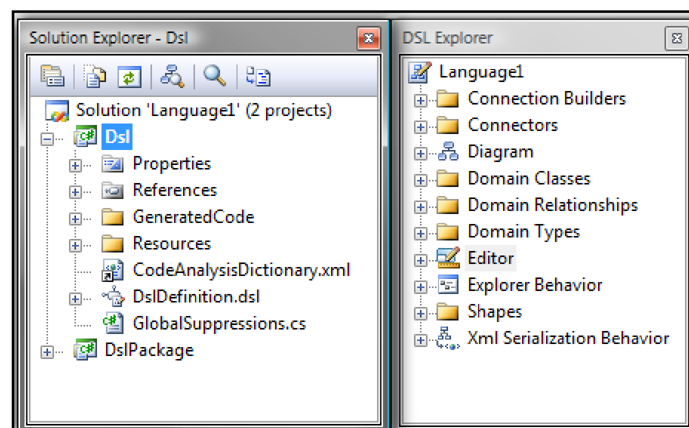


Figure 113: DSL Solution: The DSL Project

Via The DSL Project we can add/edit domain model elements, relationships, shapes, Element Merge Directives (EMD), toolbox elements, explorer view behaviour, govern how the model is serialized and define connection builders. Each of these aspects is also defined in the following sub-sections.

6.2.2 The DslPackage Project

The DslPackage Project is the second aspect of the Visual Studio DSL Solution. The DslPackage mandates how our newly created DSL Project integrates into the Visual Studio Experimental Hive or the Visual Studio Shell (Microsoft, 2007). It specifies the menu items, context menu items, toolbar buttons and the templates which are available to us at runtime via the Hive or the Shell. The organisation of the DslPackage Project is shown below in Figure 114.

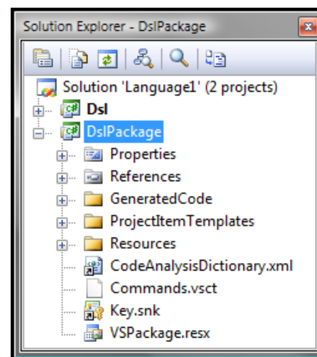


Figure 114: DSL Solution: The DslPackage Project

6.2.3 Path Syntax

Microsoft DSL Tools uses a unique path syntax to help developers navigate their way around the model. Let us look at an example to best explain it.

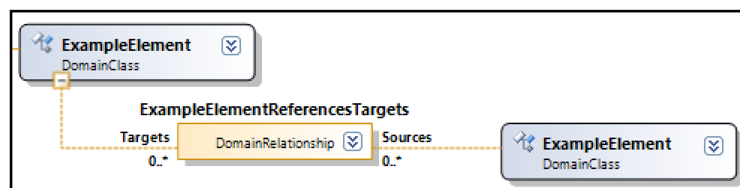


Figure 115: Path syntax example

Figure 115 above shows the *ExampleElementReferencesTargets* relationship between two *ExampleElement* model elements. *ExampleElementReferencesTargets.Targets/ ExampleElementReferencesTargets!Target* is an example of the path syntax matching the above figure.

Each segment represents a step from either an element to a relationship (*Relationship.Property*) or from a relationship to an element (*Relationship!Role*).

Minute details about the path syntax are not relevant for this thesis.

6.2.4 The Toolbox

The toolbox allows the end-user to drag-drop model elements onto the designer canvas. The toolbox can contain essentially two items: shapes and connectors. Shapes represent model elements whereas connectors represent relationships between model elements. Any model element or a relationship which needs to appear in the toolbox requires a matching tool item. A tool

item simply maps the toolbox item to its respective model element. An example of a toolbox item is shown below in Figure 116.

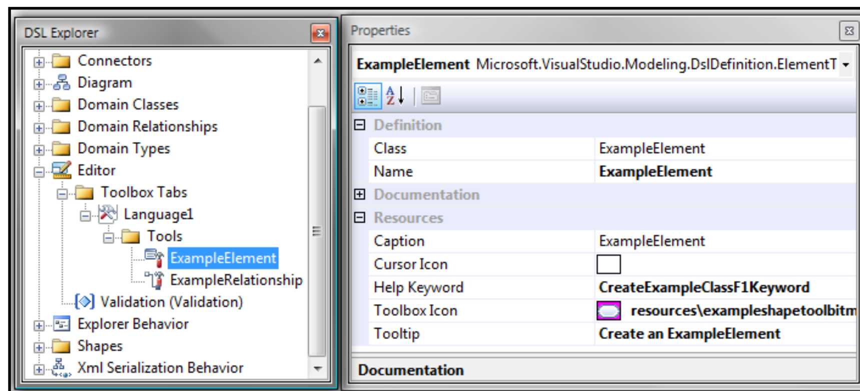


Figure 116: Toolbox creation example

In the above example the *ExampleElement* model element is mapped to the toolbox item. On the right hand side of the figure we see the various properties of this toolbox item. We can set its caption, tooltip and the icon which it appears as within the toolbox. The resulting toolbox is shown below in Figure 117.

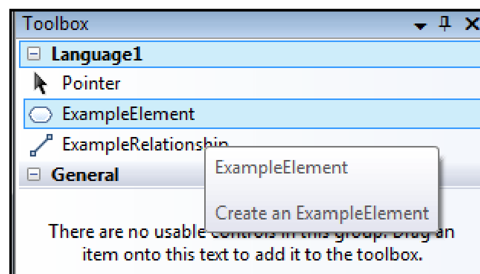


Figure 117: Toolbox example

6.2.5 Element Merge Directives

An Element Merge Directive (EMD) determines what happens if a domain element is merged with another domain element (Cook, Jones, Kent, & Wills, 2007). A merge in the DSL sense occurs when:

- User drags and drops a shape from the toolbox onto the design surface or onto a shape already existing on the design surface
- User creates an element using the menu provided via the model explorer
- User adds an item to a compartment shape
- User moves an item from one swimlane to another
- Custom code invoking a merge directive

An example of this within our Class Diagram Approach would be when a *ModelClass* is dragged and dropped onto a *ModelInterface*. This would essentially mean that the DSL user is trying to make the *ModelClass* inherit from the *ModelInterface* so the EMD automatically forms the relationships needed between them.

An example from the RWM Shell Approach would be when the end-user drags and drops a *Column* model element onto a *Print* model element. We can assume that they are actually trying to

represent the column being printed, so the EMD forms the appropriate relationships between the *Column* and the *Print* model elements.

6.2.6 The Model Explorer

The Model Explorer provides a tree-view perspective to the model elements. Each node in the explorer is the result of an embedding relationship which was introduced in Section 5.7.2.2. By default the name of the model element appears in the explorer although this can be configured to be any valid property. We see an example of the explorer configuration and then the actual explorer in figure blah and blah respectively.

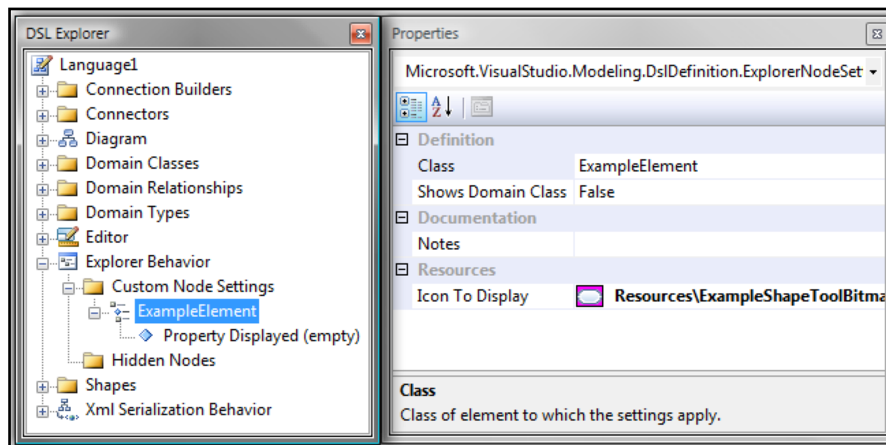


Figure 118: Model Explorer configuration

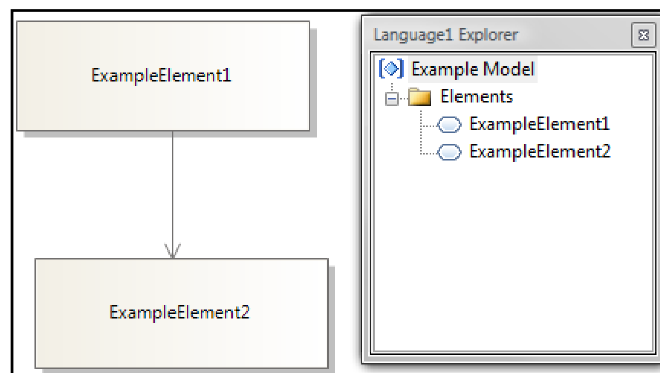


Figure 119: Example Model Explorer

On the left hand side of Figure 119 we see two model elements which are part of the model and these are represented in the Model Explorer as two child nodes of the main model (*Example Model*) indicating that each of these model elements is embedded within the main model.

6.2.7 Connection Builders

A connection builder is invoked by the connections tools on the toolbox. These builders allow connecting two model elements depending on where the user started and ended the connection. For example, the connection we see between *ExampleElement1* and *ExampleElement2* in Figure 119 is managed by the connection builder shown in Figure 120.

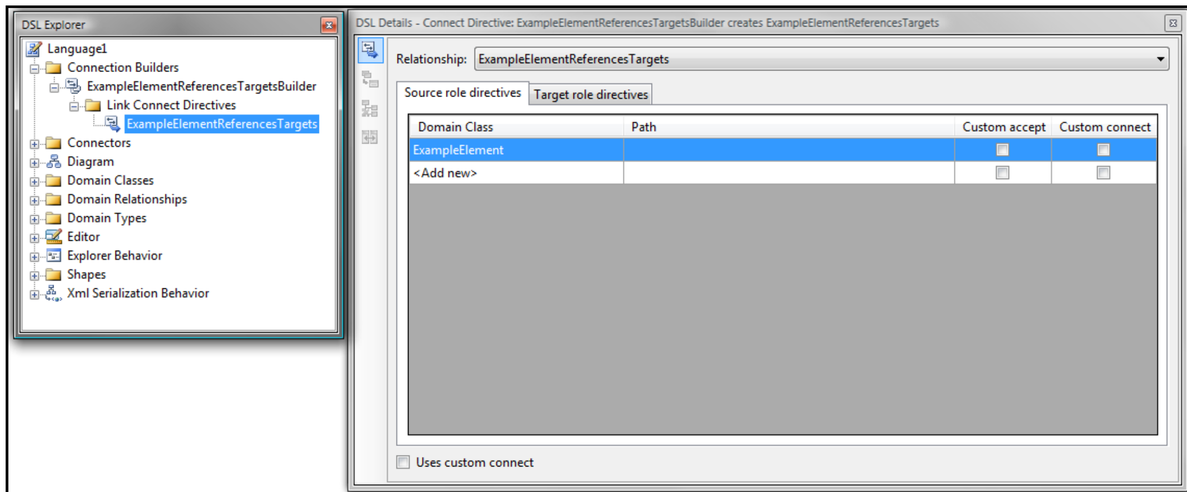


Figure 120: Connection builder

From the above figure we can see the connection builder *ExampleELementReferencesTargetsBuilder* on the left hand side and its properties on the right hand side. From the properties of the connection it can be seen that this connection builder creates a relationship of type *ExampleElementReferencesTargets* which was introduced in Figure 115. Also note that the connection builder has a notion of where the connection was started (*Source role directives*) and where it was finished (*Target role directives*).

6.2.8 External Types

External types allow the DSL designer to add custom classes which can be utilized by the domain model. External types can be either an enumerated value or an object. Both examples are shown in Figure 121 (*ExampleEnumerand* and *ExampleObject*).

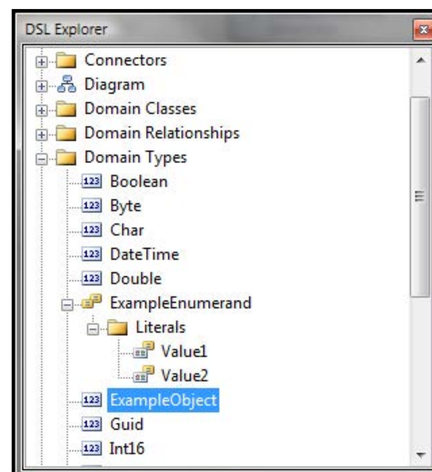


Figure 121: External types

After an external type is defined we can use it within any model element. Figure 122 shows a newly added domain property on model element *ExampleElement* which is of type *ExampleEnumerand*.

Figure 122: External type usage

Within the Class Diagram Approach we use several custom enumerated values. An example of this is to represent the accessibility of a ModelClass. The DSL designer can select whether a ModelClass is *public*, *private* or *internal*. These three values are represented via a custom enumerand.

Within the RWM Shell Approach we use enumerands extensively to represent a closed set of values which the end-user can select from. An example of this is the *ReportType* field within the *ControlLine*. The type of report an end-user can design could be either *Standard*, *List*, *General*, *Period* and *Summary*. These three values are represented via a custom enumerand.

6.2.9 Serialization

The DSL Tools offer an automatic and customizable mechanism for us to serialize a given model and its corresponding layout information. This allows the end-user to save and load a model file. The DSL Tools serialize the file using an XML format. The details about the format are irrelevant for this thesis. The generated serialized files for the model shown on the left hand side of Figure 119 are shown below in Figure 123 and Figure 124.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <exampleModel dslVersion="1.0.0.0" Id="51912076-9fe4-4fdf-a551-c791e7a719f2" xmlns="http://schemas.microsoft.com/dsltools/Language1">
3    <elements>
4      <exampleElement name="ExampleElement1">
5        <targets>
6          <exampleElementMoniker name="/51912076-9fe4-4fdf-a551-c791e7a719f2/ExampleElement2" />
7        </targets>
8      </exampleElement>
9      <exampleElement name="ExampleElement2" />
10   </elements>
11 </exampleModel>
```

Figure 123: Serialized model information

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <minimalLanguageDiagram dslVersion="1.0.0.0" absoluteBounds="0, 0, 11, 8.5" isCompleteView="false" name="Sample">
3    <exampleModelMoniker Id="51912076-9fe4-4fdf-a551-c791e7a719f2" />
4    <nestedChildShapes>
5      <exampleShape Id="bd7c707e-1dcc-48ee-a11a-600a1d478f4f" absoluteBounds="4.75, 0.625, 2, 0.75">
6        <exampleElementMoniker name="/51912076-9fe4-4fdf-a551-c791e7a719f2/ExampleElement1" />
7      </exampleShape>
8      <exampleShape Id="bef3338d-ad2b-4324-b1b3-2af911c763f2" absoluteBounds="4.875, 2.25, 2, 0.75">
9        <exampleElementMoniker name="/51912076-9fe4-4fdf-a551-c791e7a719f2/ExampleElement2" />
10     </exampleShape>
11     <exampleConnector edgePoints="[(5.8125 : 1.375); (5.8125 : 2.25)]" fixedFrom="Algorithm" fixedTo="Algorithm" TargetRelationshipDomainClassId="a229ab47-c544-4ae1-bcee-3ad5d81533d1">
12       <nodes>
13         <exampleShapeMoniker Id="bd7c707e-1dcc-48ee-a11a-600a1d478f4f" />
14         <exampleShapeMoniker Id="bef3338d-ad2b-4324-b1b3-2af911c763f2" />
15       </nodes>
16     </exampleConnector>
17   </nestedChildShapes>
18 </minimalLanguageDiagram>
```

Figure 124: Serialized model layout information

The way a particular model is serialized can be changed by the DSL developer if need be via the DSL Tools shown below in Figure 125. For the implementation of the solution for this thesis we have not customized the serialization behaviour therefore details on how to go about the changes are not discussed further.

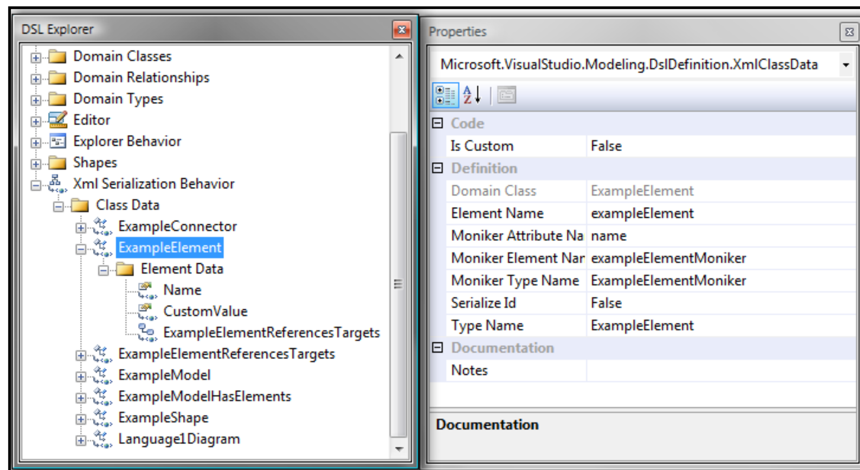


Figure 125: Customizable serialization

6.2.10 Text Templates

We introduced the concept of text templates for code generation in Section 2.5.1.5, this sub-section looks at some of its details. Figure 127 shows a simple text template based on the model shown in Figure 126.

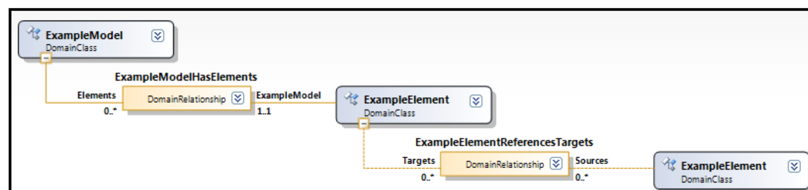


Figure 126: Text template example model

```
1  <#@ template inherits="Microsoft.VisualStudio.TextTemplating.VSHost.ModelingTextTransformation" #>
2  <#@ output extension=".txt" #>
3  <#@ Language1 processor="Language1DirectiveProcessor" requires="fileName='Sample.mydsl1'" #>
4
5  Generated material. Generating code in C#.
6
7  <#
8    // When you change the DSL Definition, some of the code below may not work.
9
10   foreach (ExampleElement element in this.ExampleModel.Elements)
11   {
12 #>
13     <#= element.Name #>
14 <#
15   }
16 #>
17
```

Figure 127: Text template

The above text template outputs a file with a *.txt* extension and requires model file called *Sample.mydsl1*. It outputs the *Name* property of all the *ExampleElements* in an instantiated *ExampleModel*.

The output of the template shown in Figure 127 executed against the model shown in the left hand side of Figure 119 is shown below in Figure 128.

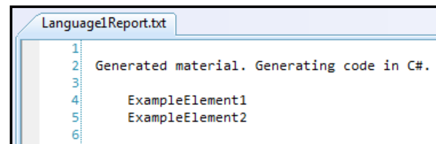


Figure 128: Text template output

Note the file extension and formatting of the output shown in the figure above. The formatting is identical to that within the text template.

6.3 Class Diagram Approach Implementation

The Class Diagram Approach was introduced in Section 3.4.2.1. This section looks further into the details of that approach and the how it was implemented. The Class Diagram Approach created a meta-meta-model tool using the Microsoft DSL Tools which allowed us to create the RWL meta-model which was consumed by the WPF UI to create RWL models, therefore this section will be divided up into three main sub-sections, each representing a different stage in the implementation process. The three main stages of implementation (Meta-Meta-Model Development, Meta-Model Development and Shell Host Development) for the Class Diagram Approach are formed with respect to the overall high level approach defined in Section 3.4.2.1 (shown below).

1. Design a meta-meta-model structure using Microsoft DSL Tools
2. Design our RWL meta-model using the newly created meta-meta-model
3. Generate code representing our meta-model using text templates
4. Design the end-user UI using WPF which uses the generated code representing the meta-model as its back-end
5. The WPF UI generates the required RWL script

The first point (1) in the above list is the meta-meta-model development (Section 6.3.1). Point two and three form the meta-model development phase (Section 6.3.2). Point four and five form the shell host development phase (Section 6.3.3).

6.3.1 Meta-Meta-Model Development

The meta-meta-model we implemented was essentially a UML-like meta-model which gave us the ability to create highly configurable class diagrams which we could then use to create the RWL meta-model. The DSL Tools allow us to create this UML-like meta-model via its Class Diagram wizard. This UML-like meta-model is essentially our meta-meta model.

We started with a DSL model as shown in Figure 73. This model already allowed us to create class diagrams. The following sub-sections detail the steps taken to make this generic class diagram model into our meta-meta-model.

6.3.1.1 Common Hierarchy

The class diagram wizard is intelligent and had a pre-built hierarchy for us. This hierarchy has an abstract root called *NamedElement* which all model elements inherit from, either directly or indirectly. This model element has a property called *Name* which holds the name of the model element when instantiated. Therefore, we can make our new model elements inherit from this model element and we will automatically have this property. This hierarchy is shown below in Figure 129.

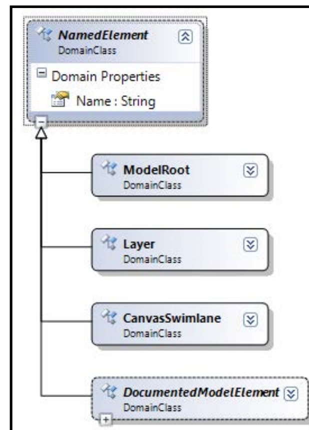


Figure 129: Class diagram common hierarchy

6.3.1.2 Layer Support

Looking at the static UML meta-model designed in Step N of Section 5.4, we have two distinct swimlanes. We wanted to depict this in our meta-model therefore we added layer support. We determined that any meta-model element would belong to one and only one layer and our layer would in-turn belong to the root of the model. Each layer would be mapped to a swimlane shape so it appears as a vertical band just like in our static meta-model design. Therefore we have the following model elements and relationships, shown in Figure 130.

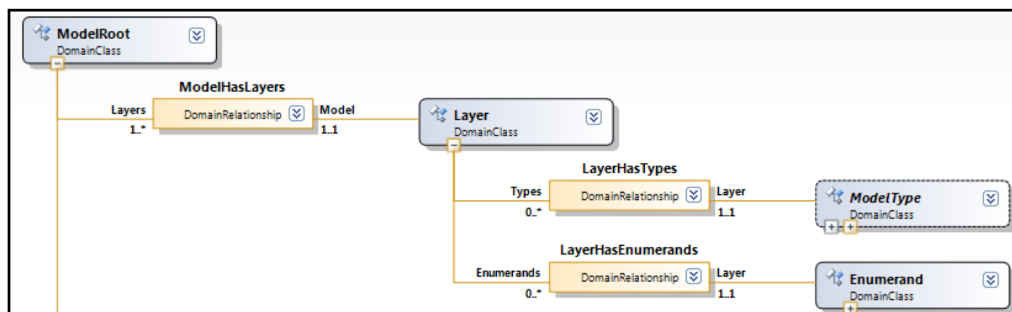


Figure 130: Layer support

Note in the above figure that all relationships are embedding relationships (denoted with a solid line and introduced in Section 5.7.2.2) meaning that all *ModelType* and *Enumerand* elements have to embedded on one *Layer*. The *Enumerand* model element is explained in the following sub-section (Section 6.3.1.3) although let us look at the *ModelType* model element.

We have designed the *Layer* model element so that all elements are embedded within it. In a class diagram we have three main types of objects. Namely: enumerands, classes and interfaces. We have already shown how the *Enumerand* object is embedded within the *Layer* but have not shown how classes and interfaces are embedded within it. For this we have to look closer at the *ModelType* model element. Note that the surrounding border of the *ModelType* model element is dashed, indicating that this is an abstract class. Expanding this, we have the hierarchy as shown in Figure 131.

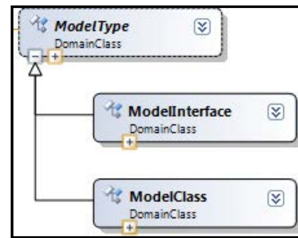


Figure 131: ModelType inheritance

From the above figure we can see that the *ModelInterface* and *ModelClass* model elements both inherit from *ModelType* meaning that both these elements display the same behaviour as its parent, *ModelType*. Thus, all three class diagram objects: enumerands, classes and interfaces have to be embedded within a *Layer* model element. This conforms to our static UML diagram specifications.

6.3.1.3 Enumerand Support

We wanted to be able to add enumerated values to our meta-model and our existing class diagram created by the wizard did not support this. Enumerated values would allow the DSL designer to easily select a value from a closed set of available values. This means that validation could be made simpler as there are only a few values to choose and also the DSL designer would not need to know the valid values which can be used as they are readily available within the enumerated list.

Therefore, we added enumerand support into our model by representing it using the model element *Enumerand*, shown in Figure 130. An enumerand also needs a name, although this has already been taken care of as we inherit from *NamedElement* as explained in Section 6.3.1.1.

An enumerand needs to be able to represent various enumerated values, we added support for this by adding another model element called *EnumerandValue* which is embedded within the *Enumerand* model element and the generated relationship is shown below in Figure 132.

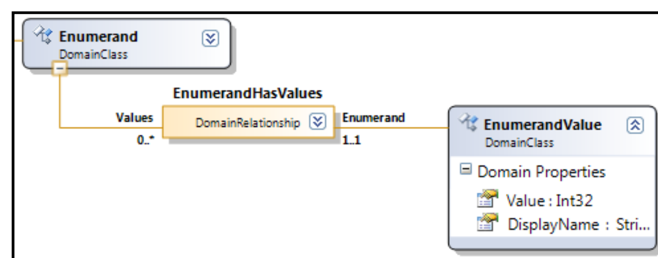


Figure 132: Enumerand support

From the above figure we can quickly notice that an enumerand can contain 0..* (zero to many) enumerated values and that an enumerated value has to belong to one and only one (1..1) enumerand. Note that the *EnumerandValue* model element has two properties, *Value* and *DisplayName*, Figure 133 below shows what they represent.

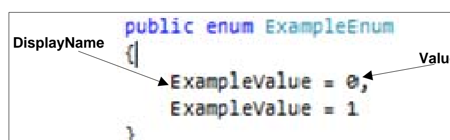


Figure 133: EnumerandValue properties

6.3.1.4 Canvas Swimlane Support

The notion of canvas swimlanes was borrowed from Scribe (outlined in Section 2.6). Scribe made report writing modular by allowing the report designer to divide up a report into various sections such as the header, variables and body. We attempt to make a similar distinction by dividing up a given canvas into various sections called canvas swimlanes. We further make a distinction by allowing our meta-model (which will be designing using this meta-meta-model) to dictate which classes (*ModelClass*) can be added to which canvas swimlane (*CanvasSwimlane*). This relationship (*CanvasSwimlaneMapesClasses*) is shown below in Figure 134.

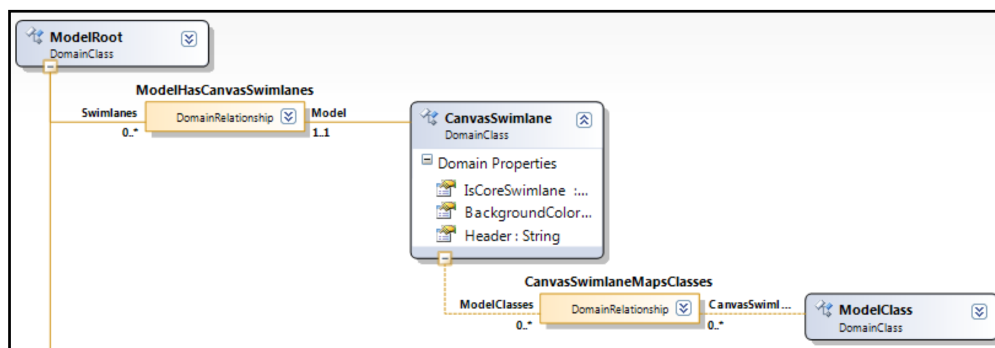


Figure 134: Canvas swimlane support

Note from the above figure that the *CanvasSwimlane* model element is embedded within the root of the model as opposed to within the *Layer* model element. This is because ModelClass elements belonging to different *Layer* model elements could be potentially mapped to a common *CanvasSwimlane*.

Also note the properties of the *CanvasSwimlane* model element. The *IsCoreSwimlane* property allows us to indicate in the meta-model whether a given swimlane is the main swimlane so we can allow for special conditions. *BackgroundColor* and *Header* property allow the meta-model to indicate the colour of the swimlane and its title respectively.

A usage example of this is shown in the meta-model generated in Figure 157 and this meta-model get translated to the report sections (*Report Header*, *Report Footer*, *Report Variables* and *Report Body*) which are shown in Figure 108.

6.3.1.5 UML Constraints

The meta-model which will be mapping the RWL will essentially be a UML like diagram following the meta-meta-model described here. Therefore it is mandatory that our meta-meta-model enforces constraints which will mandate the creation of a valid meta-model. These constraints are illustrated in this sub-section.

- *Validations*

  Like any valid UML diagram, every class which will be part of our meta-model will have to contain a unique name which is not empty. More so, the unique name should adhere to the final implementation language constraints. For e.g. the generated code from our meta-model will be in C#, therefore class names can only start with alphabets and then contain any alphanumeric character and cannot contain a space.

Therefore, what we did was design a validation rule on the *NamedElement* model element which validates its *Name* property. We implemented a regular expression validator which validated the C# requirements and added custom code which guaranteed that a class name is unique. The event chain for this validation is shown below in Figure 135.

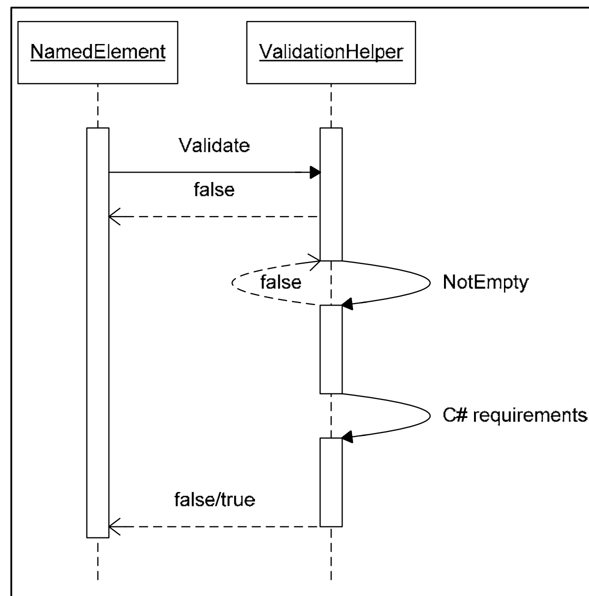


Figure 135: Class Diagram Approach: Validation implementation

If any of these rules are violated we see intuitive error messages which are shown in Figure 136.

Figure 136: Class Diagram Approach: Validation results

- *Inheritance cycle*
  In .NET CLR programming languages recursive inheritance hierarchies prove to be a major problem especially when we have deep inheritance. To avoid this we decided to add a validation rule which potentially checks for recursive inheritance cycles and informs the user.

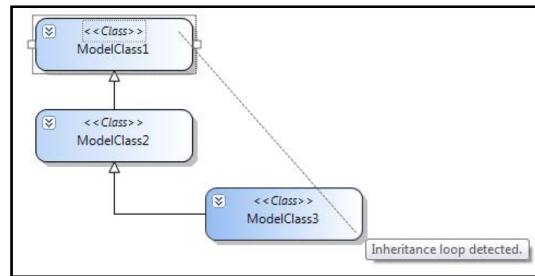


Figure 137: Class Diagram Approach: Inheritance cycle

In the above figure we are potentially trying to create an inheritance from *ModelClass1* to *ModelClass3* (indicated by dotted line), this will to a recursive hierarchy clearly see from the figure, although when the meta-model gets complicated and with deeper hierarchies this will not be clear.

- *Self inheritance*
  It is also irrational allowing a class or an interface to inherit from itself (indicated by dotted line in Figure 138) this is also catered for by a validation rule. The figure below shows this validation rule in action.

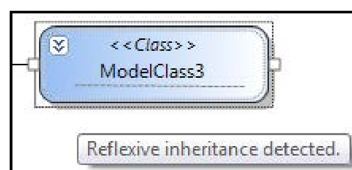


Figure 138: Class Diagram Approach: Self inheritance

- *Multiple inheritances*
  Our intended implementation language, C#, disallows a class to inherit from more than one class (note that a class can inherit from more than one interface). We have to cater for this requirement, although this can be done via our meta-meta-model, shown below in Figure 139.

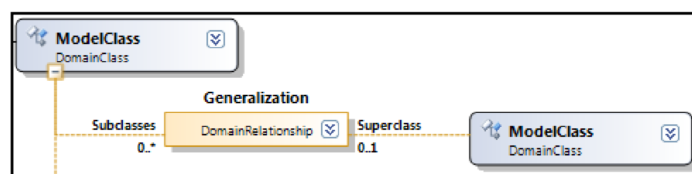


Figure 139: Class Diagram Approach: Multiple inheritances

Note in the above figure the *Generialization* relationship denotes a class inheriting from a class. Also note the multiplicity on the *SuperClass* role player is 0..1, this means that a class can have at most one superclass (parent).

6.3.1.6 Custom Field Editors and Mandatory Fields

Section 5.7.2.5 introduced us to the need for custom field editors. We wanted to allow developers to choose whether a specific field or attribute has a specialized editor attached to it. This allows the WPF UI to give end-users a sophisticated interface which they can be used to populate the value for that field.

We implemented this by adding an external enumerand (Enumerands were introduced in Section 6.2.8) called *EditorType*. This enumerand at this point in time (new values can be added if need be) contained two values: *DefaultEditor* and *WindowSelectionEditor*. We then added a property to our *Attribute* meta-meta-model element and the type of this attribute was mapped to our newly create enumerand *EditorType* (demonstrated below in Figure 140). This allowed the meta-model developer to indicate whether a given attribute is edited by a specialized editor (currently only the *WindowSelectionEditor*).

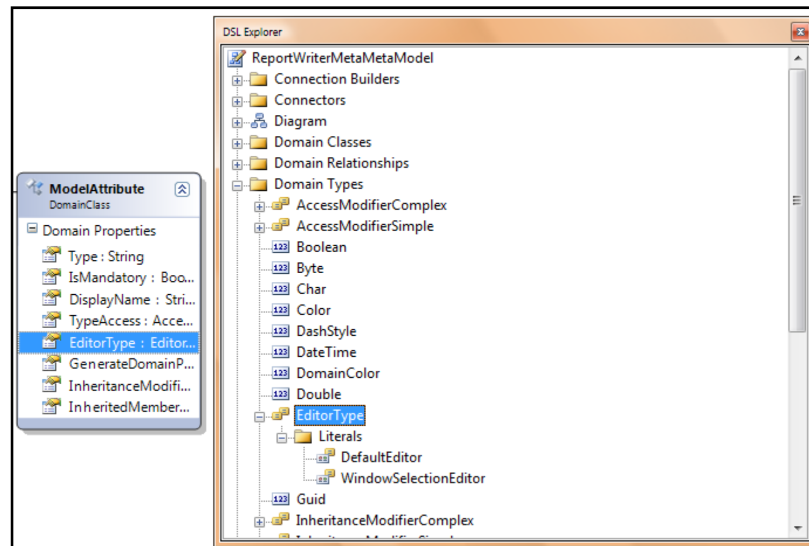


Figure 140: Class Diagram Approach: Custom field editor implementation

Whether a given attribute was mandatory or not was easily implemented by adding a field on the *ModelAttribute* model element called *IsMandatory*. This is a Boolean type, true if the field is mandatory and false otherwise. This field is also shown in the above figure.

6.3.1.7 Custom UML Editors

While creating the meta-model of the RWL the developer will be constantly be adding and modifying methods (functions/operations). To make this easier for the developer a simple operation signature editor was provided by the meta-model. This was implemented in WPF and is shown below in Figure 141.

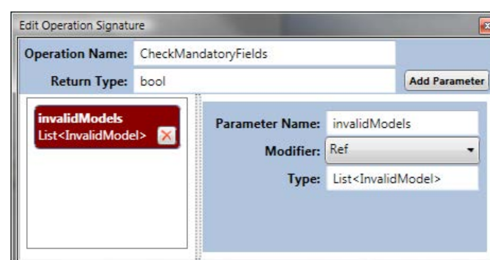


Figure 141: Class Diagram Approach: Operation signature custom editor

The above interface has validation control and enforces simple rules such as disallowing empty values for parameter and operation names and ensuring they meet the C# language constraints as explained in Section 6.3.1.5.

6.3.1.8 Shape/Connector Definition

The shape definition of the meta-meta-model elements was kept consistent with the way the .NET class diagrams work to reduce the learning curve for new developers. Figure 142 below shows the shape mappings for the *ModelClass* and *ModelInterface* model elements.

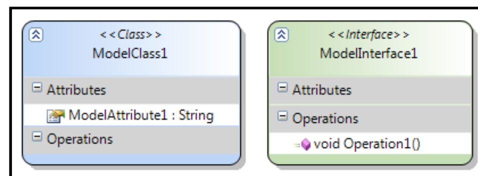


Figure 142: Class Diagram Approach: Shape definition

The rest of the model elements follow a similar shape mapping where on the top left side we have an expandable/collapsible button following by a stereotype definition and then the name of instantiated object. Most shapes also have compartments to represent their embedding fields. For e.g. in the above figure, both the class and the interface have compartments which show their attributes and operations.

We have also implemented minor variations on how the shape looks depending on some properties. For e.g. if a class in the meta-model is abstract, it has a dashed border. If a given operation is an override or virtual operation it has a special icon next to it. If an attribute is marked as mandatory, it appears with a red asterix. Examples of the aforementioned are shown below in Figure 143.

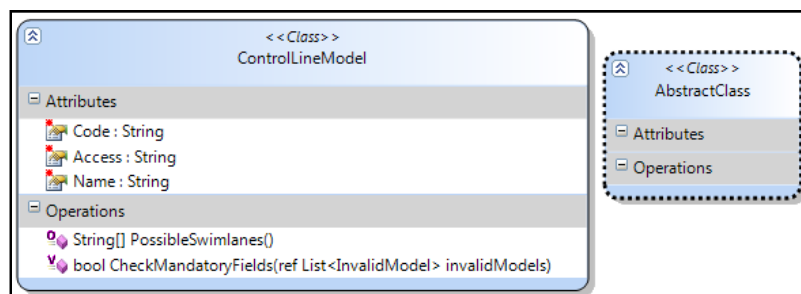


Figure 143: Class Diagram Approach: Shape definition variability

All relationships (embedded/aggregation and reference/association) were represented using lines. We use directional arrows for some relationships such as generalization. Figure 144 shows an example of the *CanvasSwimlaneMapsModelClasses* relationship; we see four lines going between *CommentModel* to the *CanvasSwimlanes* indicating that this particular class can be successfully added to these swimlanes.

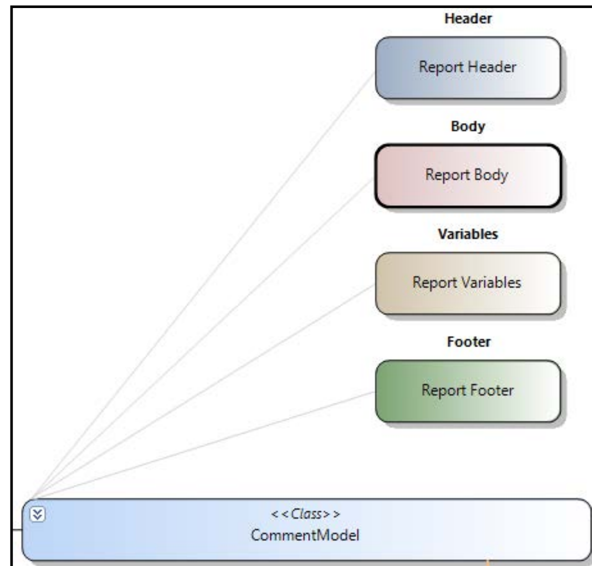


Figure 144: Class Diagram Approach: Connector definition

6.3.1.9 Custom Context-Menu Support

We introduced the DslPackage project in Section 6.2.2 which specifies the menu items and commands available to the developer while they are using our meta-meta-model.

CanvasSwimlanes and its relationship with a ModelClass were outlined in Section 6.3.1.4 and demonstrated in Figure 144. As the meta-model grows the number of instantiated ModelClass elements will increase and so will the number of relationships it has with the respective canvas swimlanes. Each of these will be represented by a line which will clutter the meta-model, to get around this problem we introduced an extra menu item which the meta-model developer can toggle to hide and show these relationships. This menu item (called Toggle Shape Maps) is shown in the figure below, notice how the relationships between the class and the swimlanes are not shown. These relationships still exists on the underlying meta-model only their representation is invisible to avoid clutter.

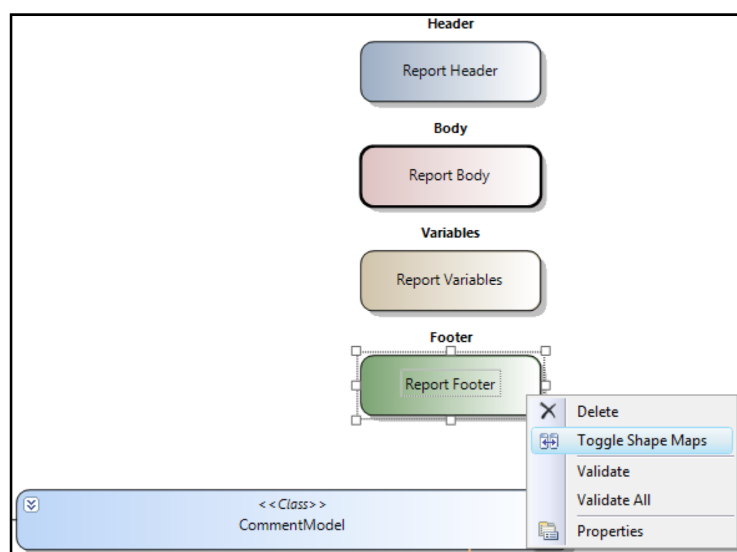


Figure 145: Class Diagram Approach: Custom context-menu

We implemented this menu item by simply adding a new value into the Commands.vsct file within the DslPackage project. The VSCT (Visual Studio Command Table) file allows developers to add custom commands to the visual studio menu structure. The VSCT is an XML file making it very intuitive. The code snippet of this file which implements the new menu item shown in Figure 145 is shown below in Figure 146.

```
17  <Commands package="guidPkg">
18      <Bitmaps>
19          <Bitmap guid="guidImages" href="Resources\swimlaneclassmap.png"
20                  usedList="toogleShapeMaps"/>
21      </Bitmaps>
22      <Buttons>
23          <Button guid="cmdToggleShowShapeMapsGuid" id="cmdToggleShowShapeMapsId" priority="0x0902" type="Button">
24              <Parent guid="guidCmdSet" id="grpidContextMain"/>
25              <Icon guid="guidImages" id="toogleShapeMaps" />
26              <CommandFlag>DynamicVisibility</CommandFlag>
27              <Strings>
28                  <CanonicalName>cmdToggleShowShapeMaps</CanonicalName>
29                  <ButtonText>Toggle Shape Maps</ButtonText>
30                  <ToolTipText>Hide/Show shape map relationships (Unclutter your model)</ToolTipText>
31              </Strings>
32          </Button>
33      </Buttons>
34  </Commands>
35
36  <Symbols>
37      <GuidSymbol name="guidImages" value="{9D4B780D-34B2-428f-B132-D9F687F92842}" >
38          <IDSymbol name="toogleShapeMaps" value="1" />
39      </GuidSymbol>
40
41      <GuidSymbol name="cmdToggleShowShapeMapsGuid" value="{40B24DAA-0D1B-4243-BFD1-030CA5D680F6}">
42          <IDSymbol name="cmdToggleShowShapeMapsId" value="0x810" />
43      </GuidSymbol>
44  </Symbols>
```

Figure 146: Class Diagram Approach: VSCT code snippet

6.3.1.10 Method Automation

Automation was added to help the meta-model easily evolve as new classes were added to it by developers. After adding a few base classes (classes which contain generic methods) and interfaces we were ready to add specialized classes (class which cater for special cases) which could inherit from these base classes and interfaces. Therefore, each abstract or interface method (function/operation) had to be replicated in the specialized class. Manually copying these functions was possible for the first few specialized class although as more classes get added or even if we add/modify the functions within the base class, each change has to be manually propagate to its specialized classes. We implemented a mechanism which automates this synchronization process to allow the meta-model developer concentrate on the development of the meta-model.

Figure 147 below shows a part of the meta-model we developed as part of the class diagram approach. Notice the similarities between this figure and the final meta-model shown in Section 5.4. We will look at the details of this meta-model in the following sub-section.

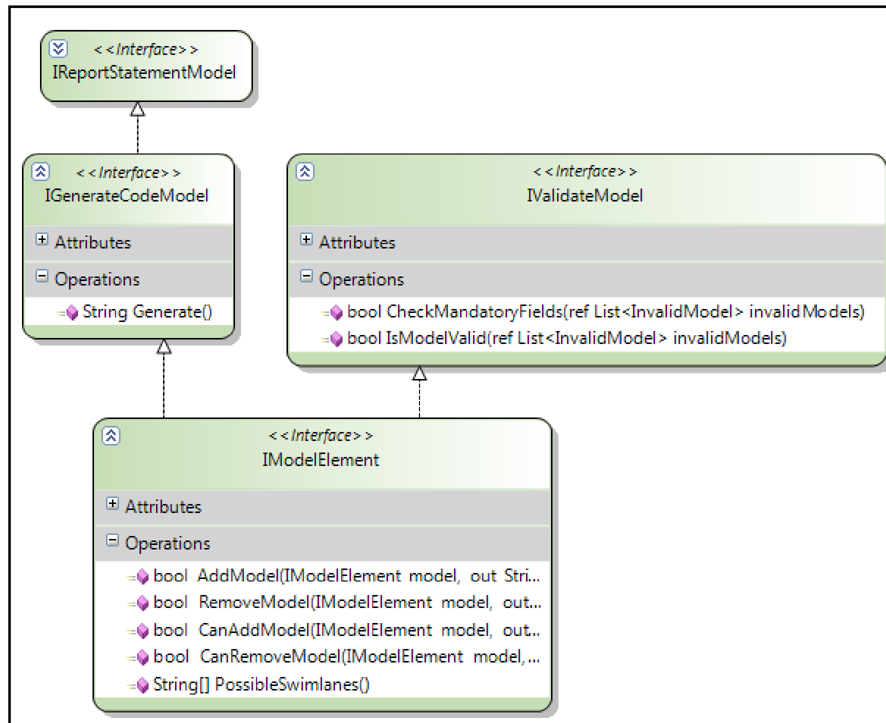


Figure 147: Class Diagram Approach: Method automation example

Now if we wanted to add a class which inherited from the *IModelElement* interface shown in the above figure we would have to manually copy all the methods from IGenerateCodeModel, IValidatedModel and IModelElement. This would be very time consuming without our automated helper, although after our helper was implemented, all the developer needed to do was add an inheritance relationship between the new class and the required interface (Figure 148) and all functions would be copied over (Figure 149). Note that all functions will be copied over recursively going up the hierarchy making this a very powerful tool. I made extensive use of this functionality while designing and developing the RWL meta-model using the Class Diagram Approach.

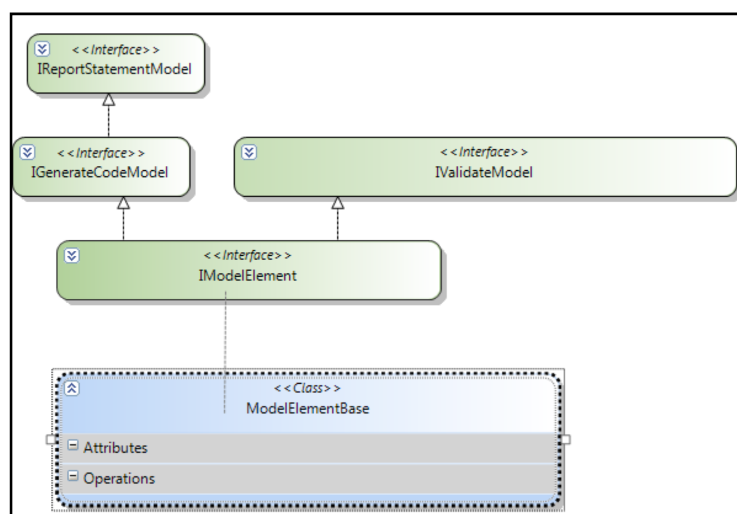


Figure 148: Class Diagram Approach: Method automation helper, step 1

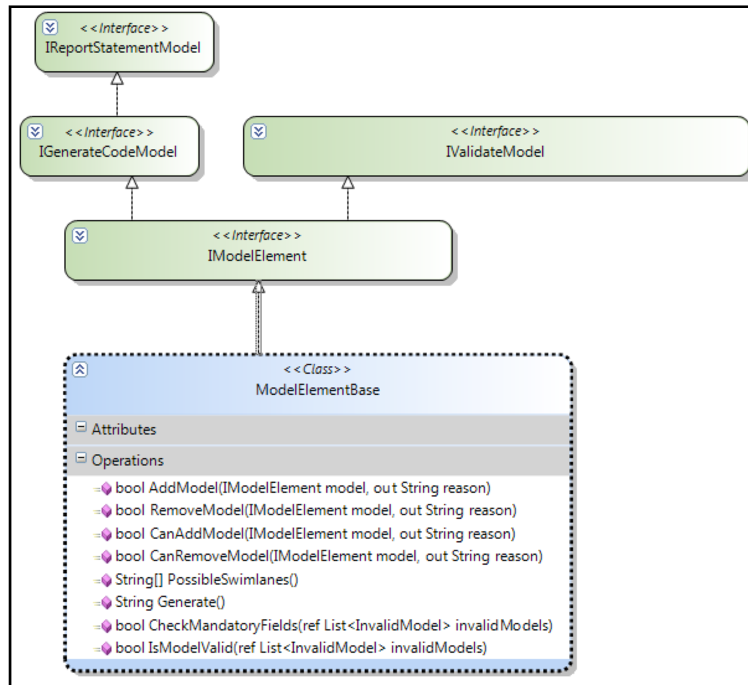


Figure 149: Class Diagram Approach: Method automation helper, step 2

Note that in Figure 148 and Figure 149 above we have collapsed our parent interfaces so that we have a small concise figure.

6.3.1.11 Explorer/Toolbox View

We implemented a simple tree-view explorer for the meta-model developer to easily allow them to see the meta-model they have created. This explorer was designed using icons matching those from the toolbox (Figure 150) which the developer used to create the meta-model elements, making it intuitive. A screenshot of the explorer is shown in Figure 151.

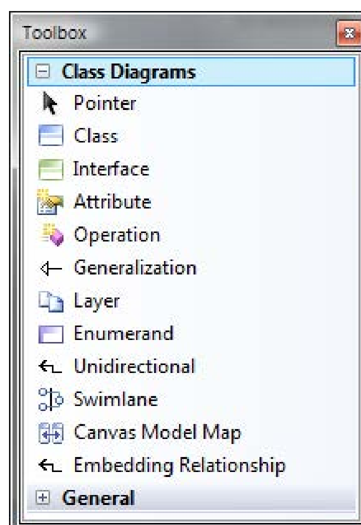


Figure 150: Class Diagram Approach: Meta-meta-model toolbox

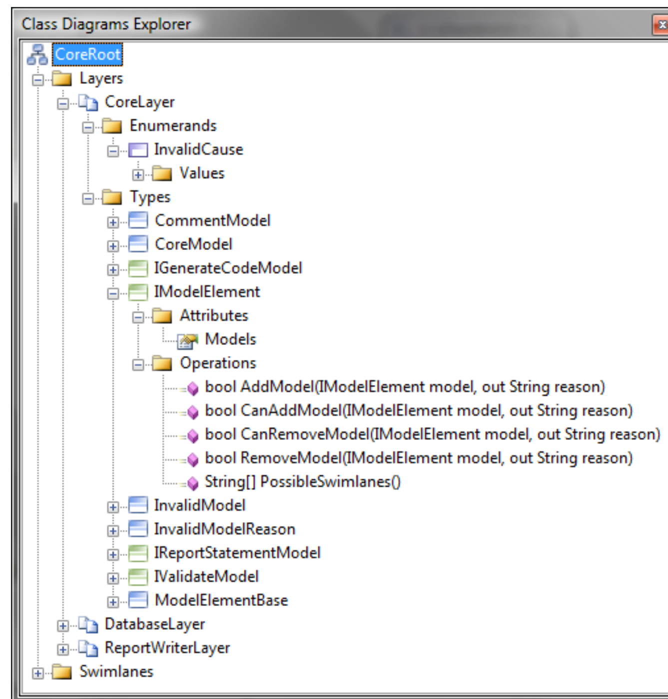


Figure 151: Class Diagram Approach: Meta-Model model explorer

6.3.2 Meta-Model Development

After developing our meta-meta-model we started implementing the meta-model and putting together a high level class diagram of the RWL constructs starting from the base. This approach was abandoned in the early stages of implementation therefore only a small portion of the RWL is modeled via this meta-model.

We divide this section up into two sub-sections. We look at how the meta-model was developed and the importance of the base classes and the hierarchy and then proceed onto showing how the code was generated using this meta-model.

6.3.2.1 Meta-Model

We started creating our meta-model by adding four core interfaces: *IReportStatementModel*, *IGenerateCodeModel*, *IValidateModel* and *IModelElement*. *IReportStatementModel* and *IGenerateCodeModel* were demonstrated in Section 5.4.

- *IReportStatementModel - Interface*
  A simple interface without any functions or attributes. All RWL constructs will inherit from this so that they have a common base.

- *IGenerateCodeModel - Interface*
  Contains a single function called *Generate* which returns a string. This function has to be implemented by any RWL class which can potentially generate RWL script. Gives us the capability of configuring the RWL script generation process.

  Also to make our inheritance hierarchy simpler, *IGenerateCodeModel* inherits from *IReportStatementModel*.

- *IValidateModel Interface*
  Contains two functions *IsModelValid* and *CheckMandatoryFields*. Any RWL construct which needs to provide validation has to inherit from this class and provide implementations for the two functions. It is intended that *IsModelValid* will call *CheckMandatoryFields* which is automatically implemented against the mandatory fields indicated by the developer.

- *IModelElement - Interface*
  Shown below in Figure 152.

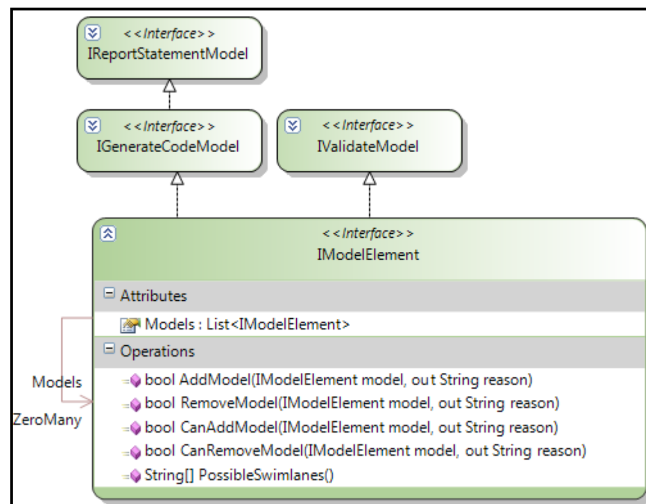


Figure 152: Class Diagram Approach: IModelElement interface

This interface brings together *IGenerateModel* and *IValidateModel* interface and also adds extra functions.

Models (an Attribute)
We anticipated that a given RWL construct could contain other RWL constructs, for e.g. a *Scan* construct could contain *Print* constructs, this relationship is mapped by this attribute. It represents all *IModelElement* objects contained by this *IModelElement*.

AddModel(...)/RemoveModel(...)
A function to add/remove one *IModelElement* object to another. It returns true or false indicating whether the addition/deletion was successful. It is intended that this function will call *CanAddModel(...)/CanRemoveModel(...)* to determine if an addition/subtraction is possible.

PossibleSwimlanes(...)
Returns a list of swimlanes which this *IModelElement* object can be added to.

- *ModelElementBase – Abstract class*
  Shown below in Figure 153.

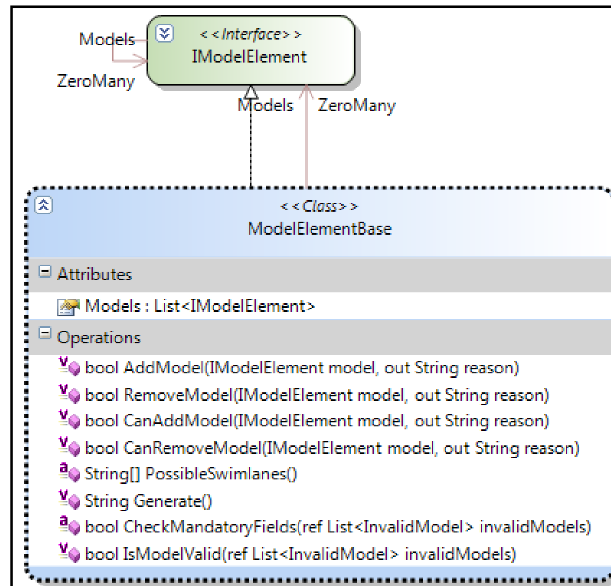


Figure 153: Class Diagram Approach: ModelElementBase abstract class

This class provides basic implementations to all the functions exposed by the *IModelElement* interface (and any interfaces which *IModelElement* inherits from and so on).

- *CoreModel - Class*
  Shown below in Figure 154. Inherits from *ModelElementBase* and represents the root model of the meta-model.

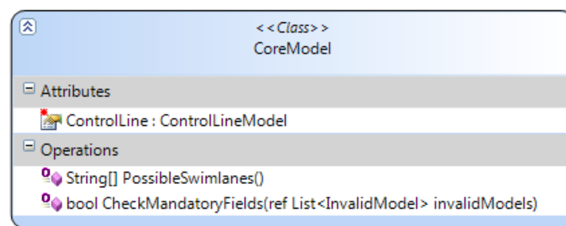


Figure 154: Class Diagram Approach: CoreModel class

Note in the above figure that the attribute *ControlLine* is marked as mandatory, this represents the RWL constraint that every RWL model should have one and only one *ControlLine*.

- *ControlLineModel - Class*
  Shown above in Figure 154. Inherits from *ModelElementBase* and represents the RWL *ControlLineModel* construct.

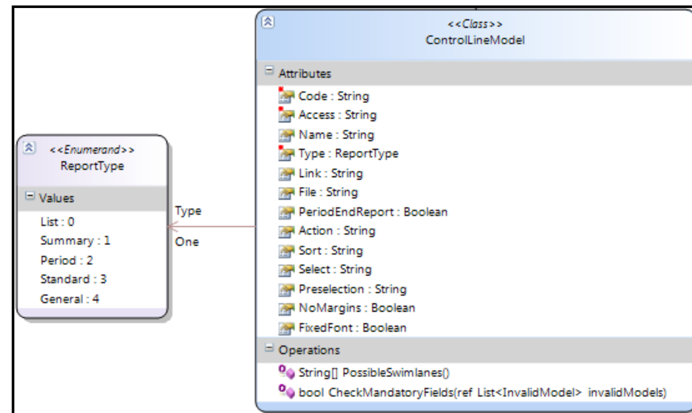


Figure 155: Class Diagram Approach: ControlLineModel class

Note in the above figure the three mandatory attributes of the ControlLine construct (*Code*, *Access* and *Type*). We have also shown an enumerand type in the above figure, the enumerand represents the possible values that the attribute *Type* can take.

- *Helper classes*
  Figure 156 shows the remainder of the classes in the meta-model. These classes help us pass validation information to the user interface.

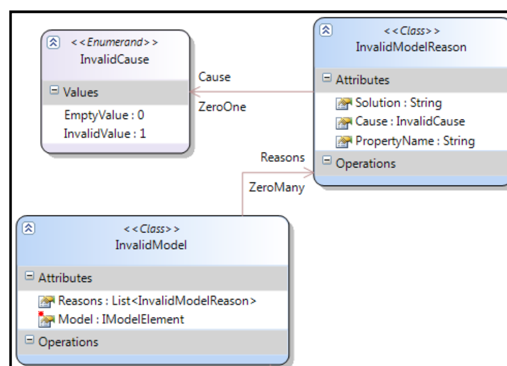


Figure 156: Class Diagram Approach: Added helper classes

The entire meta-model is shown below in Figure 157, note how the *ControlLineModel* is mapped to the *ReportHeader CanvasSwimlane* indicating that it can be only dropped in the header section of the report:

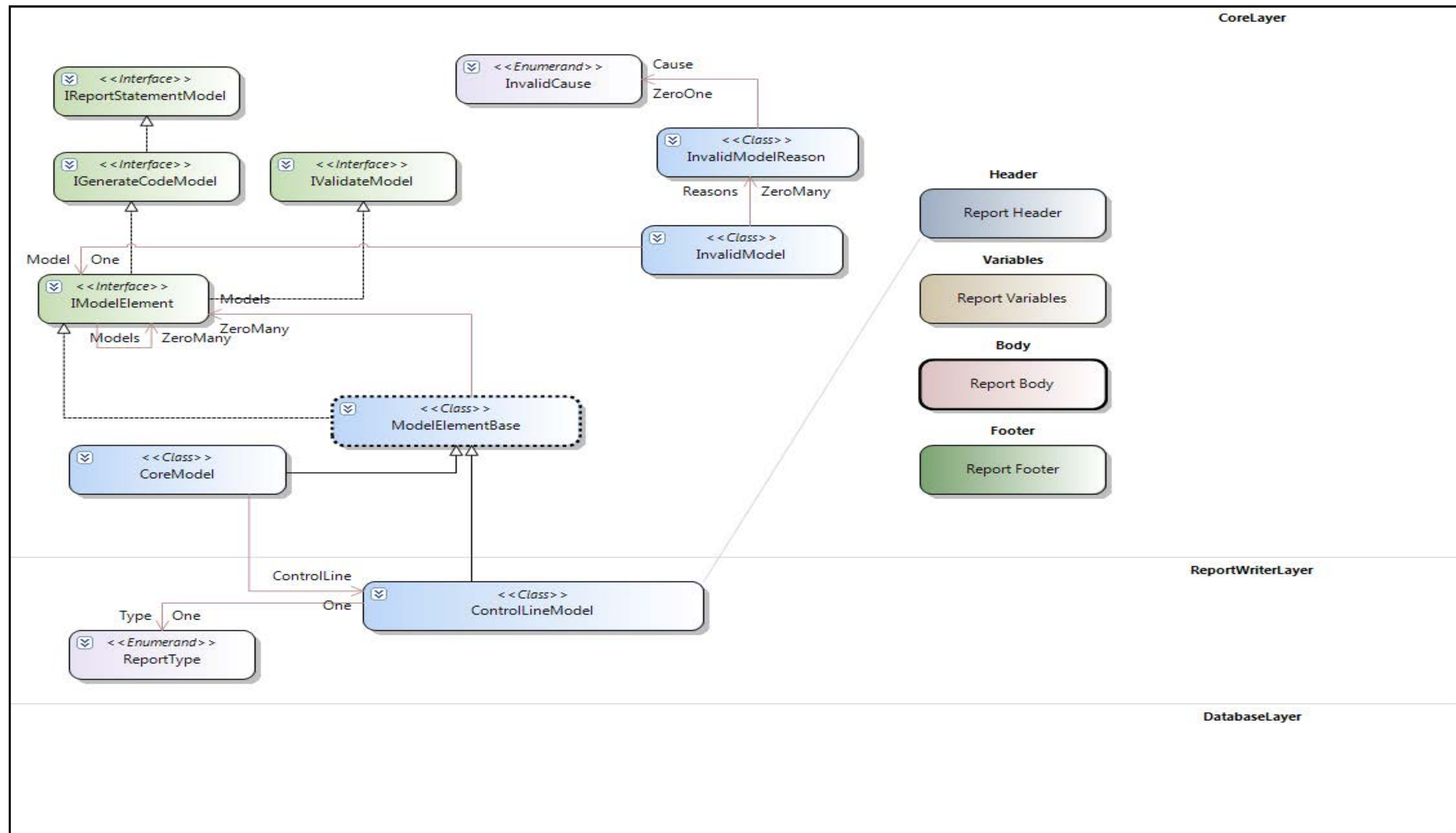


Figure 157: Class Diagram Approach: Meta-model

6.3.2.2 Code Generation

Table 3 below shows the mapping between the model element and what it gets converted to within the C# language and which physical file it resides in:

Table 3: Class Diagram Approach: Code generation mapping

Model Element	C# Language Construct	Physical File Name
ModelClass (incl. Abstract)	Class	Classes.cs
ModelInterface	Interface	Interfaces.cs
Enumerands	Enum	Enumerands.cs
CanvasSwimlanes	Class	Swimlanes.cs

Other files generated include:

- *DomainPropertyHandler.cs*
  Contains a generic event handler which allows event notification when attributes change their values.

- *ModelToolbox.cs*
  Exposes a given instantiated *ModelClass* object to the UI by adding it to this class. At runtime, the UI toolbox reads this information and populates itself accordingly.

- *Attributes.cs*
  Contains .NET CLR attributes which we can set on attributes belonging to *ModelClass* elements allowing them to open custom editors if required.

We generated our code from the model using the T4 Text Templating Engine (outlined in Section 5.7.2.3) in a sequential fashion and the flow is shown below in Figure 158:

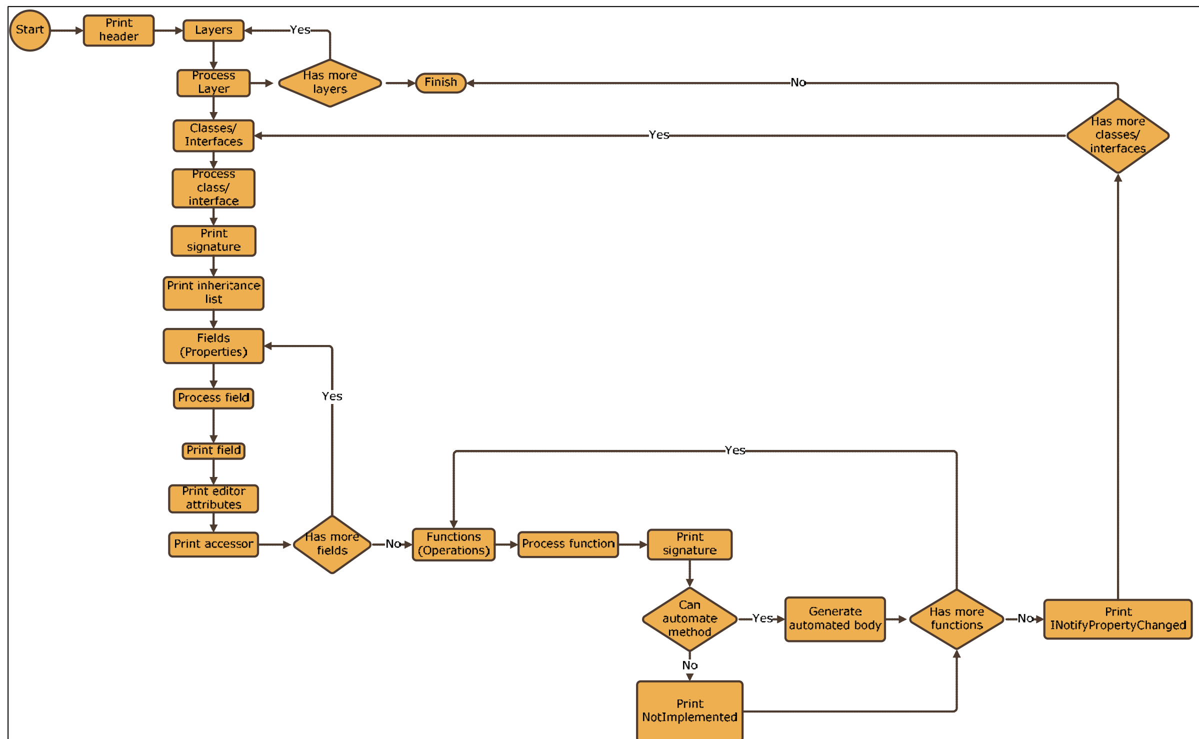


Figure 158: Class Diagram Approach: Meta-model code generation

The following points are based on the above figure:

- *Print Header*
  Header information includes items like import statements and information about the generation process.

- *Layer processing*
  We see in Figure 157 that we have three layers represented as horizontal swimlanes. We process each layer separately so that each class which is embedded within that layer resides in a common namespace.

- *Class/Interface processing*
  Each class or interface is processed by initially printing its signature. The signature of a class or interface includes its accessibility, name and its abstract nature. For e.g. `public abstract partial class ModelElementBase` indicates that the class *ModelElementBase* (class name) can be accessed by any other class (public) and is abstract. Note this matches the definition of the class provided by the meta-model which we examined in Section 6.3.2.1.

  After the signature the inheritance list of a particular class or interface is printed. This indicates the implementation chain specific to this class. For e.g. `public abstract partial class ModelElementBase : INotifyPropertyChanged, IModelElement` indicates that the class *ModelElementBase* inherits from *INotifyPropertyChanged* and *IModelElement*. Note that inheriting from *IModelElement* matches the meta-model in Section 6.3.2.1 although inheriting from the *INotifyPropertyChanged* interface is indicated by a property (Figure 159)

on a given class. If set to *true* the interface name is generated. This interface allows us to trigger events which notify the UI when a particular property is modified.

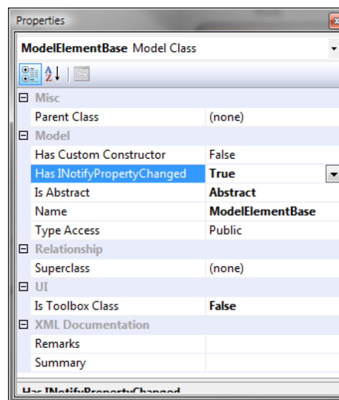


Figure 159: Class Diagram Approach: INotifyPropertyChanged attribute on a ModelClass object

- ***Field (properties) processing***
  Embedded within a class or an interface are its fields (properties). We process each field by printing the actual field (e.g. `private List<IModelElement> _models;`) and then printing its accessor (e.g. `public List<IModelElement> Models { get { return _models; } set { _models = value; } }`). Note that if we are printing a field within an interface we only print its accessor (e.g. `List<IModelElement> Models { get; set; }`).

  Just prior to printing the accessor for a given field we print the editor attribute for that field (e.g. `[EditorAttribute(Editor=EditorType.DefaultEditor)]`). This editor attribute indicates if the user modifies this particular field using a custom editor. This is extracted from the *EditorType* property shown in Figure 140.

- ***Function (operation) processing***
  We process functions (operations) after we finish processing embedding fields within a class or an interface. We start by printing the signature of the operation which includes its accessibility, name, abstract nature and its parameters (e.g. `public abstract bool CheckMandatoryFields(ref List<InvalidModel> invalidModels)`).

  If an operation belongs to an interface then this is all that is required although for a class we further check if the method signature matches the signature of an automated method. If it does an implementation is generated else the body is marked as not implemented.

  An automated method is a method which we generate the implementation for automatically. *CheckMandatoryFields* is an example of such an automated method. Its implementation solely depends on the mandatory fields marked by the meta-model designer. For e.g. in Figure 155 we see that the *ControlLine* object has three mandatory fields, the code generated by our automated generator is shown below in Figure 160.

```
public override bool CheckMandatoryFields(ref List<InvalidModel> invalidModels)
{
    bool valid = true;
    List<InvalidModelReason> reasons = new List<InvalidModelReason>();
    if (String.IsNullOrEmpty(_code))
    {
        reasons.Add(new InvalidModelReason() { PropertyName = "Code", Cause = InvalidCause.EmptyValue });
        valid = false;
    }

    if (String.IsNullOrEmpty(_access))
    {
        reasons.Add(new InvalidModelReason() { PropertyName = "Access", Cause = InvalidCause.EmptyValue });
        valid = false;
    }

    if (String.IsNullOrEmpty(_name))
    {
        reasons.Add(new InvalidModelReason() { PropertyName = "Name", Cause = InvalidCause.EmptyValue });
        valid = false;
    }

    if (_type == null)
    {
        reasons.Add(new InvalidModelReason() { PropertyName = "Type", Cause = InvalidCause.EmptyValue });
        valid = false;
    }

    invalidModels.Add(new InvalidModel() { Model = this, Reasons = reasons });
    return valid;
}
```

Figure 160: Class Diagram Approach: Automated method generator for CheckMandatoryFields

- *INotifyPropertyChanged*
  The importance of the *INotifyPropertyChanged* interface was outlined earlier. The last part of processing a class includes printing an implementation for this interface if one is needed. This basically means adding an extra function which properties can call to indicate they have changed.

We have seen how the code gets generated for a class or an interface, code for enumerands and other objects is done in a similar fashion. So far we have explained the implementation of Step 2 of Figure 79 (note that Step 1 is an automatic generation process done via the built-in templates provided by Microsoft DSL Tools which were shown in Figure 74).

Step 3 of Figure 79 involves the generation of the RWL script from the RWL model created by the end-user. This was not implemented to a great detail although the idea was that each RWL model element would implement the *IGenerateModel* interface described in Section 6.3.2.1. With this inheritance structure we would simply call the *Generate(...)* function of that particular RWL model element to give us the corresponding RWL script.

6.3.3 Shell Host Development

6.3.3.1 Detailed Architecture of User Interface (WPF UI)

The shell host in our Class Diagram Approach is the WPF UI. The UI allows the end-user to instantiate meta-model objects and put them together to model RWL reports. This sub-section highlights how this UI was implemented. Figure 161 below shows a high level architecture of the application.

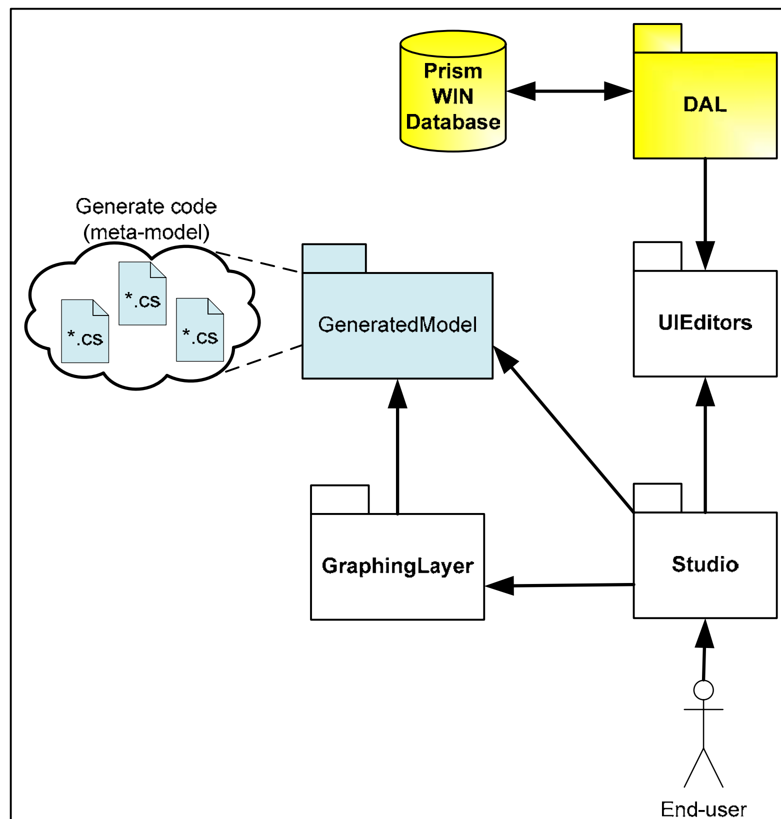


Figure 161: Class Diagram Approach: WPF UI architecture

The DAL layer shown in the above figure is written using LINQ and allows us to display Prism meta-data information to the end-user when required. The DAL layer is instantiated by the UIEditors layer which creates custom field editors for fields marked with the custom *EditorType* attribute flag.

The fulcrum of the WPF UI is the Studio layer which the end-user interacts with. This layer provides the front end application which then in turn interacts with three other layers: The UIEditors layer (described in the preceding paragraph), the GraphingLayer and the GeneratedModel.

The GraphingLayer enables user to draw shapes, connect them thus allowing them to interact with the meta-model of the RWL. The GraphingLayer therefore is the only point where the GeneratedModel is used. Although the GraphingLayer does pass back validation information to the Studio therefore it is necessary that the Studio has some notion of the GeneratedModel.

6.3.3.2 The User Interface

This sub-section looks at the implementation details of the WPF UI. We divide the UI up into smaller sections for further investigation.

- *The Toolbox*
  The toolbox holds all the RWL constructs exposed by our meta-model. It is populated by reading the contents of *ModelToolbox.cs* (introduced in Section 6.3.2.2). The file structure is shown below in Figure 162.

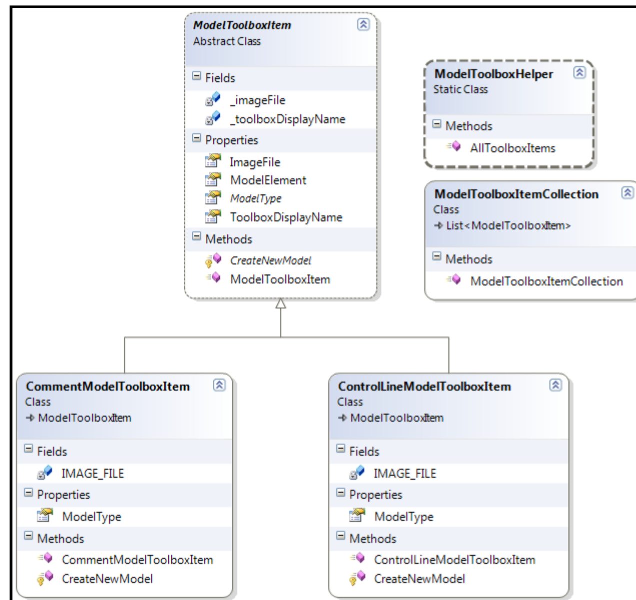


Figure 162: Class Diagram Approach: UI toolbox population

Examining the above figure we can see that we have a helper class which is called by the UI to give us all the toolbox items. As this approach was abandoned in its early stages we only had two toolbox items: *CommentModelToolboxItem* and *ControlLineModelToolboxItem*. All toolbox items were generated by inheriting from a base class called *ModelToolboxItem*. This class contained fields which gave the UI information about the image to show (*ImageFile*), the model element which it was mapping (*ModelElement*), the type of model element (*ModelType*) and the displayed name of the model element (*ToolboxDisplayName*).

This allows our UI toolbox to be populated as shown below in Figure 163.

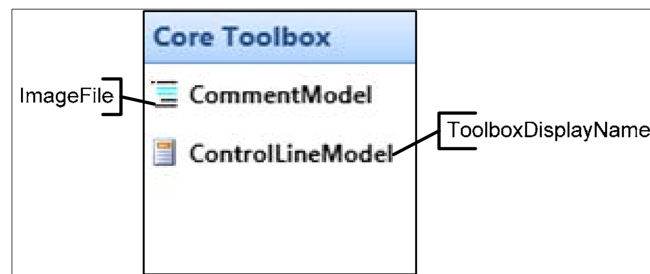


Figure 163: Class Diagram Approach: UI Toolbox

When an item from the toolbox is dragged onto the drawing canvas, we grab the mapping meta-model element by calling its *ModelElement* field which in turn instantiates an object by calling the *CreateNewModel* function. This is how a meta-model element is instantiated by the end-user.

- *Toolbar/menu*
  A modern toolbar/menu structure was implemented as part of the WPF UI. This is shown in Figure 107. The end-user could easily interact with the toolbar to create a new RWL model, save or open an existing RWL model, generate the RWL script along with other common functionality such as copy/paste and undo/redo. The toolbar also provided help in the form

of tooltips which informed the end-user of their functionality. An example of a tooltip is shown in Figure 109.

- *Docked windows*
  We provided docked window support to give end-users maximum viewing area of the RWL model they were designing. Docked windows enable end-users to hide and show windows when required leaving them to concentrate on the task at hand. Figure 108

- *Canvas/Shapes*
  The canvas is the drawing surface provided by the WPF UI on which model elements are plotted. These model elements are dragged from the toolbox and dropped onto the canvas. Each model element which is dropped on the canvas has a shape mapped to it. As this approach was discarded at an early stage we only have one model element and a matching shape (*ControlLineModel*). In Figure 164 below, on the left side we see the model element during the drag operation and on the right we see the model element on the canvas after it is dropped.

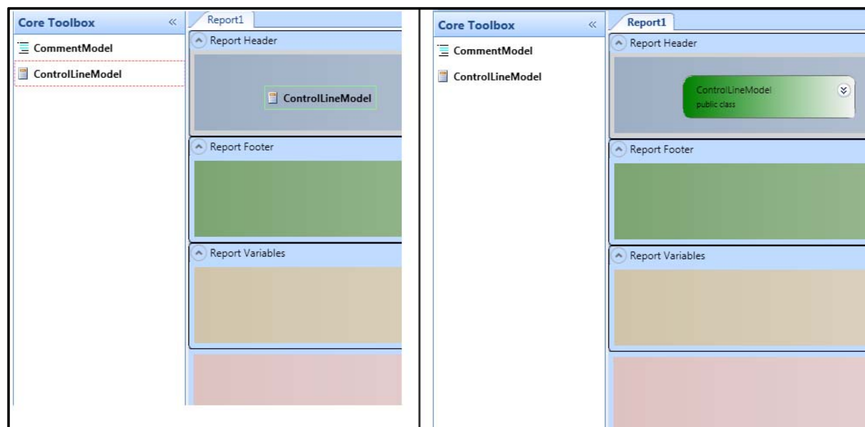


Figure 164: Class Diagram Approach: Adding a model element to canvas

During drag and before a drop is made, the canvas executes background validation checks using of the RWL meta-model. These are further explained in the following paragraph.

- *Error checking/validation*
  Currently our meta-model only caters for checking whether mandatory fields are populated (Section 6.3.1.6) and whether dropping a specific model element on a report section is allowed (Section 6.3.1.4).

  When a model element from the toolbox is being dragged by the end-user on top of a report section it will have a green border if a drop is possible and a red border if a drop is not possible. We see from the left hand side of Figure 164 that the *ControlLineModel* can be dropped on the *ReportHeader* swimlane as specified by the meta-model (Figure 157) therefore it has a green border around it. Although if the *ControlLineModel* is attempted to be dropped in a different report section we see a red border along with intuitive error information shown in Figure 165 below.

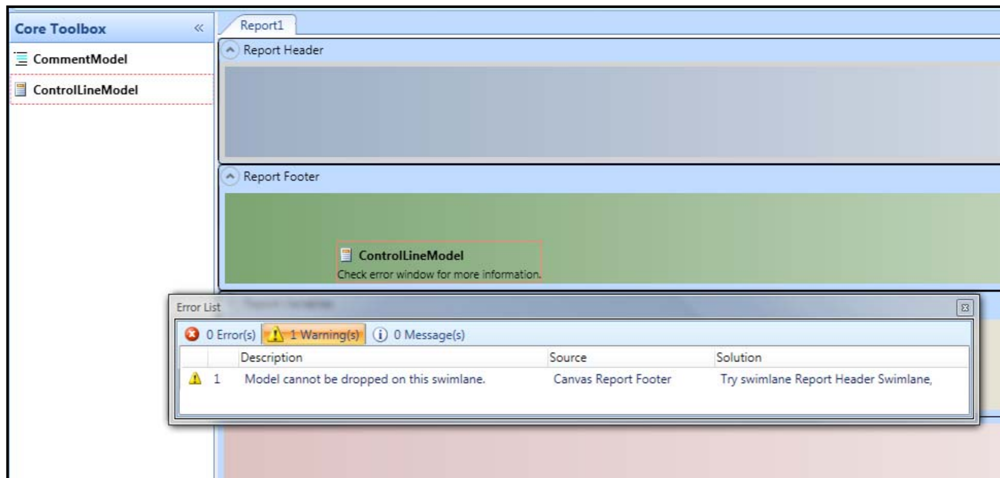


Figure 165: Class Diagram Approach: WPF UI validation check

After the *ControlLineModel* is successfully dropped on the canvas we can simply right click on it and request a validation (will validate soft constraints as explained in Section 3.3.1.3) as shown below in Figure 166.

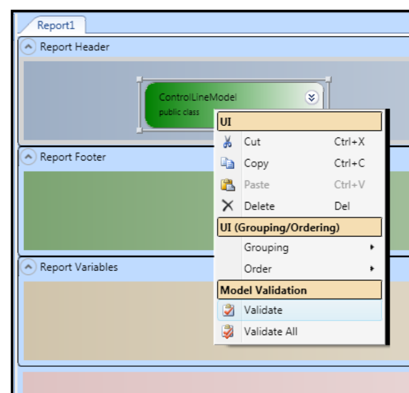


Figure 166: Class Diagram Approach: WPF UI requesting validation of soft constraints

This will trigger the *IsModelValid* function on the model element which currently only checks the mandatory fields by calling the *CheckMandatoryFields* function. The result from the above validation request is shown below in Figure 167. Note the description provided for each error below, these get populated in the *CheckMandatoryFields* function shown in Figure 160 with the help of our meta-model helper classes shown in Figure 156.

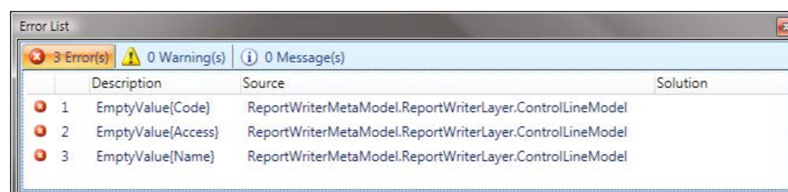


Figure 167: Class Diagram Approach: WPF UI validation results

- *Field editing*

  After a model element gets added to the canvas the next thing the report designer can do is edit its fields (properties). The UI offers this capability by exposing a property grid which

gets populated whenever a model element is selected with the mouse. The property grid for the *ControlLineModel* is shown below in Figure 168.

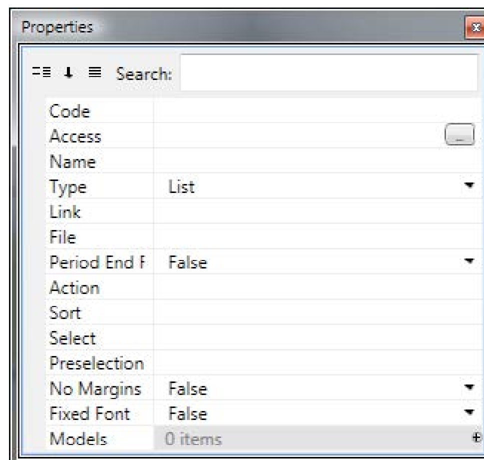


Figure 168: Class Diagram Approach: Property Editing

The UI is intelligent and recognizes custom editor attributes. In Section 4.2.1.4 we introduced an example which stated the field *Access* contained a list of Windows[2] via which this report can be executed. The field *Access* is thus marked with a custom editor tag (Section 5.7.2.5) in the meta-model. The UI respects this tag and opens the appropriate editor, in this case a *WindowSelectionEditor* to allow the end-user select which Windows[2] this report can be accessed from. Therefore, clicking on the "..." button which appears by the *Access* field in Figure 168 invokes the window shown below in Figure 169.

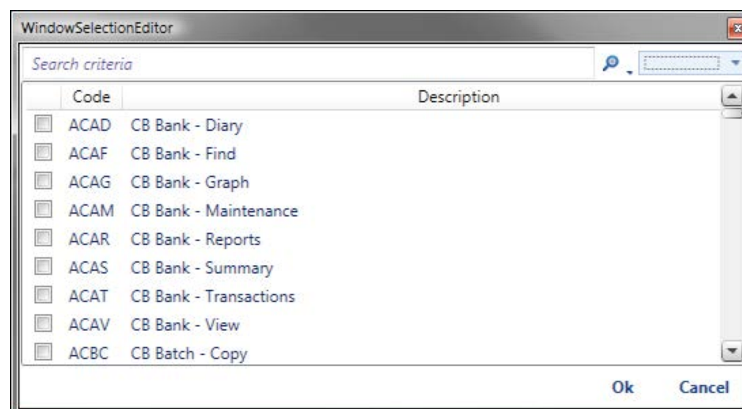


Figure 169: Class Diagram Approach: Custom editor

- <u>*RWL generation*</u>
  After editing properties the end-user can generate RWL script from this model. It was intended that the RWL script will be generated recursively by calling the *Generate* method on the *CoreModel* which calls the *Generate* method on its children. As this approach was discarded in its early stages, we only generated the RWL script from the *ControlLineModel*. The generated RWL script is shown in one of the docked windows titled "Generated Code", shown below is an example.

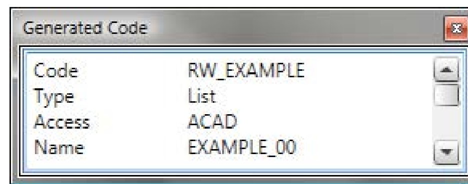


Figure 170: Class Diagram Approach: Generated RWL

6.4 RWM Shell Approach Implementation

The RWM Shell Approach was introduced in Section 3.4.2.2. This section looks further into the details of that approach and the how it was implemented. In this approach we implemented the meta-model of the RWL using the Microsoft DSL tool and then used the Visual Studio Shell to host it and allow users to create RWL models. We start of by showing how the RWL meta-model was implemented followed by showing a few examples of the RWL models created using this meta-model and the RWL script generation.

Reiterating on why we chose this approach was to reduce the complexity of designing shapes and connectors (use ones provided by Microsoft DSL Tools) as compared to with the Class Diagram Approach (where we had to design them from scratch). As we removed the WPF UI layer from this approach we basically removed an entire stage which allowed us to design a prototype of a reporting tool within the given time frame and allowed us to use the T4 templating engine to directly output RWL script (this approach) as opposed to outputting meta-model code (Class Diagram Approach).

The two main stages of implementation for the RWM Shell Approach are formed with respect to the overall high level approach defined in Section 3.4.2.2 (shown below).

1. Design the RWL meta-model structure using Microsoft DSL Tools
2. End-user designs the RWL model using the meta-model using the shell
3. Use text templates to generate the corresponding RWL script from the RWL model

The first point (1) in the above list is the meta-model development (Section 6.4.1). Point two and three form the shell host development phase (Section 6.4.2).

6.4.1 Meta-Model Development

As outlined in Section 5.8 we started our implementation from a minimal language project wizard provided by Visual Studio. This allowed us to start implementing the RWL constructs cleanly from the base.

6.4.1.1 Reporting sections (Swimlanes)

We wanted to provide a modular reporting interface to the user, an idea borrowed from Scribe (Section 2.6). This meant providing the user with disparate logical sections. The three initial sections which developed were header, body and variables. We also realized that these sections will be mandatory on every new report and that each RWL construct will be embedded in at least one of these sections. We added these model elements into our model, shown below in Figure 171.

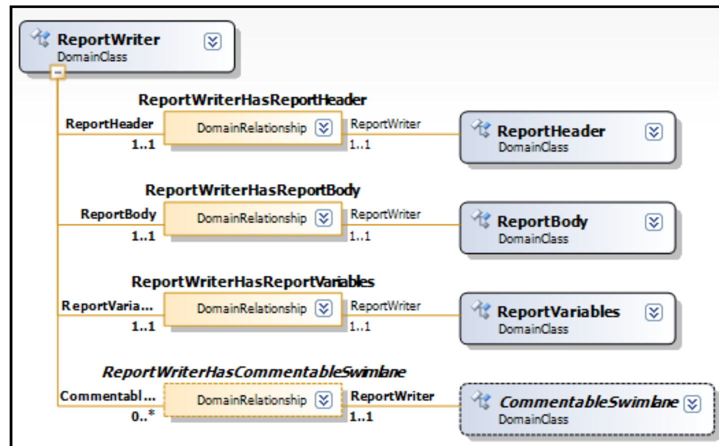


Figure 171: Shell Approach: Report sections in the RWL meta-model

Note in the above figure that the multiplicity on every report section is 1..1 which indicates that a report has to have one and only one instance of this model element. Also note the abstract model element *CommentableSwimlane*. Each of these sections will enforce certain constraints that will determine which RWL constructs can be added to them. We also realized that there is one RWL construct which can potentially be added to any section, this is the *Comment* construct. Therefore we implemented an inheritance tree for these sections which makes it easier for us to define which section can accept which RWL construct. The root of this inheritance tree is the *CommentableSwimlane* as shown below in Figure 172.

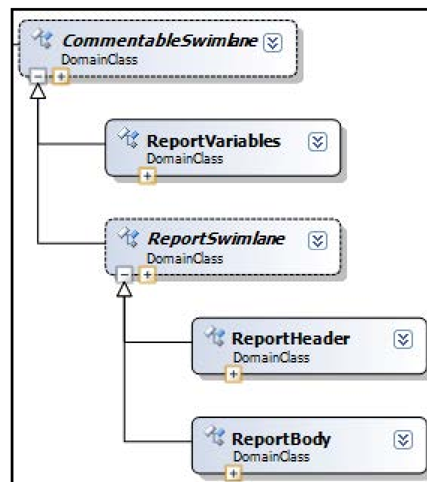


Figure 172: Shell Approach: Report sections hierarchy

Note in the figure above the model element *ReportSwimlane*. This works on a similar principle as that of *CommentableSwimlane*. There are common RWL constructs between the header and body section of the report, those constructs will be part of the *ReportSwimlane* as both, *ReportHeader* and *ReportBody* inherit from it. As for those constructs which are not common, they will be part of either one of the swimlanes *ReportHeader* or *ReportBody*.

6.4.1.2 Common Hierarchy

We wanted to build a hierarchy via which adding new RWL constructs within our meta-model is made easier. We borrowed the *NamedElement* concept from our class diagram approach although we designed our *NamedElement* with three properties as shown below in Figure 173.

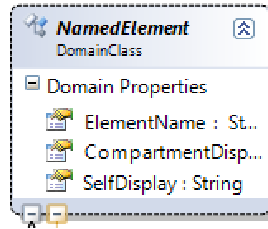


Figure 173: RWM Shell Approach: Named element

ElementName field (property) represents an internally represented name property. We made this a hidden property as it has no relevance to the end-user while designing the RWL model. The *ComparentDisplay* field is added to display information about this model element when it is contained within a parent model element and the *SelfDisplay* field displays information about this model element when it is display independently.

After adding the *NamedElement* we implemented our hierarchical model element structure. We showed the implementation of the reporting sections in the previous point so the obvious next step was to add model elements which can be added to these report sections, therefore we added three core model elements: *VariablesItem*, *HeaderItem* and *BodyItem*. Each model element is named in an intuitive manner. We can easily see that all *VariablesItem* model elements will be part of the *ReportVariables* report section and so on for *HeaderItem* and *BodyItem*. The implemented hierarchy is shown below in Figure 174.

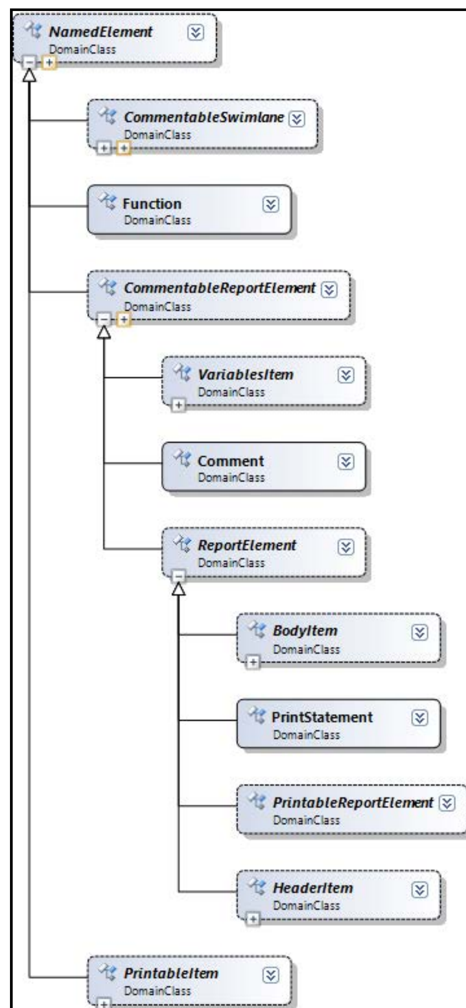


Figure 174: RWM Shell Approach: Common hierarchy

We have already looked at the *CommentableSwimlane* model element, the next two model elements we examine are the *Function* model element and the *PrintableItem* model element. The *Function* model element represents a RWL function and the *PrintableItem* represents any construct that can be attached to a *Print* construct (e.g. Literals). These inherit from the *NamedElement* model element only so that they can inherit its properties. The reason why *Function* and *PrintableItem* do not inherit from *CommentableReportElement* is examined in the next sub-section when we look at our program flow implementation.

The *CommentableReportElement* represents all model elements that can be added to a *CommentableSwimlane*. We then make a distinction between model elements and their specific report sections, therefore we have: *VariablesItem*, *BodyItem* and *HeaderItem*. Any model element that can be added to the variables section of the RWL model can inherit from VariablesItem and so on for the other sections of the RWL.

Note that the *Comment* model element directly inherits from *CommentableReportElement* as a comment can be added to any report section. This principle is also applied to the *PrintStatement* and *PrintableReportElement* which inherit from *ReportElement* which is the common parent for any model element that can be added to both, the header and the body report sections.

6.4.1.3 Program flow support

We implemented flow support by simply allowing the end-user to create a reference relationship from one model element to the next model element. This relationship was represented in our meta-model as *ElementReferencesNextElement*, also shown below in Figure 175.

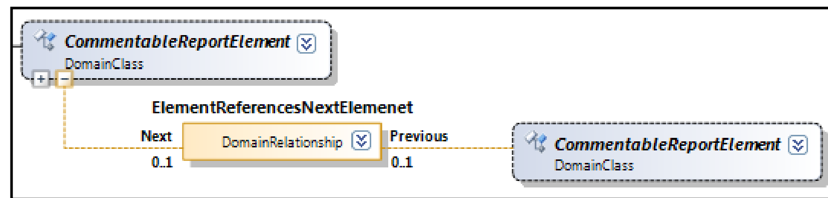


Figure 175: RWM Shell Approach: Program flow meta-model

The *ElementReferencesNextElement* relationship is formed between two *CommentableReportElement model elements*, therefore any model element which inherits from a *CommentableReportElement* can participate in such a relationship. Note the multiplicity on each role player, it is 0..1, indicating that a given *CommentableReportElement* can have at most one next and at most one previous element.

Referring back to Figure 174 we saw that the *Function* and *PrintableItem* model element do not inherit from *CommentableReportElement*. The reason behind this is that these model elements cannot participate in the *ElementReferencesNextElement* relationship.

6.4.1.4 Custom Field editors

As illustrated in Section 5.8.2.4, the DSL Tools allow us to annotate fields with special attributes. We can specify custom field editors using this mechanism, shown in Figure 93. We implemented four custom field editors: *WindowSelectionEditor*, *FileSelectionEditor*, *ColumnSelectionEditor* and *FunctionSelectionEditor*.

WindowSelectionEditor allows the end-user to select a list of Windows[2] via which a given report script can be accessed from.

FileSelectionEditor allows the end-user to select views (tables) from the Prism WIN database, used while creating *Scan* constructs.

ColumnSelectionEditor allows the end-user to select columns from a given Prism WIN database view, used to print information from a column. This editor is intelligent as it knows exactly which columns to show. For e.g. if we are printing information within a *Scan* on view *RM* the *ColumnSelectionEditor* will show only columns which reside in the *RM* view.

The *FunctionSelectionEditor* allows the end-user to select from a list of valid functions to execute on various RWL constructs. The *FunctionSelectionEditor* works in a similar fashion to the *ColumnSelectionEditor* as it only display functions which are valid for a given RWL construct. The RWL has constraints that specify which functions can be executed on which RWL constructs, for e.g. the function *Year* can only be executed on variables of type *ERA*.

Let us have a look at the implementation of the *ColumnSelectionEditor* at is an example of an intelligent editor so it gives information on how to implement a standard editor and also on how to

implement a context sensitive (intelligent) editor. Figure 176 below shows the *Column* model element and how it fits in with our *Scan* model element.

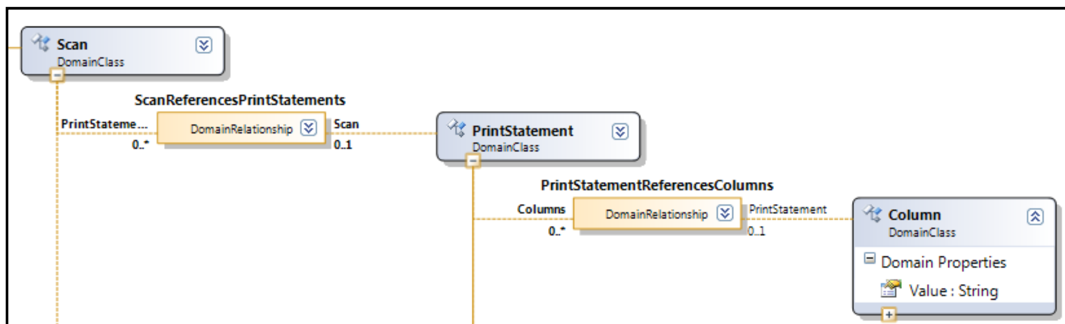


Figure 176: RWM Shell Approach: Scan-Print-Column relationship

The above figure is best explained with the following RWL script snippet:

```
1.   Scan RM
2.        Print RM_NAME;
3.   End
```

The code snippet shows the three RWL constructs which take part in the relationships shown in Figure 176. We can see that a *Scan* construct can have a *Print* construct within it and that *Print* construct can contain a specific *Column* construct. This is modeled in our meta-model as two relationships: *ScanReferencesPrintStatements* and *PrintStatementReferencesColumns*. These relationships allow us to design an intelligent *ColumnSelectionEditor*.

We start by adding an *EditorType* attribute to the *Value* field of the *Column* model element seen in Figure 176. Figure 177 below shows this attribute, it is similar to Figure 93 although notice the different editor type.

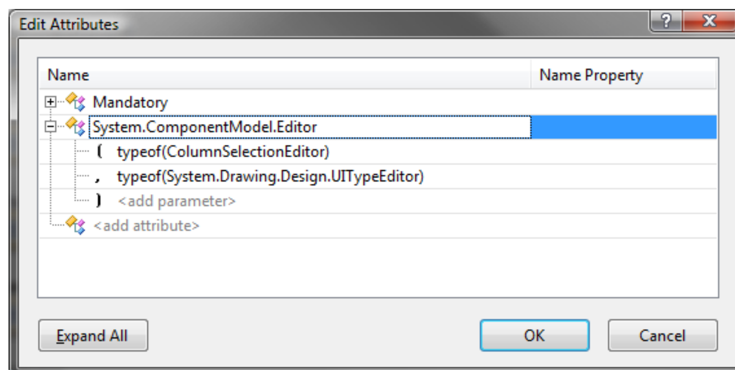


Figure 177: RWM Shell Approach: ColumnSelectionEditor custom editor

After adding the above attribute we can now start implementing our *ColumnSeclectionEditor*. Figure 178 below shows what the end-user sees while designing the RWL code snippet shown above. Notice that the figure below shows an instantiation of the model-elements and relationships shown in Figure 176.

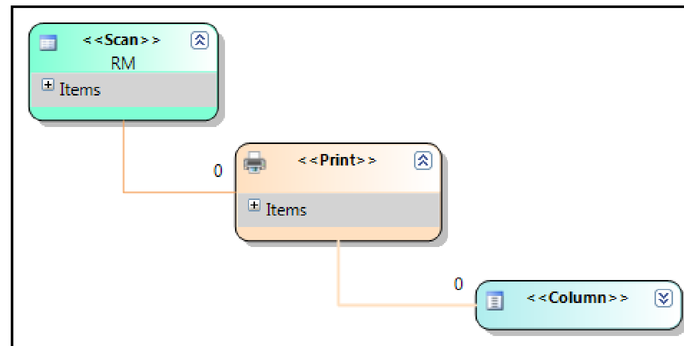


Figure 178: RWM Shell Approach: Scan-Print-Column instantiated

We now have a *Scan*, a *Print* statement within that scan to which the end-user has just added a *Column* model element. The next step for the end-user is to select the actual column which they want to print; this is where the custom editor comes into play. When the end-user tries to edit the *Value* field on the *Column* model element, the custom editor is opened (this is because we added the editor type attribute shown in Figure 177). We implemented the custom editor so we can traverse back via the relationships shown in Figure 176 all the way back to the *Scan* construct. This allows us to find out the view on which the *Scan* is being executed on, in this case the *RM* view, allowing the custom editor to only show columns which belong to the RM view. The resulting *ColumnSelectionEditor* for Figure 178 is shown below in Figure 179. Note that only columns of type RM_* are shown.

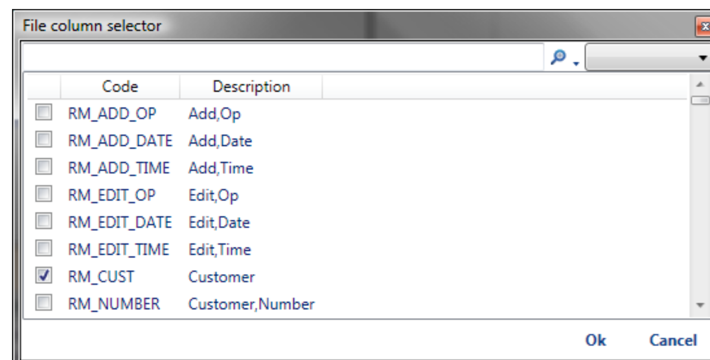


Figure 179: RWM Shell Approach: ColumnSelectionEditor for view RM

6.4.1.5 Mandatory Fields

The mechanism to cater for mandatory fields was done using attributes. Any field which requires a user input is marked with the *Mandatory* attribute; this is shown in Figure 93 and Figure 177. After a field within a model element is marked with this attribute we simply check for whether it has a value, if not, a validation error is thrown. This validation process is made easier due to our common hierarchy. As all model elements inherit from the *NamedElement* model element, we simply add our implementation code on that class because validating any model element will validate the *NamedElement* as it is the base of our hierarchy.

6.4.1.6 Containment, Hiding/Showing Children and Ordering

Section 5.8.3.4 and Section 5.8.3.6 gave an introduction on the design aspect on how we implemented containment and hiding/showing child elements respectively. This sub-section gives an insight on how we implemented it via our meta-model.

We implemented containment and the hiding/showing of children elements by using the compartment shapes provided by the DSL Tools. This concept is best explained with the help of a RWL script snippet shown below.

```
1.   Scan RM
2.       Print RM_CUST + RM_NAME;
3.   End
```

The *Scan* construct shown in the above code snippet contains a *Print* construct which prints out two *Columns* in a specific order.

We implemented containment and hiding/showing of child elements by simply mapping the *Scan* statement to a compartment shape (more on shape mapping in Section 6.4.1.8) and allowing it to display its children within a compartment (this was done for any model element that can contain other model elements, e.g. the *Print* model element). The result is shown below in Figure 180.

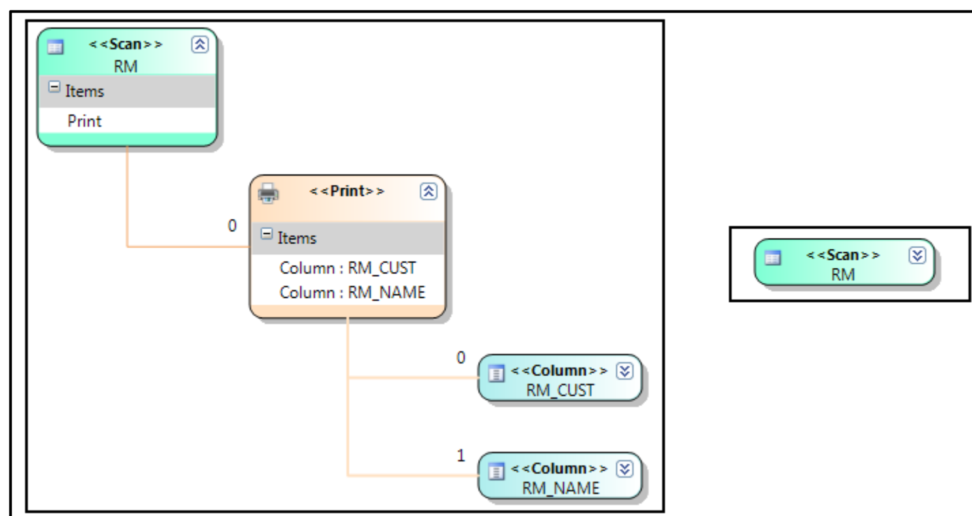


Figure 180: RWM Shell Approach: Implementation of containment/ordering and hiding of child elements

The *Scan* model element on the left side in the above figure shows its children embedded within a compartment (so does the *Print* model element). On the right hand side of above figure we see the same *Scan* model element but this time it is collapsed, neatly hiding its children (and its indirect children).

Note that the Scan-Print-Column relationship was shown in Figure 176, what was not shown were the properties of this relationship. Because any relationship which has a nesting aspect to it can have an ordering aspect, we made a base relationship which other relationships can inherit from and this base relationship has a property called *Order*, signifying the sequential placing of that particular relationship. This is shown below in Figure 181.

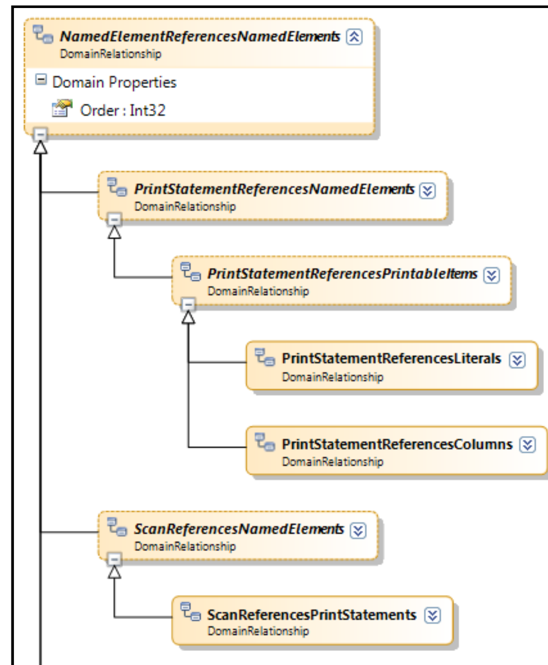


Figure 181: RWM Shell Approach: Relationship ordering

Note that relationship hierarchies are similar to model element hierarchies and in the above figure we have a hierarchy chain with the base relationship having the property *Order* meaning that the relationship we are concerned with, in this case *ScanReferencesPrintStatements* and *PrintStatementReferencesColumns* also has that property. This property is utilized in two places: showing containment and laying out child elements neatly in an ordered fashion. Both of these are shown in Figure 180, we can see that the order in which items appear in a compartment are significant and it maps the order of elements as they appear in the RWL script.

A recursive auto layout algorithm was implemented to modify the layout of child elements (and children's child elements and so on) when needed. This algorithm is triggered when:

1. *Order* property was changed
2. New child element was added
3. Explicit request was made by end-user via a context menu (Section 6.4.1.10)

The pseudo code for the algorithm is outlined below:

```
LayoutModelElement(ModelElement element)
{
  sortedChildren = Sort children via Order property;
  for each (ModelElement child in sortedChildren)
  {
     // layout code

     LayoutModelElement(child); // recursive call
  }
}
```

6.4.1.7 RWL Constraints

There are several levels of constraints implemented by our meta-model. Most constraints are enforced directly by our meta-model which are categorized under model constraints and others are more subtle constraints specific to our RWL visual language.

- <u>*Model constraints*</u>
  Hard constraints such as embedding model elements within report sections, reference relationships between model elements and multiplicity of these model elements and relationships are represented using the meta-model. Therefore, these constraints are naturally enforced for us and require no further code.

  For e.g. in Figure 171 we see that in any given RWL model we will need one *ReportHeader*, *ReportBody* and *ReportVariables* model element (represented as swimlanes).

  Other examples of model constraints can be seen in the figure below.

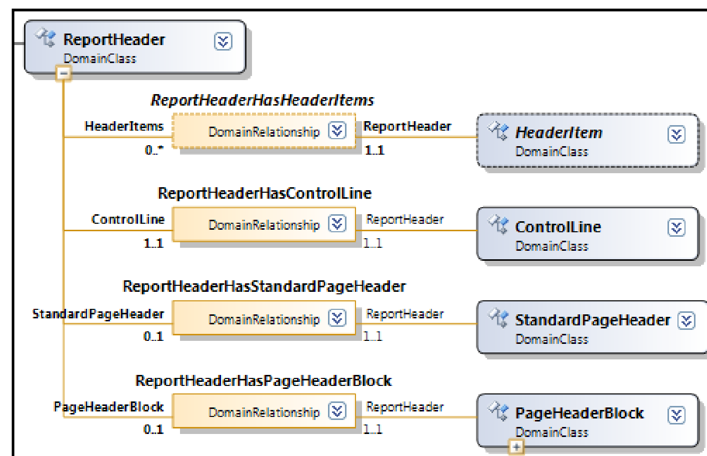


Figure 182: RWM Shell Approach: Model constraints

In the above figure we see the constraints surrounding the header section of a report. We introduced the *HeaderItem* in Figure 174 and described it as the base element for any model element that needs to be part of the *ReportHeader* section. We see this base relationship in the above figure, *ReportHeaderHasHeaderItems*. Note the multiplicity on this base relationship, indicating that we can have 0..* header items on our *ReportHeader*. If we only had this relationship we could do things like add two ControlLine model elements to our header, which would be wrong thus we further constrain this base relationship by more specialized relationships involving specialized *HeaderItem* model elements (model elements that inherit from *HeaderItem*). For e.g. in the above we have a specialized relationship *ReportHeaderHasControlLine* with multiplicity 1..1 signifying that a given *ReportHeader* section should have one and only one *ControlLine* construct thus respecting the RWL specification.

- <u>*Flow constraints*</u>
  Flow constraints are also enforced by our meta-model directly as shown in Figure 175. Although note that this constraint has a multiplicity of 0..1 on both roles, previous and next, signifying that an element may have a previous and a next reference although this leads to

"lost" elements. *Lost* elements are those elements which do not have a previous reference element therefore being unreachable and not fitting into the sequential flow of the RWL which is visibly wrong.

Even if we changed the multiplicity of both role players (next and previous) to be 1..1 we would have experienced with the first and last element of the RWL script as they would not have a previous and next reference respectively. We would also have problems with nested children. Let us consider the RWL script snippet below:

```
1.  ...
2.  ...
3.
4.  Print "Hello";
5.
6.  Scan RM
7.      Print RM_CUST + RM_NAME;
8.  End
9.  ...
10. ...
```

In the above script snippet we see that the flow moves on from the *Print* model element onto the *Scan* model element which also has a nested *Print* model element. Note that the nested *Print* model element is referenced by the *Scan* as opposed to being *next* in the flow. Therefore, according to our meta-model if we had a multiplicity of 1..1, this would result in an error as the nested *Print* statement does not have a *next* nor a *previous*.

Therefore we implemented a simple code based solution to the problem of *lost* elements. We added a validation trigger on the *CommentableReportElement* within our meta-model which gets triggered whenever a save is requested by the end-user. This trigger checks the *previous* property of element is *null* and also checks for special cases. For e.g. from our above RWL script snippet, if a *Print* model element has an associated *Scan* then it is a special case and does not require a *previous* element. The validation will fail if the element has no previous reference and is not a special case and will inform the user that the particular model element is unreachable.

- <u>*Multiple reference constraints*</u>
  Figure 183 below shows us the relationships that a given *Print* construct can participate in.

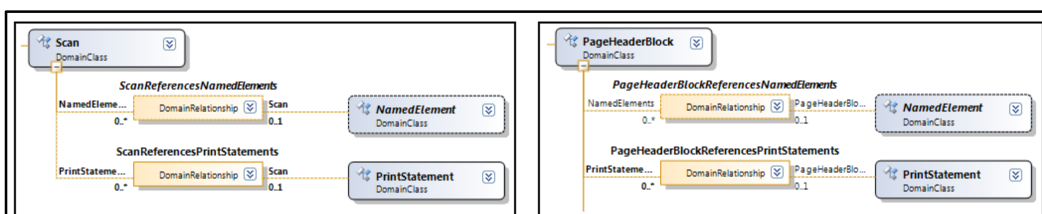


Figure 183: RWM Shell Approach: Multiple reference constraints

Note that we can have a Print statement within a *Scan* and within the *PageHeaderBlock*. This would essentially lead to meaning that the same *Print* statement can be referenced from multiple model elements. Although this enables the end-user to reuse the same *Print* statement it will eventually lead to propagation problems. If the end-user wanted to change the fields of one of the referenced *Print* statements they would be essentially changing the fields for any other model element that references that *Print* statement which would lead to

possible propagation problems. We solved this issue with a simple solution. Before any new reference is made to a model element, in this case the *Print* statement, we check whether it has any existing references. If it does we inform the user and reject the new reference.

6.4.1.8 Shape/Connector definition

Each shape and connector associated with our RWL meta-model element was designed and implemented with simplicity and the end-user in mind. For RWL elements which can contain other nested elements we used a compartment shape and for others we simply used a geometrical shape with the exception of the *ControlLine* model element which we represented using an image shape.

While designing compartment shapes we followed a consistent standard as shown in Figure 180 and Figure 184. On the top left we have an icon matching the icon of that model element within the toolbox, in the centre we have a marked stereotype for that shape and under it is the identifying information if it exists (for e.g. there is no identifying information for a *Print* statement). Each compartment shape has a single compartment labelled *Items* which contains its children.

Geometrical shapes follow a simple standard and only contain the stereotype in the centre (for e.g. *StandardPageHeader* and *StandardReportFooter* shown in Figure 184).

Relationships are represented by connectors and the core flow relationship is represented by a thick directional connector indicating the flow (Figure 184). If the connector represents an ordered child element the order is also indicated as shown in Figure 180 and also shown in Figure 184.

Note that the above mentioned shape mappings are not concrete and can be changed at any time to reflect new requirements or end-user requests.

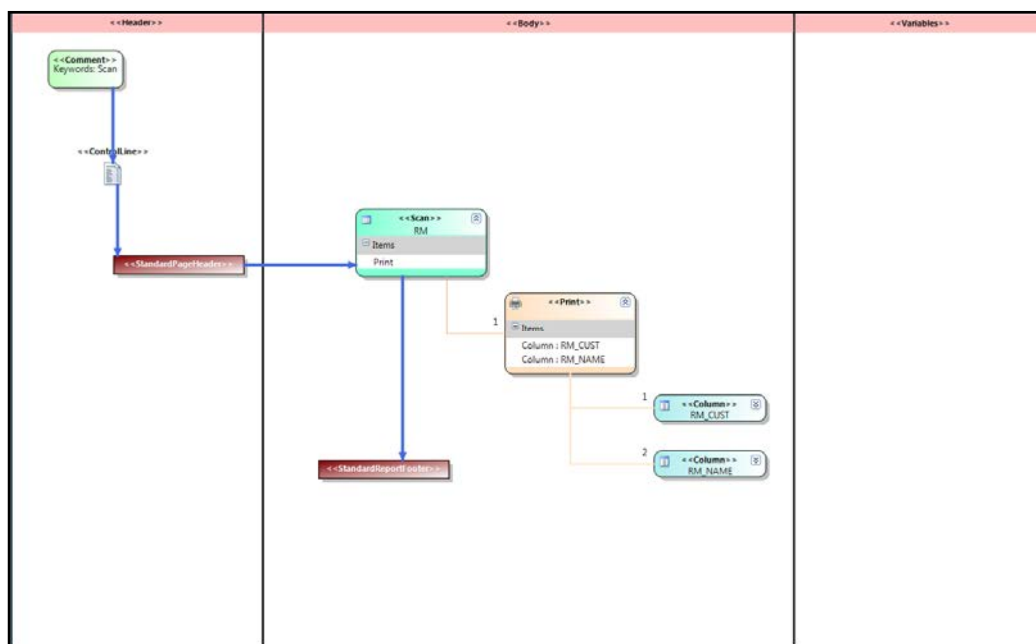


Figure 184: RWM Shell Approach: Instantiated RWM model example

6.4.1.9 Toolbox/Explorer View

We exposed our meta-model elements to the end-user with the help of the toolbox. The elements were divided into two main groups: Elements and Connectors. This made it possible for the end-

user to quickly navigate their way around different elements and connectors. Each model element was given a unique icon and the connectors were intuitively named for easy access.

Note that although we have numerous relationships between RWL constructs, for e.g. a relationship between a *Scan* and *PrintStatement* and a *PageHeaderBlock* and *PrintStatement*, all these relationships can be created by the end-user using the "Connect Elements" connector. This connector is intelligent and knows which RWL construct is the source and which is the target and automatically creates the appropriate relationship. The "Sequential Order" connector is used when the end-user wants to represent top level RWL model flow. For e.g. If a Scan follows the ControlLine construct then the end-user will represent this by using the "Sequential Order" connector by dragging from the *ControlLine* (source) to the *Scan* (target).

The explorer view was also implemented using the same icons as the user saw in the toolbox keeping the shell consistent. The explorer view named elements according to their identifying information. For e.g. a *Scan* would be identified by its underlying database view name.

The toolbox and the explorer are both shown below in Figure 185. Note that the explorer shown below is for the RWM model shown in Figure 184.

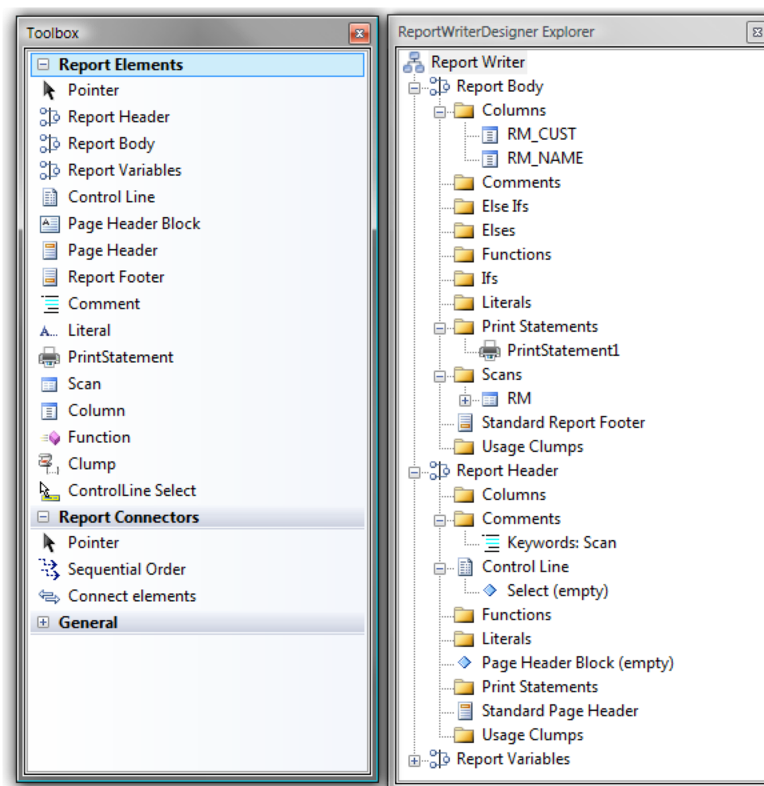


Figure 185: RWM Shell Approach: Toolbox/Explorer view

6.4.1.10 Custom Context-Menu Support

How a custom context-menu was implemented was explained in 6.3.1.9. Our current implementation only has one custom menu item for triggering the auto layout algorithm on user request. The auto layout algorithm was explained as part of Section 6.4.1.6.

6.4.2 Shell Host Development

The shell host for our RWM Shell approach was provided by Microsoft in the form of the experimental hive (while testing) and the Visual Studio Isolated Shell (while deploying). This meant that we did not have to implement any UI features within it for our newly created RWL meta-model. Although we did have to implement a set of text templates which would take an instantiated RWL model created by the end-user and generate the script from it.

6.4.2.1 Code Generation

The code generation process for the RWL model was a three step process: generate header script, generate variables and generate body script. Out of these three, two were implemented and we were slowly incorporating the variables part of the RWL. Note that we are using an Agile Development process therefore implementation is an iterative process.

We generated the RWL script from the model using the T4 Text Templating Engine (outlined in Section 5.7.2.3) in a modular fashion and the flow is shown below in Figure 186.

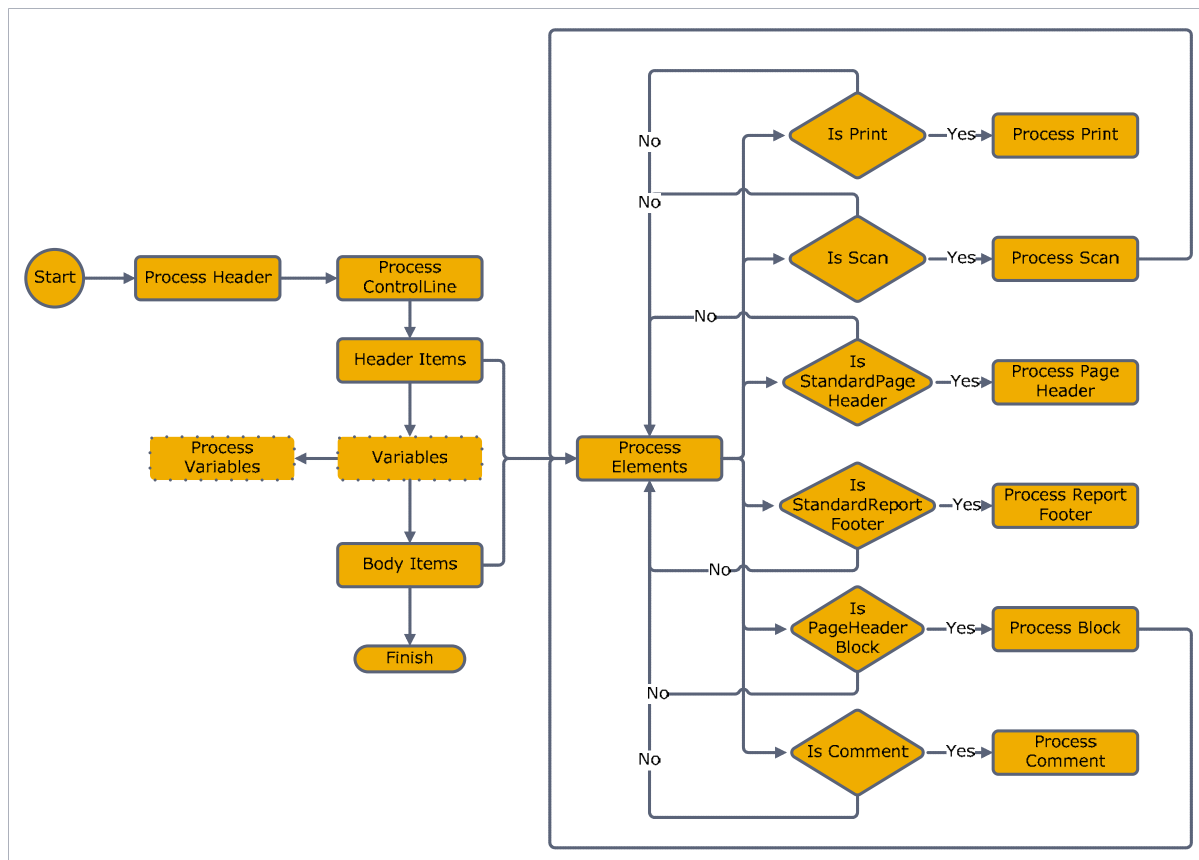


Figure 186: RWM Shell Approach: RWL script generation

As mentioned earlier each section of the report is processed separately although the processing of the variables section is not implemented yet and therefore has a dotted border in the figure above. Each element within that section is then processed. If the element is an element which can contain other nested children such as a *Scan* or a *PageHeaderBlock* we have to process those child elements

too, shown as a feedback loop within the above figure. These child elements are extracted in a sorted list according to the *Order* property of their corresponding relationship with their parent.

Processing individual elements to print RWL script is a trivial process and requires no sophisticated knowledge allowing future developers to easily modify and add enhancements to the code generation process. It also allows them to easily add support for new RWL constructs as they get added to our meta-model.

6.5 Challenges Faced

6.5.1 Class Diagram Approach

The most challenging part of the Class Diagram Approach was to represent the model elements and connectors in WPF and expose them to the end-user via the UI. Our approach involved providing a separate shape for each RWL construct and allow them to be connected to each other with the use of connectors. Because we were new to WPF technology the learning curve was quite steep and it took a lot of resources in terms of effort and time. This was one of the main reasons why we discarded this approach.

Other challenging parts of the implementation involved representing meta-model constraints in the WPF UI and validating various model elements. We also had trouble representing the meta-model of the RWL using our meta-meta-model (class diagram). Due to our inexperience with meta-meta-model and meta-model development we could not design a comprehensive class diagramming tool (meta-meta-model) which would allow us to create the RWL meta-model.

We presumed that we could have faced other possible challenges with respect to scalability, robustness and code generation accuracy if we continued with this approach. Although this approach was discarded in its early stages we do not discuss those challenges further.

6.5.2 RWM Shell Approach

During the implementation of our core solution (RWM Shell Approach) described in Section 6.4, the most difficult part was to design the common hierarchy. This was made even more difficult as the DSL Tools could not represent *Interfaces* therefore classes had to be abstract and the C# language has a constraint that a given model element can only extend one abstract class, therefore the hierarchy had to be designed so that it could cater for all possible combinations of the RWL model.

The common hierarchy is explained in Section 6.4.1.2. This structure had to correctly model the RWL and therefore had to be designed with precision and flexibility. Precision would allow our meta-model to correctly enforce RWL constraints and flexibility would allow new RWL constructs to be added seamlessly. After this hierarchy was implemented adding new RWL constructs to our meta-model was trivial.

Other challenges faced involved finding a point where we could demonstrate the prototype to an audience and also determining the robustness, scalability and code generation accuracy of the tool. Because of the time constraints we could not implement all RWL constructs within the meta-model although extra effort was put in so that the existing RWL constructs within the meta-model are robust and complete.

Scalability of the meta-model and the end-user tool were also a concern during the implementation. As the meta-model grew and new RWL constructs were added it became increasingly evident that representing a complex structure such as the entire RWL would involve precise design and planning. Screen area was a major concern while designing the meta-model as we could not fit the entire meta-model within the given screen area and ended up scrolling up/down and left/right continuously to see the big picture. These problems were also experience while we were designing example RWL models using the Visual Studio Shell. We soon noticed that larger RWL models would encompass a large screen area and may prove to be harder to follow than its textual counterpart. We get round this problem to a certain extent by allowing end-users to collapse and expand required model elements to hide and show their children giving us a bigger view of the RWL model. More sophisticated layout algorithms can also be implemented to maximise screen real estate if need be. This is discussed further as a future enhancement in Section 9.5.3.

The end goal of the prototype was to generate the corresponding RWL script from the RWL model. The challenging part of this aspect was to determine the correctness of this output. Due to time constraints we could not test every possible RWL model combination therefore we cannot say with certainty that the generated RWL will always be correct. It would be ideal to create test cases where a variety of RWL models get created and the generated RWL script can be validated. Currently the only way we can validate a RWL script is to import it back into the Prism WIN MIS system although in the future it would be ideal if the reporting engine of the MIS system could be separated out and used explicitly by the RWL model to determine the correctness of the generated RWL.

6.5.3 Approach Comparison

The preceding sections described the implementation of both approaches taken during the course of this thesis. We use Table 4 below to highlight some of the key differences and similarities between the two approaches to explain why one was preferred over the other.

Table 4: Class Diagram Approach vs. RWM Shell Approach

Criteria	Approach #1 (Class Diagram Approach)	Approach #2 (RWM Shell Approach)
Stages	1. Develop meta-meta-model 2. Develop meta-model and code generators 3. Develop UI	1. Develop meta-model 2. Develop code generators
Challenges faced	1. Implementing UI 2. Representing constraints 3. Constraint Validation 4. Representing shapes and connectors in WPF 5. Scalability 6. UI robustness 7. Code generation accuracy	1. Common hierarchy 2. Scalability 3. UI robustness 4. Code generation accuracy
Database access	LINQ	LINQ
UI Technology	WPF • Sophisticated/modern • Custom implementations of everything from menu	Visual Studio Shell • Provides basic shapes and connectors • Common functionality already

	items to shapes and connectors	implemented e.g. load, save, undo and redo
Constraint Validation	Explicitly via the meta-model which was designed using the meta-meta-model	Implicitly done directly via the meta-model within the DSL Tools
Code generation	Developer specified via the "Generate(...)" method	T4 text templates

From the above table we can see that the Class Diagram Approach involved more challenges when compared to our RWM Shell Approach. Moreover it did not provide us with the tools to implement a prototype within the time constraints of this thesis therefore we decided that Approach #2 (RWM Shell Approach) was the ideal solution given the time frame and resources.

6.6 Summary

This chapter gave a detailed look into the implementation details of our solution with respect to the two individual approaches. It gave us details on how the meta-model was implemented using the Microsoft DSL Tools, how constraints were specified, how the UI exposed the meta-model and eventually how the code was generated was generated.

From this chapter we can clearly see why the RWM Shell Approach was the preferred approach as it allowed quick development of the meta-model and the prototype using the Microsoft DSL Tools. Also as seen from this chapter, the Class Diagram Approach offers flexibility in terms of sophisticated UI development although making it hard to represent the meta-model elements, relationships and constraints graphically within the time frame allowed by the thesis.

The biggest challenge faced during the implementations of the Class Diagram Approach and the RWM Shell Approach were the implementation of the WPF UI via which the end-user could design the RWL model and the design of the common hierarchy respectively.

Chapter 7 - Case Studies

7.1 Introduction

This chapter involves looking at different examples on how the end-user and the developer may interact with the system.

We start by looking at the primary reason for the thesis which is to assist end-user to visually create reports. The other two case studies are aimed at future developers who may want to modify the meta-model which represents the RWL.

Each case study is divided up into four main parts: we initially outline the requirements followed by our design and analysis of those requirements and then we detail how this requirement can be met followed by any conclusions reached.

7.2 Scenario 1: Designing a report

Our primary case study is to give a walk through on how an end-user can potentially design a Prism report. We assume that the specifications of the report have already been established and the end-user has an idea on what the report script looks like. This assumption is mandatory as our tool only assists the end-user to design the report and offers no assistance in the specification analysis on the need for that report.

7.2.1 Requirements

This sub-section describes the requirements of the report we are going to be designing using the shell. The requirement it to design a report to extract all the customers from the Prism database and get their corresponding jobs. This should not include quotes.

To make this case study simple, the information can be printed in any layout as long as all jobs for a given customer are shown.

7.2.2 Requirement analysis

The requirement involves customers and their jobs meaning that we are dealing with the RM and QM tables respectively. Knowing this is not necessary as we can simply use the intuitive custom field editors to find out this information. Furthermore, we are only required to show jobs and not quotes, this is done by a simple conditional check on one of the columns within the QM table. We make things simpler by making the assumption that the report can be accessed via any suitable Window[2] and its unique identifier *Code* can be any valid string.

We therefore need to represent the following RWL script using the newly developed shell:

```
1.  Code      CASE_STUDY_1
2.  Type      Standard
3.  Access    STSR
4.
5.  Scan RM
6.      Print RM_CUST + RM_NAME;
7.      Print “All Jobs For ” + RM_NAME;
8.      Scan QM
9.          Choose(QM_CUST_CODE, MATCH, RM_CUST)
10.         Choose(QM_QUOTE_JOB, MATCH, QMM_JOB)
11.
12.         Print QM_JOB_NUM + QM_TITLE;
13.     End
14. End
```

```
15. Print     StandardReportFooter;
```

Note that with the above textual script the end-user has to be familiar with the Prism database structure in order to determine the correct *Scan* and *Choose* conditions. Moreover, they also need to know the semantics and the syntax of each of the RWL constructs shown in the above script to correctly form the script so that it can be executed by the Prism WIN MIS system.

7.2.3 Meeting the requirements

We show how the visual shell RWL meets these requirements by giving a walkthrough on how to represent the RWL script using the newly implemented shell. Screenshots corresponding to each step is attached as Appendix C (some figures are in line to enhance readability).

1. Create a new file of type RWM using the Shell. Give the file a name; in this case we called it CaseStudy1.rwm.
2. After creating this new RWL model, we have a new designer which automatically has three predefined swimlanes: *Header*, *Body* and *Variables* shown below in Figure 187.

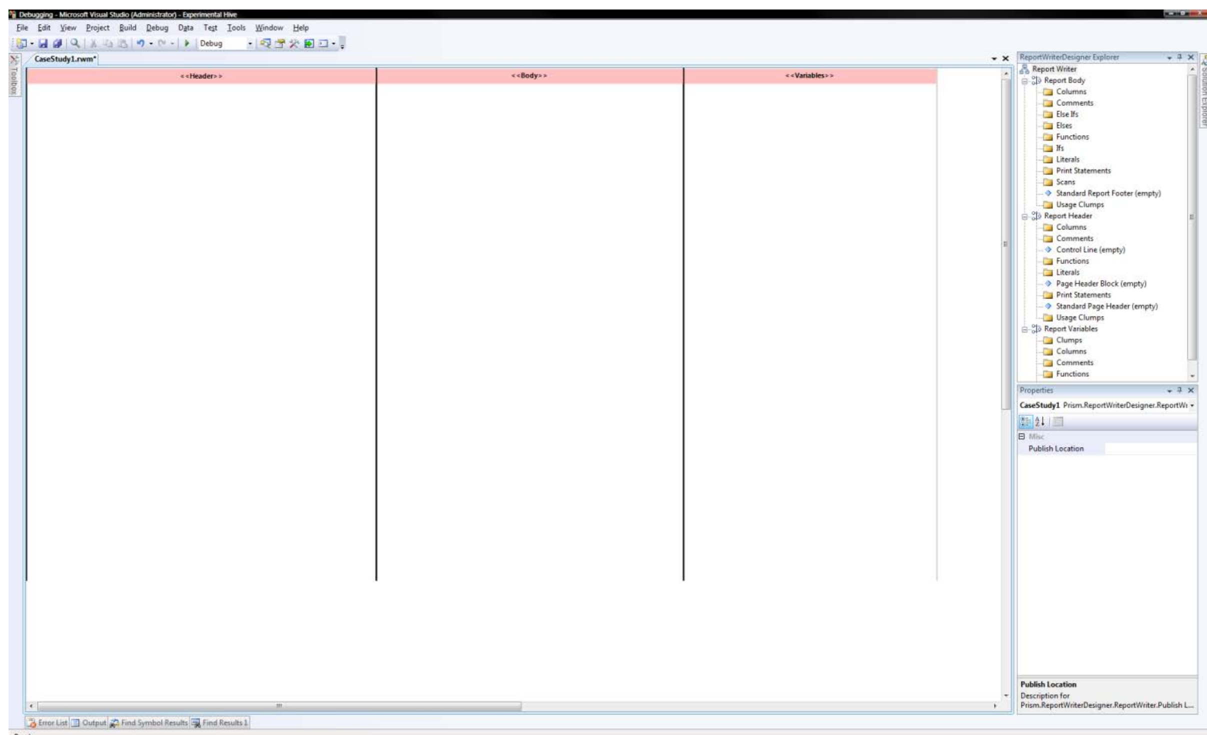


Figure 187: Scenario 1 – Initial RWL Model

3. We start by adding a ControlLine to our designer. This can be achieved by dragging and dropping a ControlLine element from the toolbox on the left hand side on to the header section of the designer.
4. Now we have a ControlLine element on our designer canvas. We now modify its properties.
    a. The Code field should hold CASE_STUDY_1
    b. The Access field should hold STSR (done by clicking on the "..." next to the field and selecting the corresponding value as shown in Figure 168)
    c. The Type field should hold Standard (done by clicking on the drop down and selecting the appropriate value)

5. We can now start adding the bulk of our report to the designer by initially adding a Scan to the designer. This is done in a similar fashion to how the ControlLine was added although we add the Scan to the body section.
6. After adding the scan we need do two things, connect the scan up with the rest of the report and indicate the table on which the scan needs to be on.
    a. The scan is after the ControlLine, we show this by connecting the ControlLine and the scan with a “Sequential Order” connector. Click on this connector, click on the source (ControlLine) and drag to the target (Scan). This will connect both these elements and you will see a directional arrow going from the ControlLine to the scan.
    b. The Table field on the scan element can simply be typed in as RM (if we did not know the table we need, we could click on “...” which opens a window showing us all the tables in the Prism database).
7. Within the scan we have two print statements. We add two print statements to our body section by dragging and dropping them from our toolbox. We can then connect them to our scan using the “Connect elements” connector. The source is the scan (where the drag starts) and the targets are the print statements (where the drag ends). Note that each scan - print statement relationship has one connector. Note the number at the end of each connection, this represent the sequence in which the print statements will be shown in the generated RWL script. Refer to Figure 188 below.

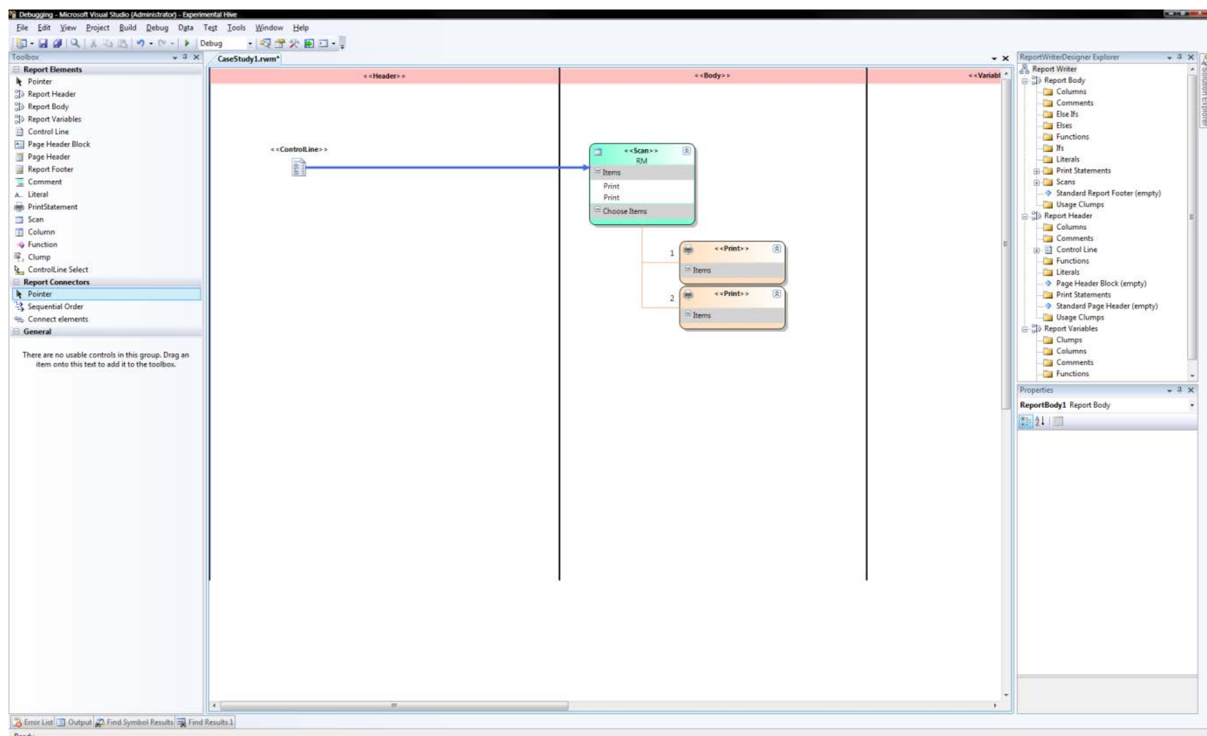


Figure 188: Scenario 1 – Partial RWM Model with connectors

8. The first print statement prints out RM_CUST + RM_NAME which are two database columns. These can be added via the toolbox by dragging and dropping them onto the body section. Connect them to the desired print statement (the first print statement) using the “Connect elements” connector.

9. Now that we have two column elements, we need to add which columns they represent. This can be done by selecting a column element and changing its “Value” field to represent the database column it maps to. Click on “...” next to the field, this will open up a window which lists all the columns belonging to a particular database table, in this case, columns belonging to table RM (as we have a scan on RM).

   Select RM_CUST for the first column element and RM_NAME for the second.
10. The second print statement prints a literal (All Jobs For) and a column, this can be done by adding a literal and another column on to the body section and connecting them to our print statement using the “Connect elements” connector. Change the literal and column by selecting each element and modifying its fields.
11. The next element the scan contains is another scan, although this scan is on table QM. Adding a scan and setting its table is already described in Step 6b. This scan needs to be connected to its parent scan (Scan RM) using the “Connect elements” connector.
12. The QM scan has one print statement which prints two columns: QM_JOB_NUM and QM_TITLE. This is similar to Step 8 and Step 9.
13. Now we have a nested scan although we do not have a joining condition, this can be done by selecting both the scans and then right clicking (shown below in Figure 189). This shows a menu which allows us to join these two scans.

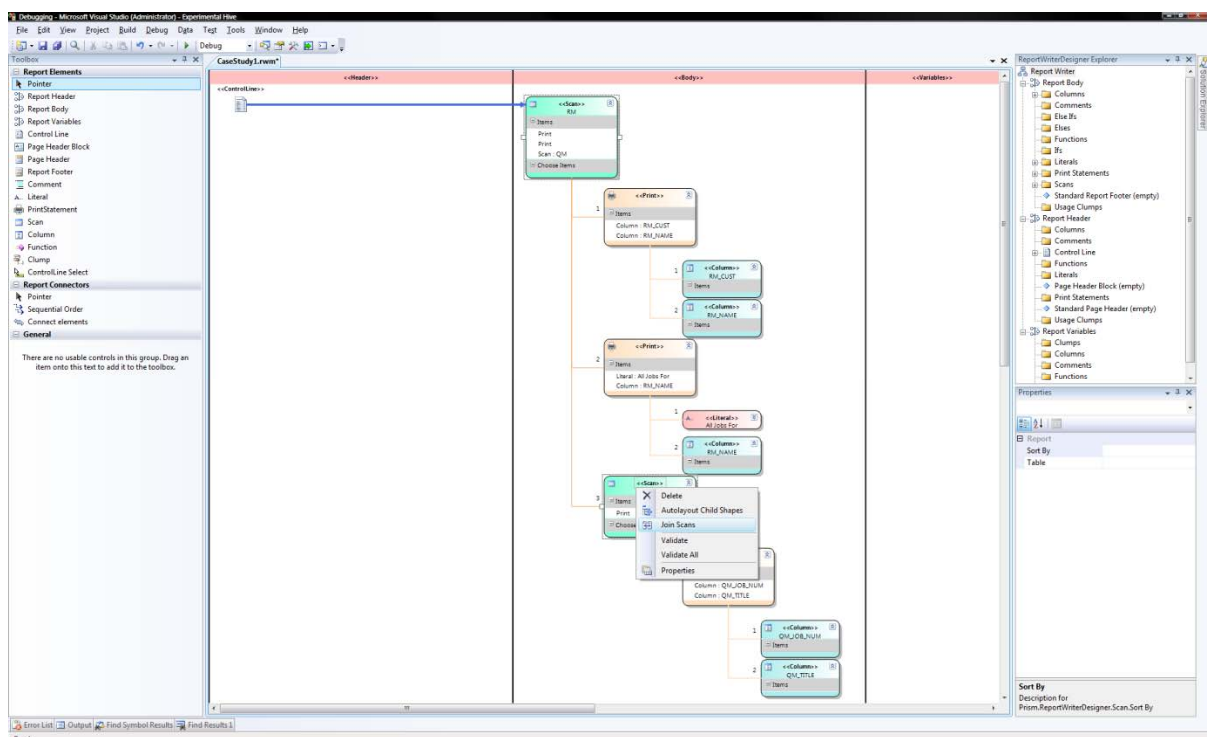


Figure 189: Scenario 1 - Joining two Scan model elements

14. Clicking on this menu item opens a window via which we can select a join condition. In this case there is only one join condition which joins RM and QM. After selecting this condition click on OK. This join condition appears as a Choose statement within the scan.

15. Note that we also need to add another choose statement as we only want jobs. This is done by right clicking on the "Choose Items" compartments and selecting "Add new Custom Choose". This adds a custom choose item to the scan.
16. Change the fields of this newly added choose item, the inner condition should point to the column which we want to match, in this case QM_QUOTE_JOB and the outer condition should contain the value. As QM_QUOTE_JOB contains only three values, these are automatically populated for you. Select the appropriate one, in this case QMM_JOB as we want only jobs.
17. Our final step is to add the report footer. This is done by dragging and dropping the "Report Footer" element from the toolbox onto the body section. We then have to indicate that the scan (Scan RM) element flows on this element by using our "Sequential order" connector in a similar fashion as described in Step 6a. The final model is shown below in Figure 190.

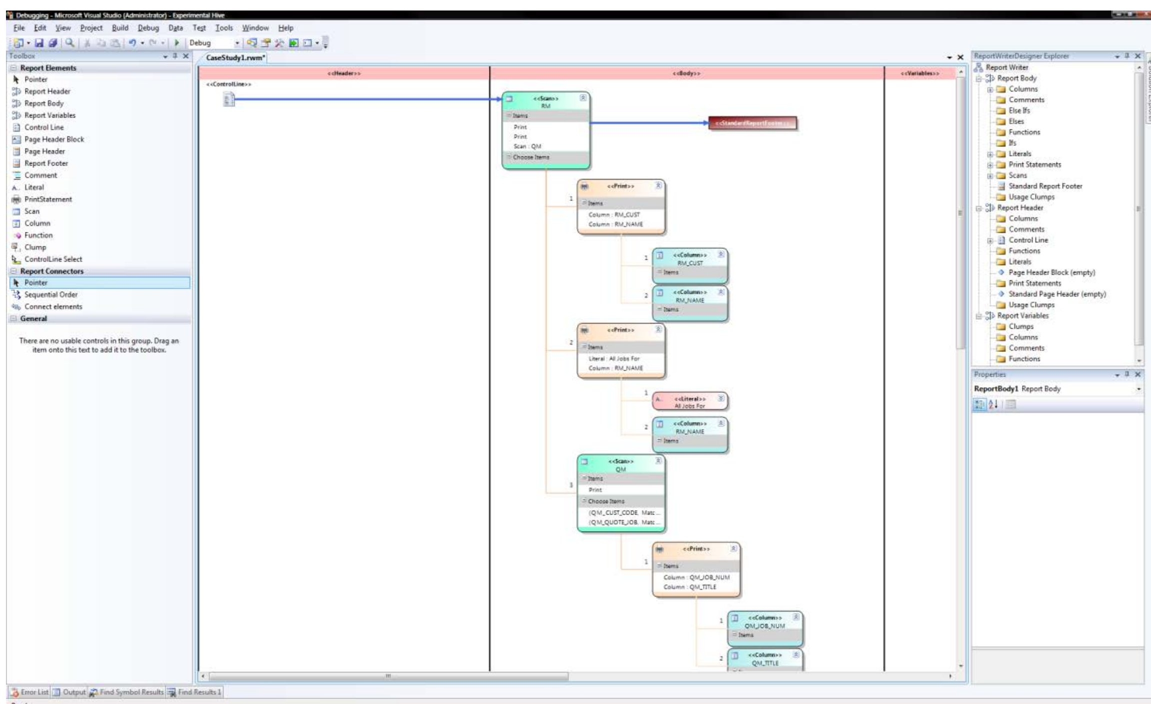


Figure 190: Scenario 1 - Complete RWL Model

18. We now have our required RWM model and are ready to generate the code from it. As we are working in the Experimental Hive this can be done by clicking on the "Transform all Templates" button on the solution explorer which will generate the corresponding RWL script.

7.2.4 Conclusion

It can be seen that if the specification of a report is known it is trivial to represent it using our newly implemented shell. Complicated aspects such as column selection and joins are made simple using automation and simple intuitive design with minimal user input. The RWL script generated from such a model is also neatly formatted and can easily be read by the end-user to determine its correctness. We do a formal evaluation of our reporting tool using end-users and the results are analyzed in the following chapter (Chapter 8).

7.3 Scenario 2: Adding new model information

This case study represents a simple developer task which involves adding new a RWL construct into our existing meta-model. The new construct fits into the meta-model without any major changes as opposed to doing it traditionally.

Traditionally we would have to change code manually to incorporate this new construct which may involve writing a toolbox menu item, designing a shape, adding connectors and compiling all the changed code. Not to mention time and resources to test a small change this would mean retesting the whole software system if it is strongly integrated. This brings us to the second key idea of the solution which is to enhance developer experience and allow them to bring about change faster and improve code quality.

We describe what construct we are going to be adding and then analyze on how it can be done followed by detailing the steps involved in making this change.

7.3.1 Requirements

The *ControlLine* construct within our meta-model has a field called *Select* according to the RWL specification described by (Prism New Zealand, 2005). This field allows users to specify run-time selections which can constrain embedded *Scan* statements within the report. We need to add functionality so that the *Select* field can hold columns and specific functions that can be executed on a column which is within the *Select* construct. This new meta-model contruct should be exposed to the end-user via the toolbox and should generate the appropriaye RWL script when used in a RWL model.

The format of a *Select* statement is:

Select column_code[.Match/From/To/Range/Ask/Not][.Force][+...].

7.3.2 Requirement analysis

The requirements expect a new domain model element to be added called *Select*. From the requirements we can see that this model element is part of the *ControlLine* model element therefore we can safely embed it within the *ControlLine*. After adding this new model element we will need to reference any other elements that the select can hold. As far as the requirements are concerned, the *Select* contains *column_codes* (*Columns*) which in turn can have functions performed on them. We can safely assume that the *Column* and *Function* model element are also implemented for us and that a *Column* can reference a given *Function*, shown in Figure 191.

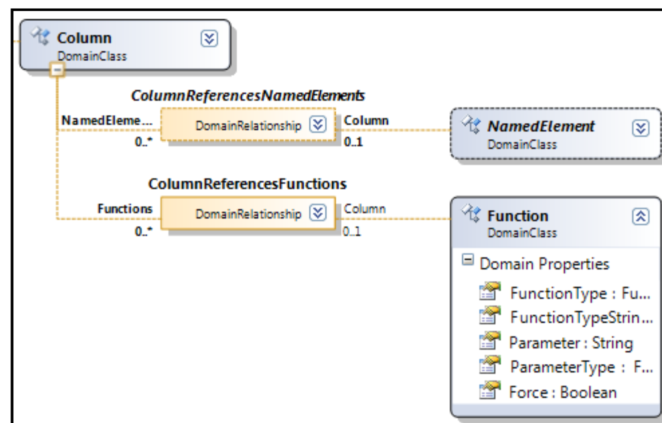


Figure 191: ColumnReferencesFunction relationship

Now that we have both our referenced classes, *Column* and *Function*, we need to confirm that the new function types exists (e.g. *Match*, *From*, *To*, *Range*, *Ask*, *Not*). Another requirement is the *Force* value on a function. This value is specific to functions which are on columns which belong to the *Select* model element therefore we need some rule which only shows this field depending on whether we are on a *Select* model element or not. After implementing the meta-model side we need to consider the design of the template which would generate code from the *Select* model element. We decided that code generation for the *Select* model element will be done as part of the *ControlLine* processor (template which processes the *ControlLine* model element).

7.3.3 Meeting the requirements

Implementing the given requirements involves a many steps therefore what we show here is a brief overview of what is required to meet the given requirements.

7.3.3.1 Adding and Connecting the Select Model Element

A new domain model element is added, we change the name of that model element and call it *Select*. Because this model element will be embedded within the *ControlLine* model element it is not required that we make it inherit from any other model element. Although all our model elements inherit from *NamedElement*, so we follow this standard and make the *Select* model element inherit from *NamedElement*. The *NamedElement* is shown in Figure 173 and detailed in Section 6.4.1.2.

Next step is to connect the *Select* model element with our *ControlLine*. We do this by using an embedding relationship where the *Select* is embedded within the *ControlLine*, what we end up with is shown below in Figure 192.

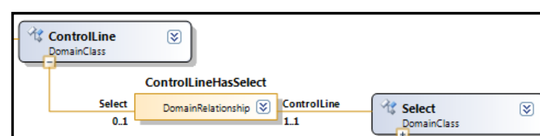


Figure 192: Embedding Select within the ControlLine

From the above figure we also note the multiplicity of the newly added *Select* model element, it is 0..1. This means that a *ControlLine* can have at most one *Select* model element because there is only one “select” field within the *ControlLine*.

After embedding the *Select* model element with the *ControlLine* we are now ready to actually define other model elements that the *Select* could reference. As the requirement stands now, the Select model only references the *Column* model element. This means that we simply add a reference relationship between the Select and the Column model element and we end up with Figure 193.

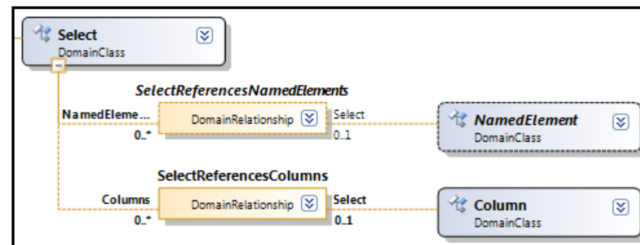


Figure 193: Select references columns relationship

Note in the above figure that we have *SelectReferencesNamedElements* and *SelectReferencesColumns*. *SelectReferencesNamedElements* acts as an abstract base relationship which we can use in the future in case we need to add more model elements to our *Select*. The relationship hierarchy is shown below in Figure 194.

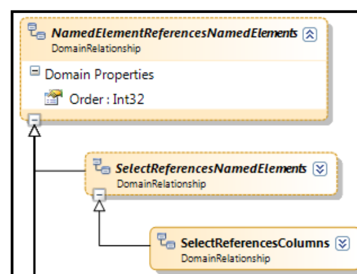


Figure 194: Select relationship hierarchy

Notice in the above figure that we also inherit from *NamedElementReferencesNamedElement* which contains an *Order* property. The significance of this relationship was explained in Section 6.4.1.6 and because columns within a Select occur in a specific ordering it is clear that our relationship has this property too.

7.3.3.2 Shape and Toolbox Definition

The next step is to map our new model element to a shape. We adhere to our shape notation standard which was described in Section 6.4.1.8. Because our new Select model element can contain other elements such as Columns it was best if we displayed it using a compartment shape. This would allow us to meet our business requirement of showing containment, ordering and hiding/showing children (detailed in Section 6.4.1.6). This allows us to end up with a shape as shown below in Figure 195.

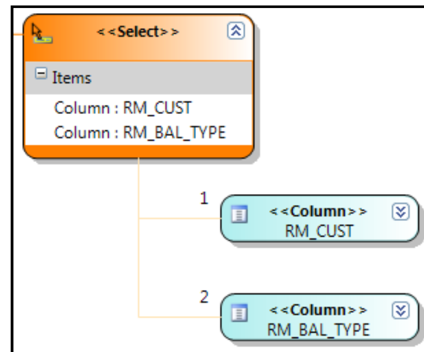


Figure 195: Select model element shape

Note the icon on the top left corner and the *<<Select>>* stereotype in the centre in the figure above. Hiding/showing children, correctly ordering child elements within the "Items" compartment and automatic layout of children are implemented in base classes which forms the essence of our visual notation. All we have to do is inherit from a base compartment shape (shown below in Figure 196) and invoke some functions.

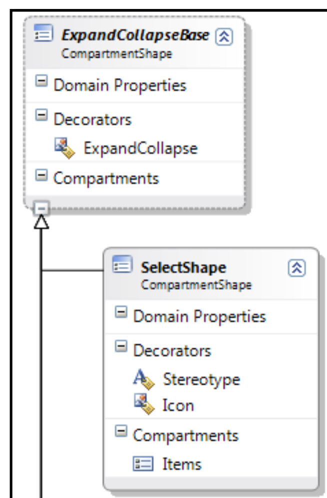


Figure 196: Select model element shape definition

For code details on how our base shape achieves hiding/showing children refer to Appendix D.

Correctly ordering child elements within the compartment is done via a simple function which takes an unordered list of child relationships (all relationships inherit from a base which contains the *Order* property) and is then returns a sorted list which is sorted on the *Order* property. The code is attached to Appendix E.

Automatic layout is also done via a simple function which takes a shape, a start point and returns the location of the last child. The code is attached to Appendix F.

After implementing the shape definition for our new model element we are now ready to expose it to the end-user via the toolbox. This is the simple part where we add a new *ElementTool* to our editor and map it to our newly added *Select* model element as shown below in Figure 197.

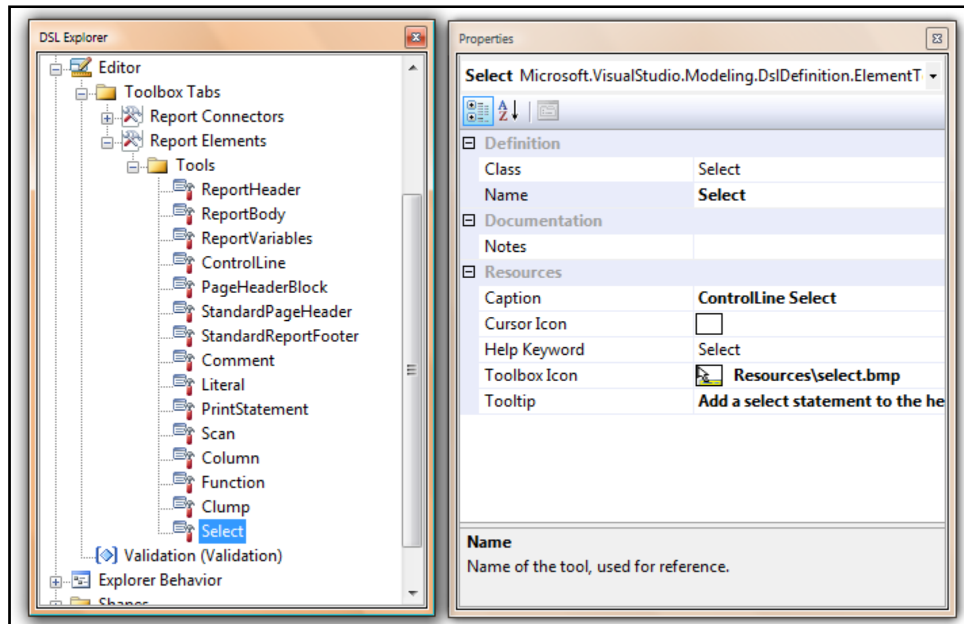


Figure 197: Select toolbox item definition

With the above definition we end up with a toolbox item as shown below in Figure 198.

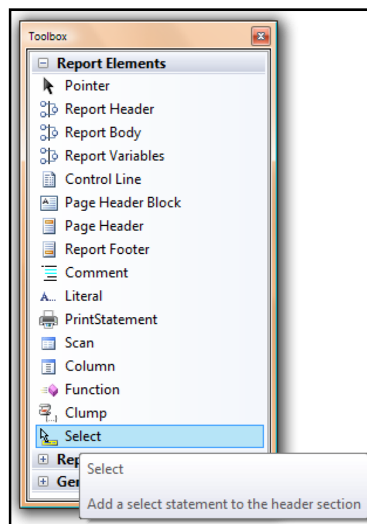


Figure 198: Select toolbox item

7.3.3.3 Select specific Functions on Columns

Figure 199 below shows all the various function types we have currently implemented in our meta-model. Note that the only functions within this list which can be part of a *Column* which is associated to a *Select* model element are *Match*, *From*, *To*, *Range*, *Ask* and *Not* as outlined in Section 7.3.1.

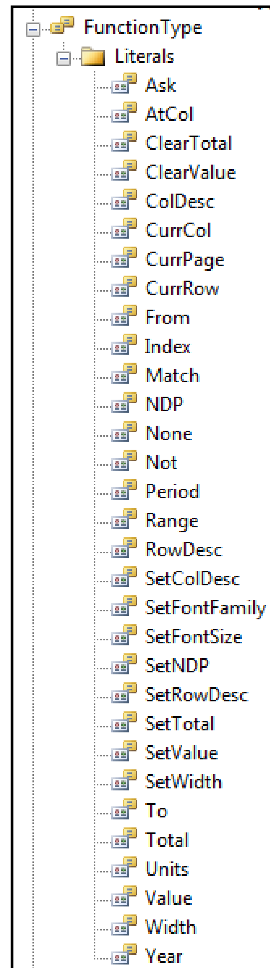


Figure 199: Various RWL function types

We can filter the entire function list using .NET CLR Attributes. We used attributes to define our custom field editors as described in Section 6.4.1.4 and we use them to annotate each function type so we can identify them and use them according to the model element which hosts the *Column*, in this case, the *Select* model element. Any *Function* which can be added to a *Column* which is on a *Select* model element is annotated with the *FunctionAllowedOn* attribute with *AllowableType.ControlLineSelect* as its parameter. *AllowableType* is simply a custom enumerand which contains values which indicate whether a given function can be executed on a column which is hosted by a given model element. For e.g. in Figure 200 below we see that the *Match* function can be executed on a *Column* which is hosted by either three model elements: *Variable*, *DataField* and our newly added *Select*.

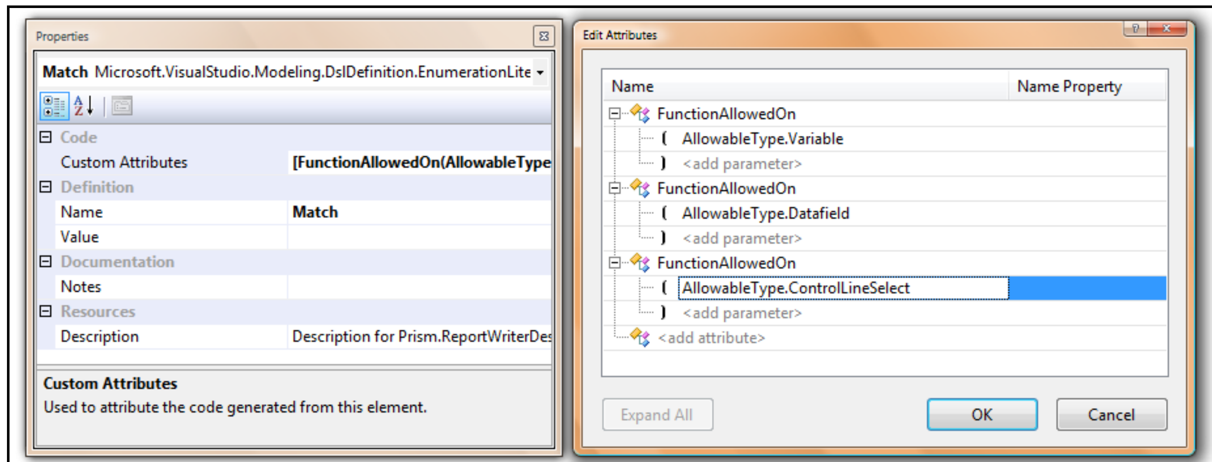


Figure 200: Select-Column functions annotated using attributes

After we annotate our function types with the *FunctionAllowedOn* attribute all we have to do is extract all function types depending on its corresponding *AllowableType* value which should match the corresponding host model element.

7.3.3.4 Implementing the Templates

We have added the model element, its connections, exposed it via the toolbox and added any rules surrounding it although we have not implemented how this model element will generate the RWL code. This is done using templates which we look at in this section. Template which caters for the Select model element is shown below.

```
public void ProcessSelect(Select select)
{
        StringBuilder builder = new StringBuilder();
        builder.Append("Select" + GetTabs(2));
        List<NamedElement> namedElements = Select.SortedChildren(select);
        for (int i = 0; i < namedElements.Count; i++)
        {
                if (namedElements[i] is COlumn)
                {
                        Column column = namedElements[i];
                        builder.Append(GetColumnAsString(column));
                }

                if (i != select.Columns.Count - 1)
                        builder.Append(" + ");
        }
#><#=builder.ToString() + ";" + Environment.NewLine#><#+
}
```

We assume that we already have templates which generate the RWL script from the *Column* model element. The *Column* templates will take care of functions so this leaves with the template for the *Select* model element which is trivial. All we have to do is get our *Select* model element from the *ControlLine* and then extract a sorted list of its children (currently it will only contain *Column* model elements). We then iterate through this list and form our *Select* clause using our already existing templates which cater for the *Column* model element.

7.3.4 Conclusion

From this case we can see that adding new model elements is systematic and comparatively easy due to our simple implementation using base classes and static helper functions. We can also see that adding a new model element simply involves five steps:

1. Adding model element
2. Connecting model
3. Defining rules/constraints
4. Shape definition (Toolbox definition)
5. Template implementation

7.4 Summary

This chapter gave us two case studies, each aimed at a different aspect of the solution. We looked at a step by step tutorial on how an end-user would design a report and an overview on how a developer can add new meta-model information. From the case studies we can see that both tasks can be performed with simplicity and do not need extensive training programmes.

Chapter 8 - Evaluation

8.1 Introduction

This chapter describes the evaluation techniques and the results of the evaluation of the solution proposed by this thesis. We evaluated our solution during design time and also after implementation. This chapter gives an in depth view into both those techniques and also compares our solution with the current software used to design Prism reports, Scribe, outlined in Section 2.6.

The design evaluation highlights the process and results of evaluating our prototype while it was being developed and was an ongoing process during the course of this thesis. The design evaluation was carried out by project mentors, project supervisors and me during weekly meetings as opposed to our survey evaluation which was done after the implementation stage. The survey was a means to get feedback from actual end-users and developers who will be using our solution. The survey was approved by the University of Auckland Human Participants Ethics Committee (UAHPEC Ref. 2008/405) and is included as Appendix J.

This chapter only evaluates our RWM Shell Approach (second approach described in Section 5.8 and Section 6.4) as we discarded our Class Diagram Approach (first approach described in Section 5.7 and Section 6.3) in its early stages.

8.2 Design Evaluation

This section highlights the evaluation techniques we used during the design of the solution described in this thesis. We evaluated our solution at regular intervals during weekly meetings which involved my supervisors and project mentors and research meetings which involved other postgraduate students. Evaluation results were constantly integrated into our solution and as soon as new features were added (meta-model elements added/modified) another evaluation was done. This formed a cycle and made our design evaluation an iterative process where our solution was driven by the evaluation results. We look at two specific evaluation techniques in this section.

Note that the design evaluation was only concerned with the UI via which the end-user would use to design RWL report models. The meta-model via which our notation was designed is a Microsoft product (DSL Tools) therefore evaluating it during design time was not necessary although we do evaluate our meta-model via a survey which is described in Section 8.3.

8.2.1 Champagne Prototyping

Introduced in Section 2.4.3.6, Champagne Prototyping involves designing a simple prototype and evaluating it on a “look don’t touch” basis. Evaluation of this prototype can be done using Cognitive Dimensions which is explained in the following sub-section.

During the course of our thesis as we adding new meta-model elements we were potentially creating new prototypes each time we tested our implementation via the Experimental Hive. These prototypes helped us with the Champagne Prototyping technique. Also, these prototypes were partially functional allowing us test the notation as well as some other features such as containment, ordering and visual layout. We look at our evaluation criteria in the following sub-section as we look at each cognitive dimension and examine how our implementation caters for it.

8.2.2 Cognitive Dimensions

Cognitive Dimensions was introduced and explained as an evaluation technique for visual languages in Section 2.4.3.6, this sub-section looks at the various dimensions of that technique and demonstrates how the solution described in this thesis caters for those.

8.2.2.1 Abstraction Gradient

This dimension allows us to measure and evaluate the maximum and minimum levels of abstraction of our visual notation. The aim of our solution was to provide a visual RWL language which abstracts the users from the textual RWL and also the underlying database. Due to the vast nature of the RWL we can explain the maximum and minimum level of abstractions using a few RWL construct examples.

Maximum Level

Whenever we start with a new report canvas using our newly developed DSL it is a representation of the root of our model. Along with the root we have our report sections, currently three: *Header*, *Body* and *Variables*. Therefore we can define our maximum level of abstraction as the root model containing three reporting sections as shown below in Figure 201.

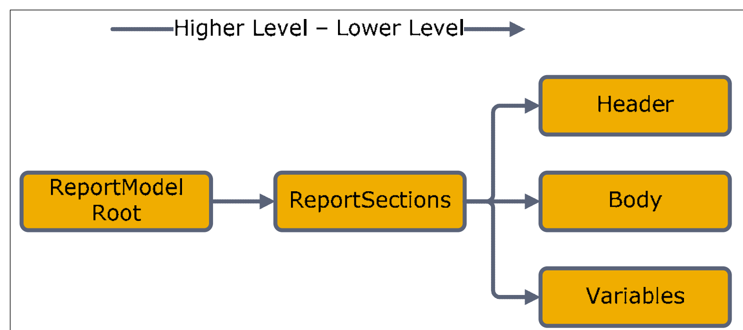


Figure 201: Maximum level of abstraction

Minimum Level

The minimum level of abstraction depends on which RWL construct we are dealing with. For e.g. if we are dealing with an independent construct such as the *ControlLine*, it has field level abstraction gradient because we are potentially abstracting each field within the *ControlLine*. This is shown below in Figure 202.

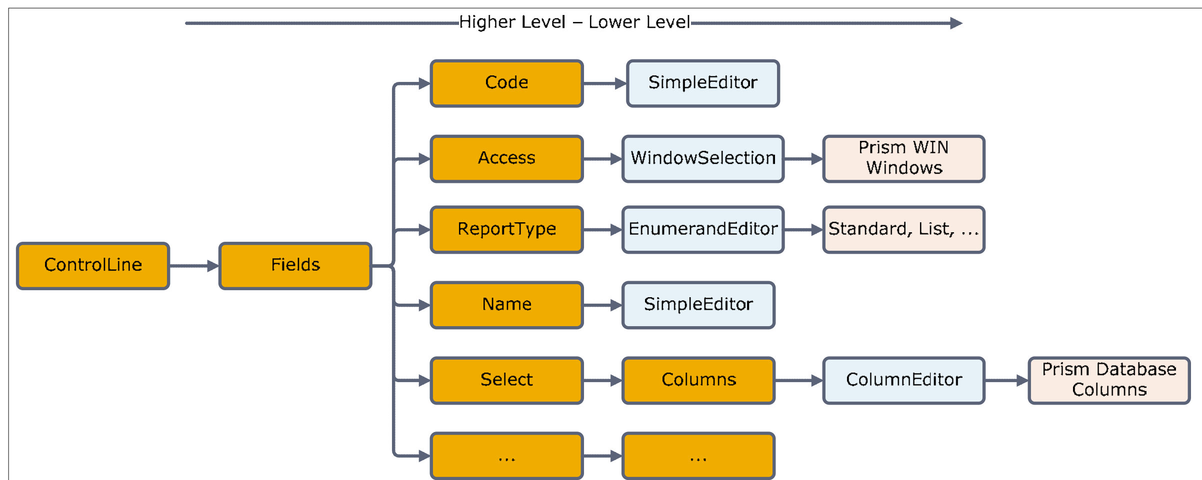


Figure 202: *ControlLine* construct minimum level of abstraction

In the above figure we see the various abstraction levels we have within our *ControlLine* construct. We can have a simple editor which the end-user uses to populate a field such as *Code* or we could have a sophisticated custom field editor for fields which abstract some aspect of the Prism database such as the *Access* field which contains a list of Windows[2] that the report can be accessed from. In that case the minimum level of abstraction would be the *WindowSelection* editor.

In our second case study which was detailed in Section 7.3, we catered for the *Select* field within the *ControlLine* which can be a combination of Columns. As shown in the above figure, the minimum level of abstraction for that field is the custom *ColumnEditor* which shows a list of columns which exists in the Prism database.

Furthermore, if we are dealing with a nested RWL construct such as the Scan we have a different minimum level of abstraction as shown below in Figure 203.

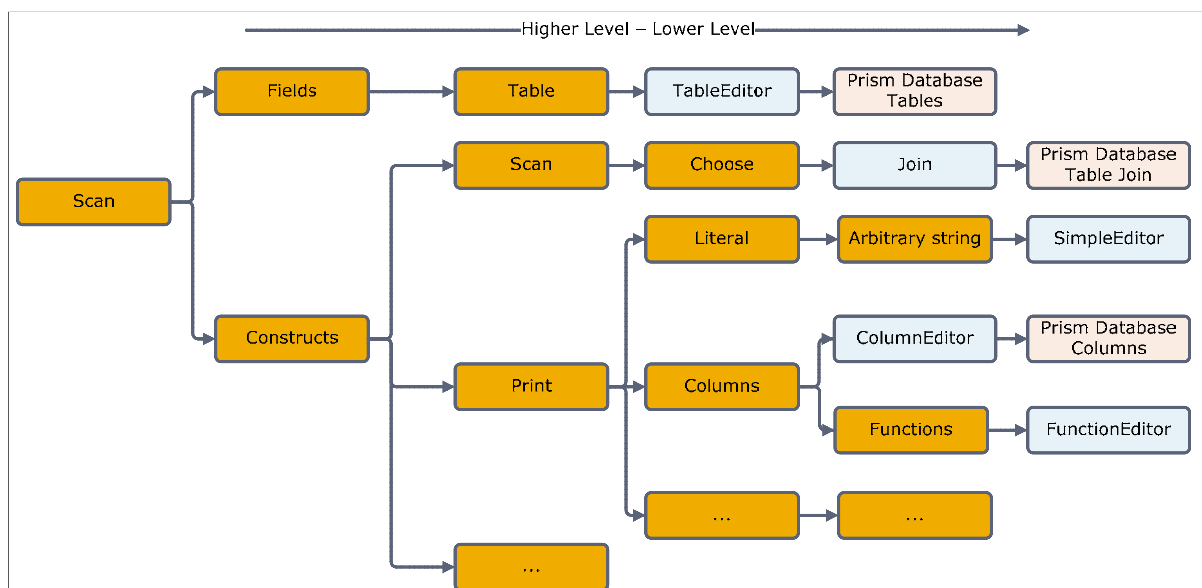


Figure 203: *Scan* construct minimum level of abstraction

In the above figure we can see that a *Scan* construct has a field called *Table* which can be edited using a *TableEditor* which abstracts the end-user from the Prism database tables. We can also see that a *Scan* can contain (nest) other constructs such as other *Scans*, *Print* statements, etc. This nesting nature can potentially be quite deep and we only attempt to show a portion of it in the above figure. For e.g. a *Scan* contained within another *Scan* can contain a *Choose* which abstracts a database join condition and a *Print* statement can contain a *Literal* (edited using a *SimpleEditor*) or *Columns* (edited via the *ColumnEditor* abstracting the columns within the Prism database).

8.2.2.2 Closeness of Mapping

This dimension measures how close our notation is to the domain we are trying to represent.

The visual notation developed during the implementation is not closely mapped to the domain as each domain aspect (RWL construct) is represented using a standard rectangular shape. We try to bring our notation closer to the domain with the use of icons which represent various domain aspects. For e.g. we represent the *Print* construct with a rectangular shape which has a printer icon on its top left corner as seen in Figure 184. Using rectangular shapes has its advantages as it correctly represents encapsulation and containment which is a key feature of the RWL (domain) we are mapping.

Other aspects of the RWL (domain) we map are its sequential nature and the order in which child elements are executed within a parent construct. The sequential nature of the RWL is intuitively represented using unidirectional arrows and ordering is represented using a numbering scheme, lower number means it is earlier in the order, both examples can be seen in Figure 184.

An alternate approach of mapping would be to allow a WYSIWYG (What You See Is What You Get) Prism Report Designer. This designer would actually allow end-users to design the report as they would expect the resulting report to look like. This approach is currently not possible as the layout of the end report is determined by complicated logic which lies deep within the Prism WIN MIS system and cannot be used. Moreover the report will have to be interpreted and executed against a database for which the Prism WIN MIS system is needed again. Due to this reason we identified the WYSIWYG designer/layout as a future enhancement which is explored further in Section 9.5.10.

8.2.2.3 Consistency

This dimension checks whether the remaining notation of a given language can be implied after part of it has been learned.

We believe that the visual notation which represents the RWL (domain) is consistent due to the fact that we use a similar shape for each construct and lines to represent relationships. If the user knows how to create one RWL construct by dragging and dropping its represented model element from the toolbox (shown in Figure 185) they can successfully do that for any other RWL construct.

This also applies to relationships. A relationship between two model elements is created simply by selecting the appropriate relationship from the toolbox (shown in Figure 185) and selecting the source and target.

Editing fields for a model element is done simply by selecting the model element and editing its properties using the Keyboard, if the field has a special editor attached then this field can be edited by simply opening that editor by clicking on the "..." next to the field (shown in Figure 168).

These are the only three aspects within our RWL notation therefore if the user can create any one of the RWL constructs from the toolbox, modify its fields and attach nested children to it then the user has potentially learnt the entire visual language.

8.2.2.4 Diffuseness

This dimension measures how many shapes or space is required for the notation to represent a required model or notion.

The visual notation representing the RWL is basically constructed from colour coded rectangular shapes and lines. Therefore we can clearly represent a RWL construct using this shape and its corresponding relationships using a line which can connect two shapes.

In our opinion the visual notation developed as part of this thesis is less diffused when compared to its textual counterpart. We come to this conclusion as each shape which represents a RWL construct is "templatized". For e.g. if the end-user wants to add a Scan construct, they add a Scan model element. This model element will potentially expand out to:

Scan

...

...

End

Therefore we can say that we use less space to represent a given RWL construct and therefore will take less space to represent a given RWL model (report).

8.2.2.5 Error-proneness

This dimension indicates whether the design of the notation induces "careless mistakes". We can safely say that because our visual notation of the RWL is simple and only contains shapes and lines it reduces the number of user errors to practically zero.

Trivial errors such as spelling mistakes made while textually designing reports are eliminated by our visual notation as there is essentially no textual input required by the end-user. Other errors which may occur due to the end-user not understanding the RWL semantics are also eliminated by our visual notation as it informs the user and expects them to rectify these errors before a RWL model can be saved.

Therefore we can safely say that the visual notation developed as part of this thesis has a zero or at least close to a zero degree of error-proneness when compared to writing RWL textually.

Also if a property of a given RWL construct is not entered and if that property is mandatory the meta-model will inform the user at validation time. We chose this approach as it reduces the error-proneness as opposed to providing them with default values.

Note that we are looking at syntactic and semantic errors in this cognitive dimension. Even though the end-user is not allowed to make syntactic and semantic errors they can still make logical and data errors. For e.g. the end-user can simply select the wrong database table or column or may even get the layout (done via functions) wrong. Therefore these errors will not be known till the end-user actually runs the generated RWL script against an actual database using the Prism WIN MIS system.

8.2.2.6 Hard Mental Operations

Indicates whether the user has to resort to pencilled annotations to keep track of what is happening.

While designing a RWL model using our visual notation the user needs to have an idea of what the eventual RWL script would contain. This could be defined as a hard mental operation. Moreover as the visual RWL model expands the user may need a process via which they can keep track of their current position.

Nested RWL constructs may also prove to be difficult to interpret by the end-user. Our notation represents nested structures with a line and a numbering scheme as mentioned in Section 6.4.1.6 to allow end-users to quickly visualize the order of such a nested structure. We also implemented an auto layout algorithm detailed in Section 6.4.1.6 which attempts to make nested structures easier to understand.

RWL constructs which represent a recurring notion may also be difficult to visualize using our notation. Items such as scans, for loops or while loops are represented using a simple compartment shape which may not depict its recurring nature correctly (may need something similar to Figure 57).

Other hard mental operation is remembering the RWL syntax. The user will need to have some idea of how the RWL sits together, although this can be learnt while using the visual notation. For e.g. if a particular shape cannot be dropped onto another shape the visual notation will not allow it. This can be seen in below in Figure 204.

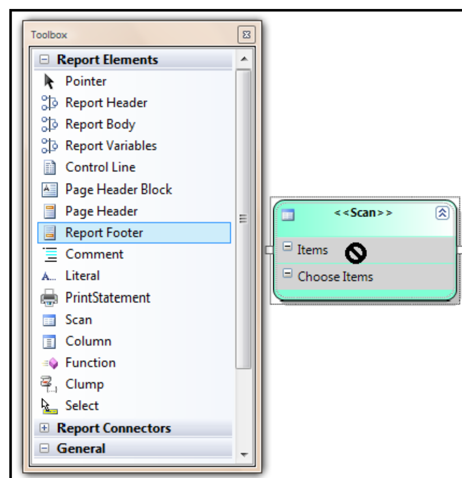


Figure 204: Illegal RWL drag and drop

In the above figure we can see that the user will not be allowed to drag and drop a *ReportFooter* model element onto a *Scan* model element (denoted by a "not allowed" icon).

8.2.2.7 Hidden Dependencies

This dimension measures how dependencies within our notation are indicated, whether they are overly represented and also whether their representation is perceptual or symbolic.

We show three primary relationships within our notation: Sequential flow, containment and ordering. We use a symbolic representation for these relationships. The sequential flow relationship is represented by a directed arrow. It is directed from one RWL construct to the next RWL construct. The containment relationship is represented by a line which joins the parent RWL

construct to its children constructs. The ordering relationship is represented by a number on the containment relationship. A lower number indicating that the RWL construct would occur before a sibling construct having a higher number. All three primary relationships are shown below in Figure 205.

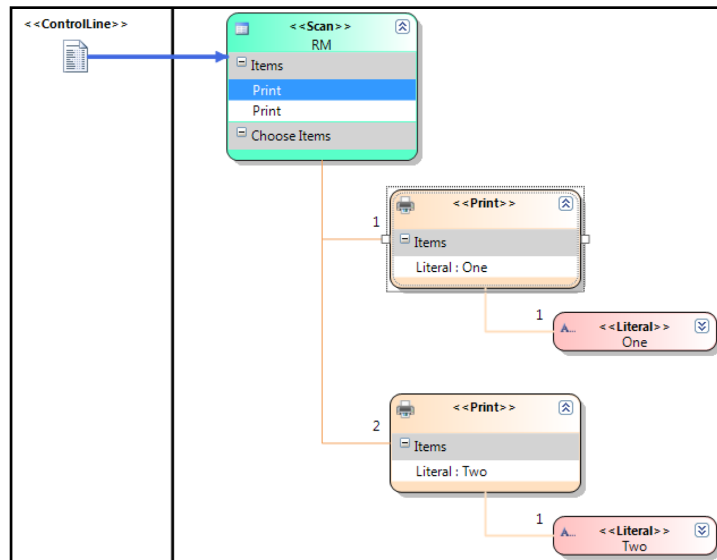


Figure 205: Dependencies within the RWL notation

In the above figure we see the sequential relationship between *ControlLine* and the *Scan*, the containment and the ordering relationship between the *Scan* and its two children (two *Print* statements).

Our notation also has two hidden dependencies: containment and instantiation/usage relationship. We looked at the containment relationship being represented using a symbolic notation in Figure 205, if we examine closer, the parent also contains a list of its children, this is essentially a hidden dependency. We attempt to make this relationship perceptual by selecting the child shape when the child item is selected within its parent. This is also shown in Figure 205, we selected the first *Print* item within the *Scan* and it has put a selection box around its corresponding shape.

The other hidden dependency occurs when we instantiate a RWL *Variable* construct and then use it. Figure 206 below shows a *Clump* variable item instantiated on the right and being using by the *Print* statement on the left. We make this relationship explicit when needed by putting a selection box around every variable usage shape when a particular variable instantiation shape is selected by the end-user, shown in Figure 206.

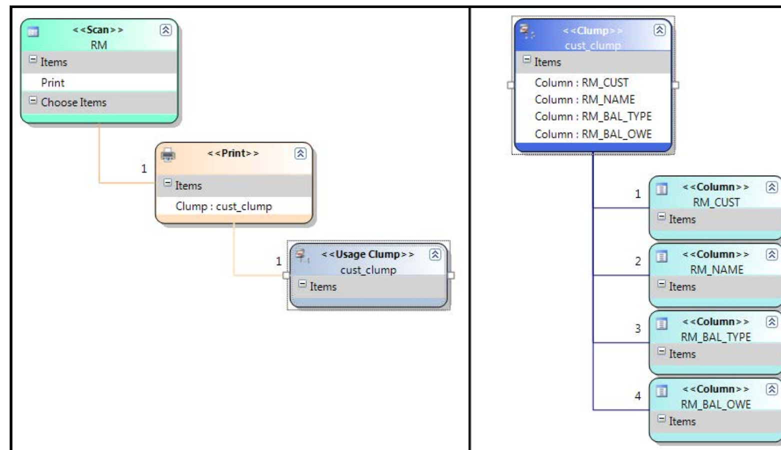


Figure 206: Variable instantiation/usage hidden dependency

8.2.2.8 Premature Commitment

This dimension indicates whether end-users have to make decisions before they have the information needed. As mentioned in Section 8.2.2.6, one of the hard mental operations is that user has to have some idea of what the eventual RWL script would look like before designing it visually. This can be argued to be a premature commitment made by user. Although as the user becomes familiar and experienced with the visual notation the degree of difficult creating new reports will decrease which will indirectly mean that the level of premature commitment required will decrease too.

8.2.2.9 Progressive Evaluation

This dimension indicates whether a partially created program (model) be executed by the user to obtain feedback on how they are doing. Our notation was developed using the Microsoft DSL Tools which allows us to validate partially completed models. If a model is valid the user can save it and successfully execute it if need be thus giving users the essential feedback required. Executing the model means generating the RWL script from a given RWL model.

The RWL script can be generated from a partially formed RWL model although this means that the partial model has no validation errors and is successfully saved. We also have to make it clear that the end-user only sees the RWL script and not the result of the execution of this script using the Prism WIN MIS system on a database.

8.2.2.10 Role-expressiveness

This dimension measures whether the user can see how each individual model component relates to the entire model. Within our notation each model element has its place within the RWL model which may either be directly involved via the sequential flow element (in the main flow of the report) or indirectly involved as a child of a model element (nested element). Therefore the end-user can clearly see how each model element relates to the entire model if need be by following the connectors coming in or going out.

8.2.2.11 Secondary Notation

This dimension indicates whether layout, colour, or other cues can be used to convey extra meaning above and beyond the official semantics of the language.

Comments can be added by end-users although these are part of the RWL meta-model and will be outputted in the generated RWL script therefore is not regarded as secondary notation. The visual notation can be altered so that we support annotations made by end-users to assist them follow the RWL model. These annotations will not be used in the generation scheme and will be ignored by the templating engine.

Layout can also be used to emphasize features and relationships. Currently we use an automatic layout algorithm which emphasizes the nested structure of the RWL constructs although other algorithms can be added to accentuate other features. User specific layout algorithms are defined as a future enhancement and are briefly covered in Section 9.5.3.

8.2.2.12 Viscosity

Measures the effort required to make a change. Our notation does not cater for explicit change management although it is presumed that because our notation is simple and only includes rectangular shapes and lines any change can be made with ease. We do offer some degree of refactoring capabilities, for e.g. we described the variable instantiation/usage relationship in Section 8.2.2.7, we offer end-users a simple way to rename the variable, if the variable name changes within the instantiation scope, that change will be propagated to every place where that variable is used.

Other changes such as dragging and dropping RWL constructs from one report section to another is currently not supported although because everything is done using the DSL Tools we can simply add constraints which regulate that if a RWL construct is moved from one report section to another so do all its nested elements and their nested elements ad infinitum.

It can be argued that small changes are hard to make using the visual notation as compared to doing them textually. This is because trivial changes such as adding a space between two print statements are cumbersome and involves multiple steps. The end-user also needs to make extra room in their visual model to cater for this which may prove to be difficult and time consuming. We do simplify this process to some extent by allowing the end-user to do an auto-layout after a change to order the visual models cleanly. This algorithm is not sophisticated and more layout algorithms are discussed as future work in Section 9.5.3.

8.2.2.13 Visibility

This dimension gives an indication whether every part of the code is simultaneously visible, if it is possible to compare two sections side by side and whether it is possible to know the order in which to read the code. Our notation caters for all of these possibilities. The entire RWL model can be easily seen at a glance assuming we have a large display. If a large display is not available the user can expand/collapse RWL constructs which allows them to view a larger area of the report or concentrate on a particular area of the report.

The order in which the code is meant to be read is apparent as it is indicated by the *Sequential Flow* relationship. If RWL constructs are nested within a parent construct then the order is indicated by our *Ordering* relationship using numbers therefore making it simple for user to read the report in a logical order.

8.3 Survey Evaluation

We designed a survey as part of the post implementation evaluation process. The survey was approved by the UAHPEC (Ref. 2008/405) and is included as Appendix J. We evaluate the RWM Shell Approach design and implementation as that was our primary solution. We approach the evaluation with the aid of a survey which is designed for the two stakeholders highlighted in Section 5.2, developers and report designers. One of the requirements of the survey is that the participant is a Prism employee, either a developer or a report designer as they will need to know some aspects of the RWL.

We surveyed 11 Prism employees, six RWL developers and five RWL users. RWL users were picked at random so we did not know their skill levels beforehand. We initially approached a potential participant with an advertisement informing them about our research (attached as Appendix G). Following this the participant was given the Participant Information Sheet (attached as Appendix H) which contained more detailed information about our research followed by a consent form (attached as Appendix I) which they were required to sign if they wanted to participate in the survey. Participants were then grouped in fours (developer and report designer groups were separate) and were given a guided demonstration of either how the meta-model works or the end-user interface. This demonstration can be viewed as brief tutorial or training programme. After the demonstration, the participants were requested to finish the survey to the best of their abilities.

The following sub-sections analyze the results of the survey with respect to the two stakeholders: Developers (evaluate the meta-model) and Report Designers (evaluate the end-user interface which instantiates the meta-model).

8.3.1 Developer

The developer survey basically evaluated the expressiveness of the RWL meta-model within the Microsoft DSL Tools. We asked developers some basic close-ended questions where we checked any prior knowledge they would have of the technologies used to implement the solution. We then asked them to complete two tasks which involved a minor and a moderate change to the RWL meta-model. This allowed us to practically determine how easy or difficult it would be for a new developer to modify the RWL meta-model to cater for changing requirements. Developers were then asked to comment on how they went about the tasks and also comment on any positives or negatives about the solution; this was done with the help of some open-ended questions. The results and suggestions of the developer survey are outlined in the following sub-sections.

8.3.1.1 Technological Preview

In this sub-section we look at how developers react to the new technology which includes the MDSD approach, the Microsoft DSL Tools, T4 templates and LINQ. We asked developers about any prior knowledge they had about these technologies and the results are shown in the table and graph below.

Table 5: Developers familiarity with solution technology

Technology	Technology Familiarity
MDSD	2
MS DSL	2
T4 Text Templates	1

LINQ/WPF	5

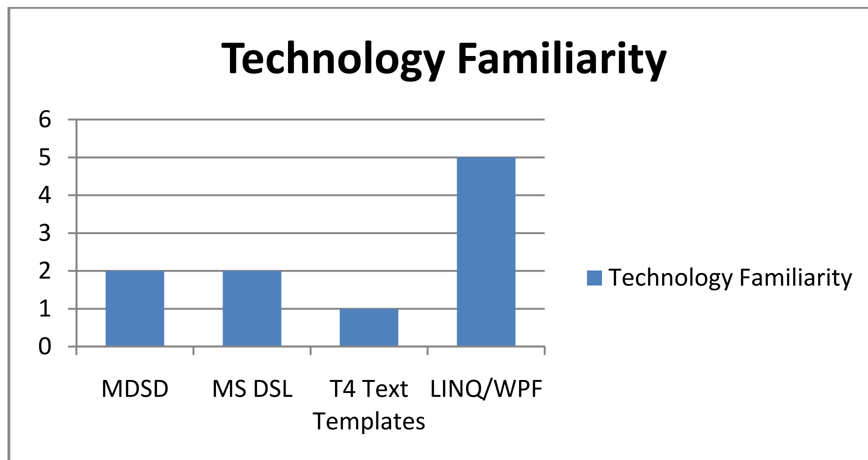


Figure 207: Graph of solution technology familiarity

It can be seen from the above graph that all Prism developers surveyed are familiar with LINQ/WPF although less than half were familiar with the other technologies which comprise the solution described in this thesis. This was the expected result therefore we hoped that the meta-model designer offered by the DSL Tools and our implementation offers a powerful but easy framework for developers to use and modify. This brings is to the next two questions in the survey.

We asked the participants to rate their proficiency with these and rate the simplicity of the meta-model developed using the DSL Tools after their guided demonstration (tutorial). The results are shown in the table and graph below.

Table 6: Developers proficiency with solution technology and meta-model perception rating after tutorial

Response	Technology knowledge	Meta-model simplicity
Expert/Easy	0	3
Can find my way around/More time needed	4	3
More time Needed/Not easy	2	0

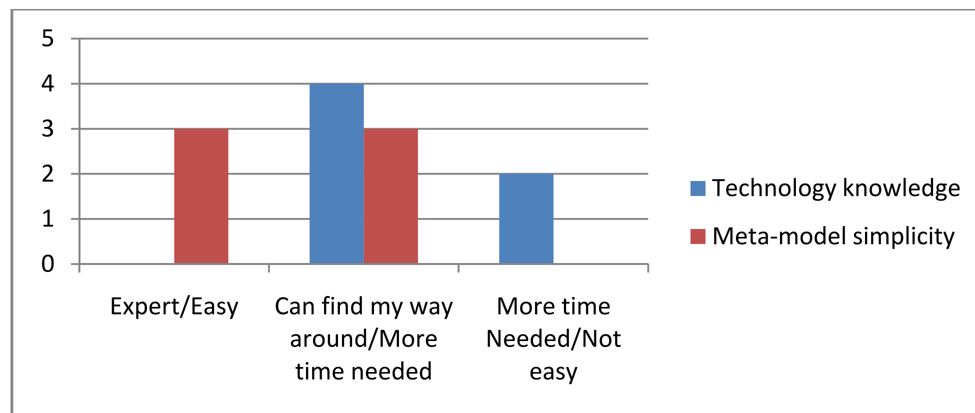


Figure 208: Graph of solution technology proficiency and meta-model perception rating after tutorial

It can be seen from the above graph that even though none of the developers felt completely confident in the technologies involved in the solution they did feel that the RWL meta-model developed using these technologies was simple and could be easily modified.

Developers were also asked to rate their proficiency with respect to the RWL semantics, this is shown below in Figure 209.

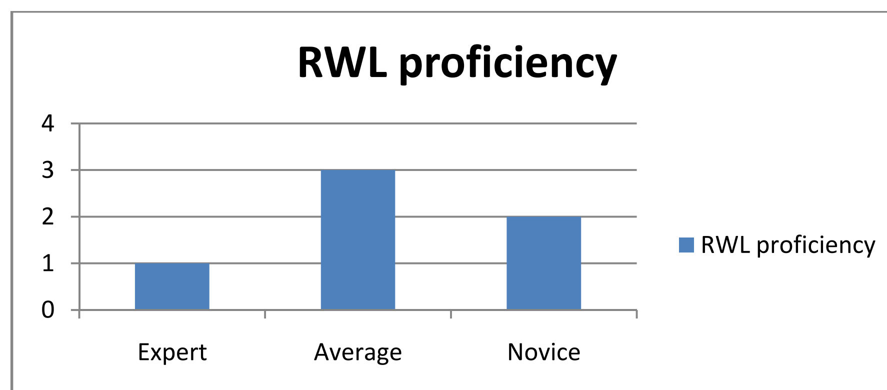


Figure 209: Developers proficiency with RWL

Note that from the above graph even though there is only one participant who is an expert in the RWL semantics, all developers (to some extent) perceived the RWL meta-model developed as part of the solution to be simple and straight forward (shown in Figure 208).

8.3.1.2 Task Effectiveness

This sub-section analyzes how the developer completed the two tasks given to them as part of the survey. The first task was comparatively easier than the second and the developers were only required to provide a step by step process on how to complete them, implementation was not mandatory.

Each task was divided up into two questions, the first required the developer to determine what the task required (analyze the requirements) in terms of the meta-model and the second required them to explain how they would use the DSL Tools and the meta-model to cater for those requirements.

All developers completed both tasks. Most developers found the first task (which asked them to add domain properties to an existing RWL model element) very easy. Developers initially did have

trouble finding the appropriate model element within the meta-model. All developers knew the basic steps in exposing these properties to the end-user and generating the required RWL. Actual implementation was not necessary as long they could note the general steps they would take to achieve this change.

The second task was comparatively harder than the first and it required a deeper change to the meta-model (adding a new RWL construct and attaching it to the rest of the meta-model). Most developers could easily go about making this change by looking at the existing meta-model and seeing how the RWL constructs were combined together. Some developers found it difficult to distinguish between the two relationships: embedding and reference although they could easily list the steps required to achieve this task which was all that was required.

Therefore, both tasks were completed by all the developers with reasonable ease. No implementation details were required as all we were trying to put across was the notion of model driven development and the idea of representing the RWL using the meta-model.

8.3.1.3 Feedback

This sub-section lists the feedback received from the developers in terms of the meta-model of the RWL and the solution it provides to the end-users. We list the feedback according to their nature and separate the positive from the negative. For the negative feedback we also attempt to outline a possible solution in the next sub-section where we analyze the feedback.

Positives:

1. Changing the meta-model was easy even for developers who had little to no experience with the RWL semantics.
2. Changing meta-model was easy as a developer could simply learn from the already implemented contructs and see how they work.
3. Changing the meta-model and the two tasks (part of the survey) helped understand the meta-model and how the various constructs were held together.
4. The meta-model allows the developers to expose the Prism RWL and the various parameters each construct may have to the end-user.
5. The meta-model helps end-users avoid making mistakes and also makes it easier for the developer to define constraints as opposed to defining constraints in an arbitrary text based form.
6. The framework and methodology can lend itself well to generate other code if need be such as Prism WIN MIS configuration scripts.
7. The explorer view (flat view) of the RWL model was thought to be quite useful by some developers as it exposed details to the end-user in a concise form allowing them to see the big picture.

Negatives:

1. Scalability
    a. Navigation will become harder as the model grows in size.
    b. With large RWL models, visual representation will be more complicated than textual representation.

2. Obvious learning curve involved with Microsoft DSL Tools.
    a. Inexperience with MDSD meant that the concept of generating (transforming templates) code was not always intuitive.
    b. Relationships were hard to visualize (which to make as a reference and which to make as an embedding).
    c. Editing templates to generate RWL script within the debugger was counter intuitive. Users would rather have templates within the source solution which get installed in the debugger.
3. Tasks would have been easier if done directly after the tutorial (guided demonstration).

The graph below shows a comparison between fixable and non-fixable items which were raised with the use of the negative feedback (listed above) gained from the developers. Fixable items are classified as items that can be rectified via design and implementation improvements and non-fixable items are those which may require the Microsoft DSL Tools be enhanced.

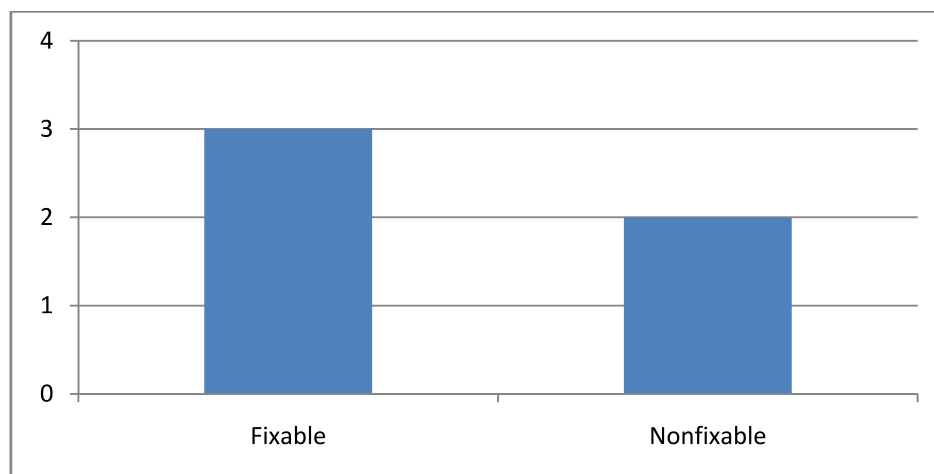


Figure 210: Fixable/Non-fixable meta-model issues

From the above graph we can see that three issues can be rectified via better design and implementation. These issues include readability of the meta-model and lowering the learning curve for developers using the DSL Tools.

8.3.1.4 Feedback Analysis

Scalability was the main concern with developers while doing the survey. The developers found it comparatively difficult to navigate the existing meta-model of the RWL due to its sheer size. A large screen area is ideal in this situation and intuitive organisation of the meta-model would prove to aid navigation. The Microsoft DSL Tools also made it difficult to add new domain model classes and relationships due to the fact that everytime you added a new construct it would append it to the bottom of the diagram and change the scroll and zoom position. These problems are embedded within the DSL framework provided by Microsoft and we hope they get resolved as the technology matures. Another solution offered by one of the participants was the use of wizards to add new RWL constructs to the meta-model. This was an excellent suggestion because adding a new RWL construct follows five basic steps:

1. Add domain-model class representing construct
2. Add properties to it

3. Form relationships
4. Add shape representing construct
5. Expose it via the toolbox

If the above mentioned steps could be encapsulated in a wizard like approach it would make expanding the RWL meta-model comparatively easy. This wizard could be provided in the form of a Visual Studio add-in, another DSL or an integrated UI based wizard within the existing DSL.

Developers also expressed scalability issues in the end-users' report designer. Because the developer survey is only concerned with the development side of the solution, i.e. the meta-model we will omit this issue and look at it in the next section where we analyze the results of the report designer survey.

Another issue expressed by the developers was regarding the learning curve of the Microsoft DSL Tools. This was an expected issue although there were three main concerns with this learning curve. The idea of code generation was not easily understood by some developers. I suspect this was due to the reason that such an approach was not tried before and the idea of executing code generated from a visual model is foreign. Although developers did quickly learn the idea of how the code was generated and what the execution of such code meant. The other two aspects which seemed to steepen the learning curve were domain relationships and text templates. In my opinion knowing which domain relationship should be a reference and which one should be an embedded one is purely subjective and developers (including myself) will improve at this with experience.

As far as text templates are concerned they are similar to any programming language which the developer has to get accustomed to. Although they are much simpler to use than traditional programming languages as all we are aiming for is to generate text from it (RWL script). The idea of text templates being part of the debugging solution is a Microsoft standard and again we hope that they provide an easier and more sophisticated way for developers to embed these templates using the source solution as opposed to using the debugger.

8.3.2 Report Designer

The survey done by end-users (report designers) was intented to give us an idea of whether the toold provided assistance to novice and intermediate users to design Prism reports. We were also looking for whether the visual designer has advantages over textually writing a Prism report. Usability, visual notation, program flow and RWL script generation were the core items being evaluated by this survey.

Report designers were asked close ended questions to determine how they currently design Prism reports and the applications they use to achieve this task. Questions were then aimed at the newly developed reporting tool and we asked questions to determe the learning curve by simply asking them to rate their profieciency after the tutorial (guided demonstration). They were also requested to list five items they would like to have in such a reporting tool or an IDE. These questions were then followed by asking them to complete two tasks where they design Prism reports. The first report was comparatively easier than the second. The survey was concluded by asking the report designers for their suggestions on the positives and negatives of the new tool. The results are outlined and analyzed in the following sub-sections.

8.3.2.1 Tool Preview

This sub-section analyzes what end-users currently use to design Prism reports and whether they are satisfied with the level of functionality it has. We also ask them to list five aspects of functionality (according to preference) they would like to see in a Prism reporting tool which would aid them and allow them to design Prism reports with ease.

The graphs below in Figure 211 and Figure 212 show what the end-users currently use to design Prism reports and whether the IDE provides the features needed to design Prism reports respectively.

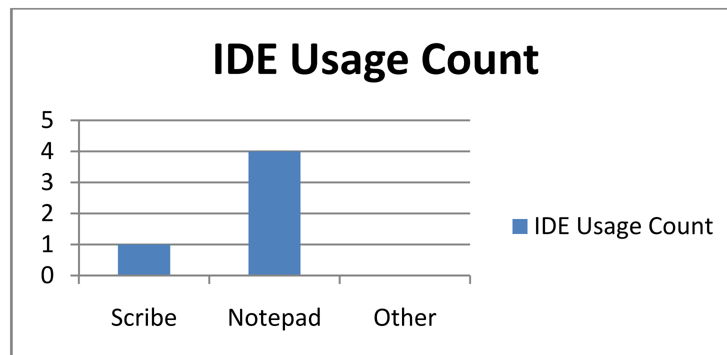


Figure 211: Current report design (IDE) tool usage

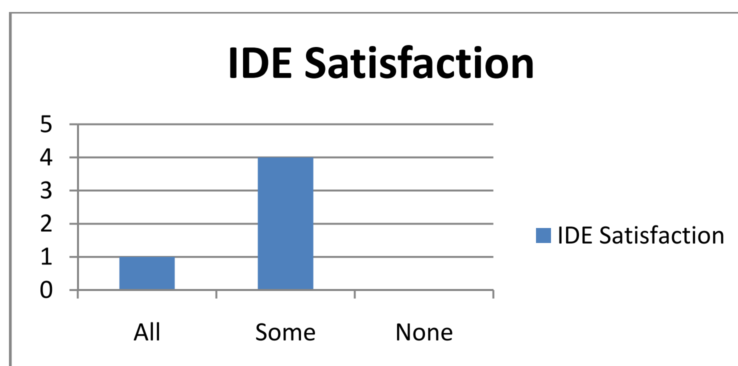


Figure 212: Current report design (IDE) tool satisfaction

It can be seen from the above graphs that most end-users do not use Scribe as a dedicated IDE for Prism report design. I presume this is the case because Scribe does not provide any extra functionality over a simple text editor like Notepad. Moreover there is a noticeable gap between the IDE the users use and the functionality which they require to be present in an IDE. We also asked them to list five features (in order of need) which they would like to see in an IDE to assist Prism report design. These are combined across the participants and are listed below in an ordered list according to need.

1. Wizards or templates to ease the initial process of report design.
2. Dynamic validation of the report.
3. Ability to run the report straight from the IDE and see the resulting report.
4. Database helpers. E.g. automatic joins, table and column selector, key selectors.
5. Intuitive UI to streamline the report writing process. Maybe via a set workflow template.

From the above list the new report writer tool provides dynamic validation (2), database helpers (3) and an intuitive UI (5). The other requirements have been identified as future work and are further elaborated upon in Section 9.5.

We followed on from the above questions and gave the end-users a brief guided demonstration (tutorial) using the new report writing tool. We then asked them whether they had a clear idea on how to use this new tool to attempt and measure the potential learning curve it may have on end-users. Their responses are captured below in the graph:

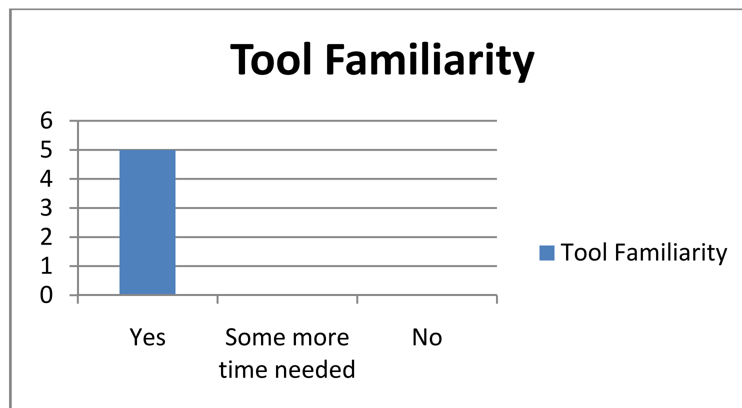


Figure 213: End-users familiarity with the new tool after guided demonstration

From the above graph we can clearly see that the end-users had a clear idea about what the new reporting writing tool was attempting to achieve and how a new user would go about using it to design a new Prism report.

In the above questions we determined the tool currently used for Prism report writing and whether the end-users were satifisfied with the features the IDE provided. We then gave the end-users a guided demonstration and determined that they had a clear understanding of what this tool was attempting to achieve. We then gave the end-users two task to complete to determine the usability of this new tool. The results are analyzed in the next sub-section.

8.3.2.2 Task Effectiveness

This sub-section analyzes the usage of new reporting tool designed and implemented as part of this thesis. This is done by asking end-users to design two reports using the new tool. The requirements for the first report are comparatively simpler than the second. The survey requests the end-users to list the steps they would carry out to design such a report and output the corresponding RWL script using the text templates already provided. We then asked them to identify whether our new reporting tool is designed and implemented in a way that would aid these steps and if not how would we make the interface better.

All participants could complete both tasks using the new report writing tool designed as part of this thesis. Some participants had forgotten details about the two different connectors: the sequential connector and the nested connector which made the task a little difficult. This difficulty could have been avoided if the participants attempted the survey immediately after the guided demonstration (tutorial) although due to time constraints this was not feasible for some participants due to prior work commitments at Prism.

The steps noted in order to achieve each task differed for each participant as the report to meet the requirements for each task could be designed in several ways. Most participants noted that they followed the validation errors given by the meta-model to achieve a valid report which helped them a lot. Database helpers were also used extensively to give users assistance regarding the tables and columns which existed within the Prism database. Most participants found the Visual Studio Shell environment overwhelming and had difficulty to use the property and solution editor. This issue can be easily resolved by customizing the Visual Studio Shell hiding the items (menus/buttons) which are irrelevant with respect to the report writing DSL.

After the tasks were completed the participants were requested to give us feedback in the form of comments and were also requested to list the positive and negative aspects of the approach. Their feedback is detailed in the following sub-section. The end-users were also requested to indicate whether visually designing Prism reports is better than writing them textually and whether the visual DSL assisted them in terms of the Prism database. Their response is encapsulated in the graph below:

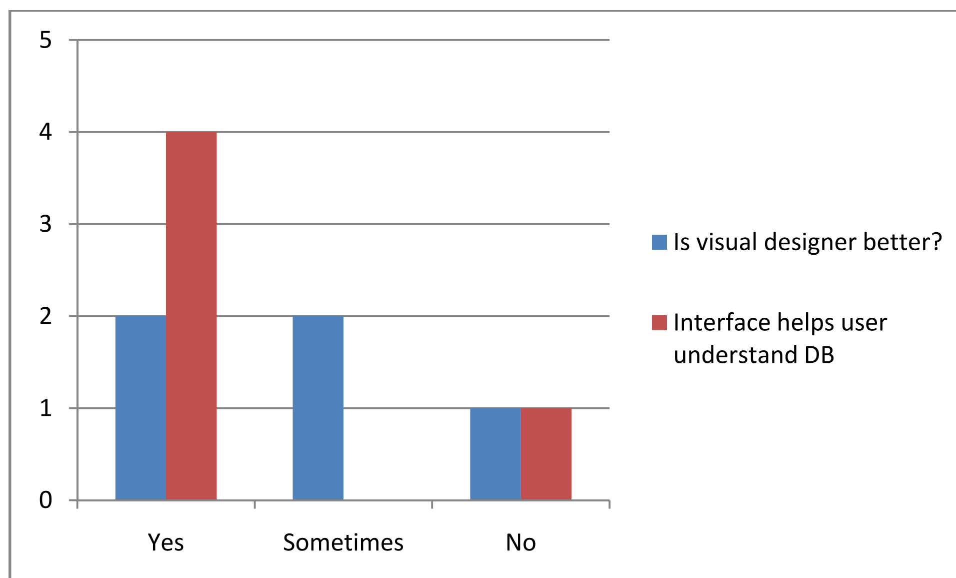


Figure 214: End-user rating visual notation compared to textual representation and its help in terms of the Prism DB

From the above graph we can clearly see that most participants found the visual DSL more useful while designing Prism reports when compared to writing them textually. We can also see that most participants (except one) found the DSL useful in terms of the Prism database. We see an equal amount of participants who feel that the visual notation to design a Prism report is only “sometimes” better than textually writing it, this arises due to the scalability issue of visual languages. Visually a large RWL model may be more difficult to interpret than its textual counterpart therefore some end-users feel that a visual representation is only “sometimes” better.

8.3.2.3 Feedback

This sub-section lists the feedback received from the report designers (end-users) in terms of the Visual Studio Shell environment which allows them to visually design Prism reports. We list the feedback according to their nature and separate the positive from the negative. For the negative feedback we also attempt to outline a possible solution in the next sub-section where we analyze the feedback.

Positives:

1. Design time validation is extremely helpful while designing Prism reports. Saves time and effort compared to saving the file, importing it into the Prism WIN MIS system and fixing the errors.
2. Database helpers which exist within the current tool helped determining the current field or table to use. They prevented the end-user from making spelling mistakes and allowed them to get a clear overview about the database and its table without understanding intimate details of its structure.
3. Visually designing a Prism report allows end-users to clearly visualize a given report script and its logical flow. The visual notation also gives the end-users the ability to design a Prism report without worrying about RWL semantics and syntax spelling.

Negatives:

1. The visual notation lends itself well to design small to medium size reports although as the report size increases it becomes difficult to visually represent it as opposed to textually.
2. Trivial changes are hard to make using the visual notation. A small change such as adding a space in a print statement requires a number of steps as opposed to doing it textually.
3. There is no synchronized view between the visual notation and the generated textual RWL script. A synchronized view would allow end-users to modify the visual model of the RWL and see the changes reflect in the generated textual RWL script making it easy to the visual model and lower the learning curve gradient.
4. The Visual Studio Shell (Experimental Hive) via which the visual notation is exposed is hard to use for non-developers. The shell contains many sub-windows, toolbars and menu items most of which are irrelevant to the exposed DSL due to this reason the UI is overwhelming for non-developers.
5. The visual notation is primarily designed for novice and intermediate report designers as it does not allow a high degree of customization when compared to textually designing a report. Due to this reason expert report designers will be constrained by the features of the tool and may hamper their production.

Categorize feedback

- UI improvements
- Notation improvement
- Code generation improvement
- Database helper improvement

8.3.2.4 Feedback Analysis

Scalability was identified as the common issue all report designers experienced. As a RWL model grows it will become more complicated and therefore harder to represent visually. This issue can be resolved (not entirely solved) by allowing the visual notation to collapse/expand nested children, hide/show parts of the model and use screen real estate wisely using sophisticated layout algorithms. Collapsing/expanding of nested children is already implemented and implementing sophisticated layout algorithms is identified as future work and further detailed in Section 9.5.3.

The visual notation allows novice and intermediate users to quickly design Prism reports easily using the dynamic validation engine and the database helpers. Although this makes the notation hard to use for expert users as they might not achieve all the flexibility they require. Moreover trivial changes in the RWL model would be simple to achieve textually as opposed to visually which may require a number of steps. This can be made easier by implementing wizards which allow end-users to carry out certain tasks. Wizards are identified as future work and are discussed further in Section 9.5.7.

Most participants found the environment (Visual Studio Shell/Experimental Hive) via which the DSL was exposed to be complicated and confusing. This was expected as we simply asked the participants to use the Visual Studio Shell to design Prism reports without any customizations. This problem can be solved by making the Visual Studio Shell thinner, meaning removing any extra toolbars, sub-windows and menu items which are not relevant to our DSL. This would make using the Visual Studio Shell much easier thus improving the usability of the visual notation.

Some participants found it cumbersome to regenerate the RWL script after small changes to the RWL model to determine how the language worked. Due to this fact some participants recommended that the RWL visual model and the generated textual script be synchronized to some degree. Currently this is a manual process as the end-user can run the templates after the visual model is saved and validated. Although it can be automated to be triggered when the visual model is saved thus keeping the textual representation synchronized with the visual model.

8.4 Cognitive Dimensions V.s. Survey Evaluation

In this chapter we looked at two different types of evaluations. A qualitative and subjective (authors perspective) approach using cognitive dimensions and a quantitative approach using developers and end-users using surveys. In this section we determine whether the results from these two evaluation techniques converge or diverge. Convergence would indicate that the user-model is similar to the developer-model and divergence would indicate that work needs to be done to bring the user and the developer model close together. The table below analyzes each cognitive dimension and indicates whether it converges or diverges (or cannot be measured) with the evaluation and gives a reason why.

Table 7: Cognitive dimensions convergence table

	Cognitive Dimension (CD)	Converge/ Diverge/Not Applicable	Why
1	Abstraction Gradient	Converge	End-users were quickly able to adapt themselves to the different levels of abstraction within the visual notation. This can be seen from the surveys as all end-users could complete the two report design tasks with relative ease.
2	Closeness of Mapping	Converge	Our CD analysis showed that even though the visual notation is not close to domain we are mapping its consistency and use of icons enables end-users to determine their representation with ease. This can be seen from the completion of the two tasks given to the participants.
3	Consistency	Converge	Our CD analysis showed that the visual notation was consistent. Each shape had a consistent look and feel to it

			and relationships between shapes can be created consistently. This is evident as both report designing tasks were completed successfully by end-users after the guided demonstration even though the guided demonstration only showcased some aspect of the notation.
4	Diffuseness	Converge	We determined that our notation is less diffused than its textual counterpart as end-users only need to care about shapes as opposed to syntax details. This dimension converged with our evaluation outcome as most participants perceived the visually design RWL model to be concise, clear and easy to understand when compared with the textual representation.
5	Error-proneness	Converge	The visual notation exposed by our DSL abstracts away semantic information of the RWL from the end-user. This allows the end-users to create Prism reports without worrying about spelling mistakes, semantic errors and syntax errors. We established that this converges with the results from the evaluation as the steps highlighted by each participant to meet the reporting requirements highlighted in the survey did not have any errors. All participants could successfully complete both tasks creating valid Prism reports.
6	Hard Mental Operations	Diverge	During our CD analysis we established that visualizing nested and recurring (i.e. Scan) structures will be difficult for the users. We also established that the end-user will need to have some idea of what the end RWL script should look like before using our tool which may be a hard mental operation. Although when the end-users completed the tasks outlined in the survey it was determined that our CD analysis diverged from the results from the survey. A possible reason for this could be that the survey did not require the end-user to generate a complex Prism report.
7	Hidden Dependencies	Converge	We analysed this dimension and came to the conclusion that each dependcy and relationship within the domain are represented with a simple shape or connector which keeps the notation simple. This was reflected in the survey as the end-users could use the notation easily and complete the two report designing tasks given to them.
8	Premature Commitment	Diverge	Our CD analysis showed us that the visual notation could get complex and may require some level of premature commitment from end-users for hard mental operations. Although the survey demonstrated that end-users could easily complete the two tasks without commiting to a concrete report design or structure, thus we established that our CD analysis diverge from the results of the evaluation.
9	Progressive Evaluation	Converge	We determined that our visual DSL allows the end-user to progressively evaluate their RWL model allowing them to visualize the generated RWL script as the model is changed. This converged with the results from the survey as each end-user progressively generated the RWL script to determine their model validity and correctness.
10	Role-	Converge	Our CD analysis showed that each RWL construct within our

	expressiveness		visual notation can be easily tracked with the help of connectors representing relationships. This converges with the results from the survey as the end-users could easily complete task two in the survey which involved an enhancement to task one. This indicates that the end-user could easily see visualize each RWL model element correctly and determine its place in the whole model allowing them to modify the required elements to complete the task.
11	Secondary Notation	Diverge	Our CD analysis exposed that our visual notation currently has no mechanism to allow end-users to add superficial annotations to the diagram to help them. The RWL meta-model does expose a *Comment* construct but that is part of the RWL and cannot be regarded as secondary-notation. We exposed automatic layout algorithms to assist users and reinforce the nested relationship. Although we say that the results from our evaluation diverge from the CD analysis as end-users did not use this automatic layout capability to its full use. An explicit mechanism to annotate the visual RWL model would have proved helpful in retrospection.
12	Viscosity	Diverge	Our CD analysis demonstrated that the visual notation developed as part of the DSL was viscous to easily allow accommodate end-user changes. This is because we have a simple visual notation comprising of simple shapes and arrows. Although our evaluation showed that end-users may find it difficult to make small changes to the RWL model using our visual notation when compared to making the same change in a textual environment. Due to this reason we determined that our CD analysis diverge from the results from the evaluation.
13	Visibility	Diverge	Our CD analysis involved determining the visibility of each element and whether the notation could represent the sequential flow of the RWL efficiently. Even though the end-user could follow the flow of the RWL model we believe that the CD analysis diverged from the results of the survey as we did not cater for the visibility of the generated RWL script. Feedback received from a participant indicated that it would have been helpful if there was a synchronized view between the visual RWL model and the text it generated.

8.5 Comparison to Scribe

We looked at Scribe in Section 2.6 and highlighted some of its features and limitations. In this section we compare our solution with Scribe as it is the primary IDE used to develop Prism reports. We developed the table below which highlights some of the key differences between Scribe and the solution developed as part of this thesis.

Table 8: Scribe vs. Our Solution

Feature	Scribe	This solution
Notation	Textual	Visual
Wizards	Common reports	-
Data Dictionary	Static	Dynamic

Error checking	-	Dynamic validation
Help	Non-context sensitive	Context sensitive
Database assistance	-	Joins and limited value matches
Modular	Reporting sections (Tabs)	Reporting sections (Swimlanes)
Code Folding	-	Show/Hide children
Serialization	Text (Binary)	XML (Schema based)
Code Libraries (Snippets)	Common code snippets	-
Technology	VB6	C#, DSL

The following sub-sections describe each aspect listed in the above table. We can see that the solution developed as part of solution supersedes Scribe although it does lack in some aspects which we address as future work.

8.5.1 Notation

Scribe relies on the end-user typing the entire report out. This tends to be tedious, error prone and requires the end-user to have in-depth knowledge of the RWL semantics and not to mention the Prism database.

Our solution improved this limitation and allowed end-user to visually design Prism reports. This meant that the end-user did not need to learn RWL or its semantics as they could simply use the toolbox to drag and drop shapes representing RWL constructs and join them using connectors. The shapes could then be linked to database objects such as tables and columns to make the report complete.

8.5.2 Wizards

Scribe provides end-users with a wizard approach to commonly used reports. This is shown in Figure 38. Wizards allow the end-user to start with a piece of skeleton code which they can modify to suit their needs thus saving time and effort.

Our solution in its current state does not provide wizards although is considered as a future enhancement and detailed in Section 9.5.7.

8.5.3 Data Dictionary

Scribe exposes a static Prism meta-data dictionary which end-users can use to find more information about Prism database tables, columns and order keys. This is shown in Figure 40. Therefore, if new Prism database tables, columns or order keys are added this data dictionary needs to be updated.

The solution described in this process exposes a dynamic data dictionary which links in with a live database making this solution far more versatile than exposing it statically. The downside of having a dynamic data dictionary means that database access will hamper performance. There are various solutions to this problem like caching the Prism database meta-data for a defined interval or improving data access code.

8.5.4 Error Checking

Scribe provides no real error checking mechanism for end-users while designing reports. The only time a report can be checked for errors is when the end-user imports the report into the Prism WIN MIS system. This means that errors within the report will not be known to the end-user until “run time”.

Our solution improves this by allowing "run time" error checking and validation. Because our solution is built on the RWL meta-model dictating its semantics it is impossible for the end-user to create an erroneous RWL model. Moreover every time the end-user saves the RWL model a validation check is done to ensure the validity of the RWL model. A validation check can also be performed at any point in time by the end-user via a menu item.

8.5.5 Help

Scribe provides help in via its syntax completion. Shown in Figure 43, syntax completion does help to an extent although there is no filtering mechanism provided. Therefore the functionality which was designed to help eventually ends up hampering usability. This is described further in Section 2.6.2.2.

Our solution improves on this and provides context specific help which does not overwhelm the end-user by providing them with superfluous information. Our solution provides context specific help in the form of custom editors, drop down boxes and menu items.

8.5.6 Database Assistance

Scribe provides no database assistance as such and the only assistance it provides in terms of the database is the data dictionary which was described in Section 8.5.3. The solution described in this thesis provides innate database assistance in the form of *Choose* statements (which join two nested *Scan* constructs together) and limited value helper for *Choose* statements (if a column contains a set of enumerated values, we show these enumerated values). Both examples are given in our first case study as part of Section 7.2.

8.5.7 Modular

Scribe provides us with a tab based approach to reporting which is shown in Figure 39. This makes the report modular and allows the end-user to concentrate on any particular section of the report.

We borrowed this approach from Scribe and made our solution modular by dividing the report up into three sections: Header, Body and Variables. More reporting sections can be easily added via the meta-model if need be.

8.5.8 Code Folding

Code folding is a mechanism provided by modern IDEs via which code blocks can be collapsed to save screen real estate and allow end-users to view a larger section of a program or a model.

Scribe does not have any support for code folding although the solution described in this thesis allows for such a requirement. Our notation allows the end-user to collapse and expand a parent RWL construct which hides or shows its nested children respectively. This allows the end-user to concentrate on a particular part of the report. Moreover if the RWL model spans a larger area, collapsing a part of the model would allow other sections of the model to fit easily in the given screen area.

8.5.9 Serialization

Scribe saves the Prism report file created by the end-user in a proprietary binary format on the disk. This format does not lend itself to any kind of external manipulation. Although scribe does offer limited capability to the end-user to import simple text files and convert them to a Prism report.

The RWL model created using the solution developed as part of this thesis saves the file in a XML format which has a standard schema association. This allows other 3rd party tools to easily read in the file, manipulate it, transform it and save it back. Saving a file in XML also means that it is easily extensible and lends itself well to model transformation techniques such as XSLT transformations.

8.5.10 Code Libraries (Snippets)

Scribe offers a set of common code snippets as shown in Figure 42. This allows the end-user to simply copy paste code which suits their needs. These snippets are not customizable and there is no mechanism for end-users to create their own snippets.

Although our solution does not provide any mechanism for code snippets it is considered as a future requirement and is further explained in Section 9.5.8.

8.5.11 Technology

Scribe is written in VB6 and often requires highly experienced developers to make small simple changes. VB6 is also no longer supported by Microsoft (Microsoft, 2008). Moreover Scribe is not a modern application and has trouble running on Microsoft's latest operating system, Vista.

The solution designed as part of this thesis was entirely written in C# using the DSL Tools making it extensible and maintainable. The DSL Tools allow developers to make changes and simply regenerate the code which would incorporate those changes into the end-user interface and allow end-users to use these changes reducing the time to market for new features. Using DSL Tools also allows us to create software with comparatively less bugs (as all code is automatically generated), improve code quality and allow novice developers to make changes.

8.6 Summary

This chapter detailed the evaluation of the solution developed to allow end-users to visually design Prism reports. It also evaluated the expressiveness of the meta-model by surveying developers to determine how easy a change can be brought about.

Evaluation was done in two phases, during design phase using cognitive dimensions using champagne prototypes and during a post implementation period using surveys. Using cognitive dimensions we determined that our solution met with most of the requirements needed for a user-friendly graphical DSL which helped end-users achieve a task, in this case designing Prism reports. The results from the surveys done during the post implementation phase converged with most of the cognitive dimensions analysis we did. This can be seen from the participants (developers and report designers) completing the tasks outlined in the survey with ease. The main concern of the participants in terms of the meta-model and the reporting tool was scalability. Whether the meta-model is able to sustain a growing number of RWL constructs and whether the visual reporting tool is able to represent large RWL models were the core issues.

The chapter concluded with an empirical comparison of our solution to Scribe, the current IDE used to design Prism reports. We noted that our solution was better in some aspects when compared to Scribe although to make the solution a well rounded product which could be used by Prism we would have to implement other features which are outlined as future work.

Chapter 9 - Conclusions and Future Work

9.1 Introduction

This chapter concludes this thesis by highlighting the contributions it has made to Prism Software in terms of solving its problems of database and the RWL complexity. It highlights some of the evaluation results and also critics itself by exposing some current and potential issues. The chapter concludes by highlighting some of the future work that can be carried out with this thesis as its base.

9.2 Thesis Contributions

The contributions of the thesis are highlighted in this sub-section with respect to each stakeholder. Two core stakeholders were identified in Chapter 5, namely Prism as a company and Prism customers (end-users).

9.2.1 Prism Software

This thesis provides an empirical study into how MDSD can assist Prism in its long term goals to solve its core problem of database complexity. The thesis highlights some of the approaches Prism as a company can take to lower production cost, increase productivity and develop a framework which be centralized and used by all of its products. We summary these approaches in the following sub-sections.

9.2.1.1 MDSD methodology

The thesis highlights some of the core properties of MDSD and how it can be advantageous to an enterprise if used correctly. We showed this in this thesis by developing a rapid prototype of an application which would have usually taken a lot of resources in terms of developers and time. The thesis also gave an in-depth account of some of the cutting edge technology which can be utilized to achieve a model-centric environment. It showed how the Microsoft DSL Tools can provide all the required resources for Prism to design robust models of any system and expose these to developers and end-users via the Microsoft VS Shell.

Other technologies were also introduced which would allow Prism to utilize the full power of MDSD. These technologies were model transformations, software factories and Metamodeling.

9.2.1.2 Modeling framework

The thesis laid the groundwork for Prism software to design further models which could be utilized by other applications besides the prototype described. The thesis introduced and elaborated on the core ideas and concepts built within the Microsoft DSL Tools to allow Prism as a company to invest its resources into it to gain a substantial edge in today's software market.

We also introduced the T4 templating engine shipped with the Microsoft DSL Tools which would expose any model and allow developers to potentially output any text file built around this model. This would assist Prism in numerous ways ranging from trivial configuration scripts to enterprise level SQL upgrade scripts.

The modeling framework would assist Prism staff in developing models even if they do not ultimately get used in end-user applications. Developers could generate models which would eventually be consumed by business analyst. Text templates could be designed around these models which would output information required by business analyst. Some examples of outputs

could be a text file showing a developers progress through a particular programming task. This flow can be expanded further, where business analyst can design models which can be consumed by the management team.

Therefore, in short the modeling framework has given Prism the ability to abstract any enterprise level task, such as report writing (shown in this thesis) and ultimately allow respective users to work in a domain they are most familiar with.

9.2.2 Prism Customers (End-Users)

The thesis provides Prism customers an application which encapsulates a specific model developed as part of this thesis. This application is described as the RWL shell host and is currently in a prototype stage.

The thesis developed a prototype which allows end-users to visually design reports. The prototype was built upon the modeling framework described in the previous sub-section and gave a user the ability to design RWL models which would output the required RWL script. The prototype developed introduced Prism to the Microsoft Visual Studio Shell which potentially allows any modeling language to be hosted within it and also exposes the core functionality surrounding any sophisticated IDE such as save, open, undo, redo, etc.

The prototype also exposed the meta-model of the RWL and encapsulated all its semantics using constraints. The prototype proved the ability of the modeling framework to provide a method via which a particular enterprise task, such as report writing, can be abstracted from the end-user. The prototype allowed novice end-users to quickly develop Prism reports without the steep learning curve which hampered their progress prior to the model-centric approach.

The prototype also provided essential context sensitive help to the users in terms of the Prism database which potentially meant that the end-users did not need to have intimate knowledge of the its structure to extract information from it.

Therefore, in short the RWL shell host provided end-users to design Prism reports working on an abstraction level which exposed domain concepts they are most familiar with. The domain in this respect was the actual Report Writer Language and some Prism database concepts such as views and relationships.

9.3 Conclusions and results

We have shown that an enterprise task or process, if structured in a consistent way, can be abstracted using models.

A complicated task such as writing Prism Reports was made comparatively simpler by designing a meta-model on top and then exposing this meta-model to the end-user via a user interface. This allowed us to hide details such as semantic constraints of the language, complicated database tables and views and their corresponding relationships.

We also proved that given a meta-model, if changes are made in the enterprise process or task, these changes by quickly replicated in the meta-model and released to the end-user thus reducing the time-to-market of a product, increasing productivity, improving software quality and reducing the number of problems.

The above mentioned conclusions were reinforced by the developer and end-user surveys carried out as part of the evaluation of the solution. We determined that both the meta-model and the visual notation that exposed the meta-model were easy to use for developers and end-users respectively. Changes in the meta-model such as implementing new RWL constructs and changing existing ones were straightforward and simple. The evaluations also led us to the conclusion that it is comparatively simpler to visually design a small to medium sized report than writing it textually making it easier for novice users who are new to the Prism RWL. The main issue arising from the survey was scalability. Both, the meta-model and the visual notation were speculated to become very complex and large as the solution (meta-model) grew and thus enabled large Prism reports to be designed. Meta-model scalability can be solved by making the meta-model modular and cohesive and the visual notation can be designed for scalability by allowing end-users to concentrate on sections of the report by allowing them to collapse/expand nested constructs, hide certain constructs and search the visual RWL model. Note that collapsing/expanding of parent constructs is already implemented in the solution although improvements can be made to it.

9.4 Current Limitations

Limitations mentioned here are only relevant to our RWM Shell Approach. We abandoned the Class Diagram Approach due to the time constraints imposed.

9.4.1 Meta-Model

The major limitation while designing the meta-model was the lack of representing interfaces. This made it essentially hard to design a hierarchy of common RWL constructs as we were only allowed to deal with abstract classes and thus could only extend (inherit) from one parent class. Even though this limitation was overcome the result was more complicated than a possible solution with interfaces. The lack of variety of shapes to represent model elements is also a limitation. Also some simple customizations like changing the display font style do not currently exist within the DSL Tools.

Another obvious limitation is that not all of the RWL constructs are currently implemented in the meta-model and also that the meta-model offers assistance to novice and intermediate report designers. Expert report designers may feel slightly constrained by how the meta-model works.

9.4.2 RWM Shell Host

The obvious limitation of the RWM Shell Host is the fact that it is essentially a Visual Studio Environment. This makes it hard to customize and end-users that are not familiar with the Visual Studio environment will have to go through a learning curve.

Another limitation is related to the customization issue, adding menu items and other commands are made complicated via the VSCT which is described in Section 6.4.1.10. Note that although learning VSCT is not difficult, it does involve a learning curve.

9.5 Future Work/Enhancements

9.5.1 Expand and Refine the Meta-Model

The RWL meta-model developed as part of this thesis only encapsulates part of the RWL therefore we can invest more time in the future to add more RWL constructs and make the meta-model complete. We could also refine the meta-model and make it semantically richer by improving Prism

database meta-data access, allowing the end-user to extract and view detailed information about each construct within the model.

9.5.2 Improve Visual Notation

The current visual notation as it stands uses most of the built-in shapes provided by the Microsoft DSL Tools. This can be drastically improved by allowing a graphical or a user interface designer to aid us in finding the right balance between the end-user mental model and the notation. Icons representing each of the RWL constructs can also be further elaborated and improved allowing a richer end-user experience.

9.5.3 Improve Model Layout Algorithms

The RWL shell currently only allows to layout children of a particular RWL construct according to the order in which it occurs; this can be further expanded by adding richer layout algorithms. Some examples of layout algorithms include:

- *Auto layout algorithm:* Spaces all model elements accordingly and fits them within a given area
- *Tree-based layout algorithm:* Orders all model elements in a tree like structure
- *Program-flow based algorithm:* Places each construct depending on where they occur in the RWL script
- *User defined algorithm:* User can add handlers which allow them to completely determine the placement of each model element on the canvas

9.5.4 Versioning

If the meta-model representing the RWL changes in case the developer modifies or adds new information there is a potential that existing RWL models would not correctly work. This is due to the fact that the older version of the RWL models was designed against a different meta-model (schema) which has not undergone change. This flaw needs to be rectified in the future as the meta-model will change as often as need be to incorporate new RWL constructs.

We envisaged a solution for this problem by using a transformation service as shown in Figure 54. As explained in Section 4.2.1.7, this transformation service will automatically execute when a given RWL model version does not match the meta-model version. The transformation service will then make the necessary model changes which will allow the end-user to successfully load it into the UI. In case this model transformation cannot be fully automated for a number of reasons, from model complexity or model error, the user will be asked to intervene in the process at appropriate junctures.

9.5.5 Edit Points

Edit points would allow advanced RWL users to tweak the generated RWL script. They would also be capable of detecting where custom RWL code has been added and preserve these changes the next time the RWL script is generated from the model. Edit points would allow expert users to represent complex RWL logic within the script which would be difficult to depict in the visual RWL model.

```
1  Code    RW_EXAMPLE
2  Type    Standard
3  Access  STSR
4
5  PageHeader
6      Print StandardPageHeader;
7  End
8
9  // EDIT_POINT_START {EC3617B5-CB48-4020-8D38-9191A29193EF}
10 // Custom code goes here
11 // EDIT_POINT_END   {EC3617B5-CB48-4020-8D38-9191A29193EF}
12
13 Scan RM
14     Print RM_CUST;
15 End
16
17 // EDIT_POINT_START {086F7339-B8BB-4152-B92E-909DD7BD9B60}
18 // Custom code goes here
19 // EDIT_POINT_END   {086F7339-B8BB-4152-B92E-909DD7BD9B60}
20
21 Print StandardReportFooter;
22
```

Figure 215: Edit points within RWL

The figure above (Figure 215) shows a sample implementation of *edit points*. We see each edit point uniquely identified by a GUID (Globally Unique Identifier) which could possibly be read in prior to regenerating script from the model and then maintained at the right position within the newly created script.

9.5.6 Import RWL Script

The Prism WIN MIS currently ships with a set of standard RWL scripts. Prism customers also have hundreds of pre-existing customised RWL scripts. A future application or plug-in could be developed which an end-user could use to import these scripts into the RWL model designer and visually represent it using the RWL Shell Host. This would allow an end-user to edit already existing RWL scripts using our newly developed visual language via the shell.

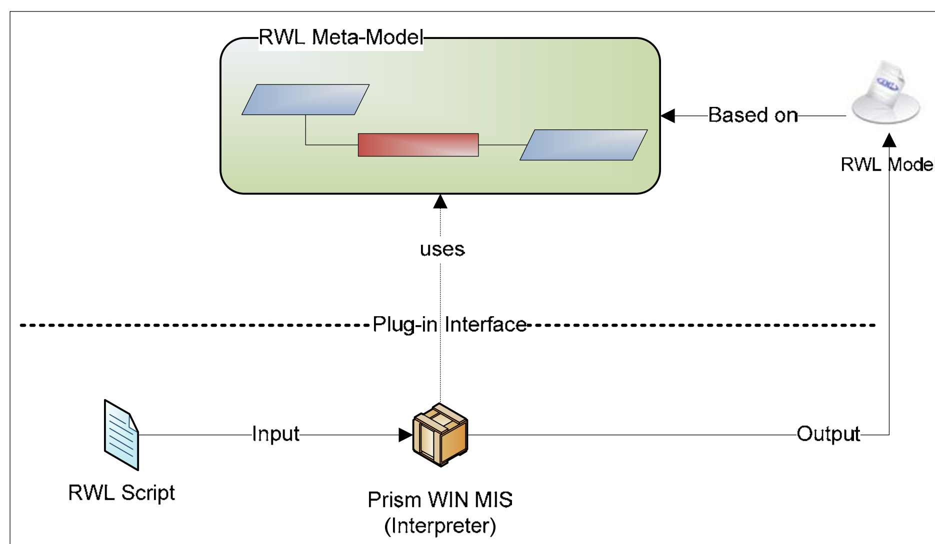


Figure 216: Sample architecture of RWL Script to Model Handler

Figure 216 above shows the sample architecture of a possible solution. Note the role of the Prism WIN MIS as a RWL interpreter, this is necessary as the RWL is not XML based and therefore does not have any schema as such. Thus, a potential RWL script could be written in any format and the only way to standardize it is by running it through the interpreter and allowing the interpreter to interact with our newly created meta-model to create the corresponding RWL model.

9.5.7 Wizards

Our solution has no template or wizard mechanism to allow end-users to create commonly used reports. This could be easily catered for as a future enhancement in the form of extra XML files which can be distributed along with the visual designer. Because our RWL model will be XML based we could potentially load any valid XML model into the designer allowing us to pre-create RWL models representing common reports and distribute them to the user base.

9.5.8 Code Snippets

The concept of code snippets is similar to that of wizards although it works on a smaller scale. There are many commonly used groups of RWL constructs, for e.g. if we want to get all the jobs for a given customer we would have the code snippet as shown below:

```
1.  Scan RM
2.      Scan QM
3.          Choose(QM_CUST_CODE, MATCH, RM_CUST)
4.          Choose(QM_QUOTE_JOB, MATCH, QMM_JOB)
5.      End
6.  End
```

This code block is potentially a code snippet which could be distributed along with the visual designer. Code snippets would make common repetitive tasks easier for end-users as they simply drag and drop code snippets which would give them the required visual elements on the canvas. Taking this one step further we could give users the ability to save commonly used code snippets which they could reuse at a later time.

Once again this can be implemented using XML as our visual RWL model is XML based.

9.5.9 Debug Generated RWL Script

Currently there is no possible way the end-user could hook into the text templating engine to check the RWL script generation. Therefore if the end-user finds a generated script which does not match their specifications they can trace back to the model element which generated the discrepancy and fix the problem. This trace can be trivial in a simple model although as the RWL models grow in complexity a simple visual check will not be feasible, this is when a debugger would prove essentially useful to step through the model to accurately analyze how the generated script is formed.

9.5.10 WYSIWYG Output Layout

The RWL is a reporting language which allows user to not only extract information from the Prism WIN database but also to perform layout operations on that data to display it using a customizable layout strategy.

This layout information is entered within the model via functions (like *AtCol* and *AtRow*) although it is not represented visually in any form via the model. A future enhancement could be added in the form of a layer on top of the visual model which would allow the user to visually see how the generated RWL script would look like when run by the Prism WIN MIS reporting engine. A potential problem is that the layout information is not known at compile time and is only determined dynamically when executed by the Prism WIN MIS reporting engine. Therefore to allow WYSIWYG output layout via the shell it will have to be tightly integrated with the Prism WIN MIS reporting engine.

9.6 Summary

In this chapter we concluded that we made a significant contribution to the software development processes at Prism by providing a framework which can be used by developers to modelise enterprise tasks. The primary contribution was the design of a report writer tool for end-users (Prism customers) allowing them to visually design Prism reports using a simple graphical notation.

The chapter also gave a list of limitations from which the most important one is that this solution is aimed at novice and intermediate report designers. The chapter finally gave a list of further enhancements that can be made to the solution of which expanding and refining the meta-model is of utmost importance.

References

Afonso, M., Vogel, R., & Teixeira, J. (2006). From Code Centric to Model Centric Software Engineering: Practical case study of MDD infusion in a Systems Integration Company. *Proceedings of the Fourth Workshop on Model-Based Development of Computer-Base Systems and Third International Workshop on Model-Based Methodologies for Pervasive and Embedded Software.* IEEE Computer Society.

Alessandro. (2007, July). Retrieved July 2008, from Trolltech Labs: http://labs.trolltech.com/blogs/2007/07/11/develop-qt-applications-in-eclipse/

Balaguer, F., & Yoder, J. W. (2001). *Adaptive Object Model.* Retrieved May 2008, from http://www.adaptiveobjectmodel.com

Beydeda, S., Book, M., & Gruhn, V. (2005). *Model-Driven Software Development.* Leipzig, Germany: Springer.

Bézivin, J., Hillairet, G., Jouault, F., Kurtev, I., & Piers, W. Bridging the MS/DSL Tools and the Eclipse Modeling Framework. *ATLAS Group (INRIA & LINA, University of Nantes).*

Blackwell, A. F., & Green, T. R. (1999). Does Metaphor Increase Visual Language Usability? *IEEE Symposium on Visual Languages.* Tokyo.

Blackwell, A. F., & Green, T. R. (1999). Investment of Attention as an Analytic Approach . *Collected Papers of the 11th Annual Workshop of the Psychology of Programming Interest Group*, (pp. 24-35). Leeds, UK.

Blackwell, A. F., Burnett, M. M., & Jones, S. P. *Champagne Prototyping: A Research Technique for Early Evaluation of Complex End-User Programming Systems.*

Brown, A. W., Conallen, J., & Tropeano, D. (2005). Introduction: Models, Modeling, and Model-Driven Architecture (MDA). In S. Beydeda, M. Book, & V. Gruhn, *Model-Driven Software Development* (pp. 1-16). Leipzig: Springer.

Cook, S., Jones, G., Kent, S., & Wills, A. C. (2007). *Domain-Specific Development with Visual Studio DSL Tools.* Boston: Addison-Wesley.

Cook, Steve. (2005, June). Retrieved June 2008, from Steve Cook's Weblog: http://blogs.msdn.com/stevecook/archive/2005/06/03/424897.aspx

DevSource. (2007). *Domain-Specific Modeling*. Retrieved March 2008, from DevSource: http://www.devsource.com/c/a/Using-VS/DomainSpecific-Development-with-Visual-Studio-Part-1/

Eclipse.org. (2008). *GMF Gallery*. Retrieved October 2008, from Eclipse: http://www.eclipse.org/modeling/gmf/gallery/index.php

Ferguson, R. I., & Hunter, A. *MetaBuilder: The Diagrammer's Diagrammer.* Sunderland, UK: University of Sunderland, School of Computing, Engineering and Technology.

Green, T. (1996). An Introduction to the Cognitive Dimensions Framework. *MIRA workshop.* Cambridge, UK.

Greenfield, J. (2005, August). Software Factories. *Perspectives of the International Association of Software Architects* , 2-7.

Grundy, J. (2008). Other Meta Tools. Auckland, New Zealand: University of Auckland.

Grundy, J. (2008). *Visual Languages/Notations.* University of Auckland. Auckland: University of Auckland.

Grundy, J., Hosking, J., Huh, J., & Li, K. Marama: an Eclipse meta-toolset for generating multi-view environments. Auckland.

Himalia. (2006). Retrieved March 2008, from Himalia: Model-driven user interfaces: http://www.himalia.net/

Klein, F. J. (n.d.). *Relativity - Business Technology Solutions*. Retrieved June 2008, from Basic Features of the Agile Software Development Model: http://www.relativitycorp.com/projectmanagement/article5.html

Liu, N., Hosking, J., & Grundy, J. (2007). MaramaTatau: extending a domain specific visual language meta-tool with a declarative constraint mechanism. *VLHCC* (pp. 95-103). IEEE CS Press.

McIntyre, D. (1994, December). Retrieved July 2008, from Visual Languages: http://www.hypernews.org/~liberte/computing/visual.html

MetaCase. (2008). *MetaEdit+ Modeler - Supports your modeling language*. Retrieved October 2008, from MetaCase: http://www.metacase.com/mep/

Metzger, A. (2005). A Systematic Look at Model Transformations. In S. Beydeda, M. Book, & V. Gruhn, *Model-Driven Software Development* (pp. 19-33). Leipzig: Springer.

Microsoft. (2007). Retrieved October 2007, from Overview of Domain-Specific Language Tools: http://msdn.microsoft.com/en-us/library/bb126327(vs.80,printer).aspx

Microsoft. (2007, October). *Building Software Factories - Part 1, what are we building and why?* Retrieved September 2008, from MSDN Architecture Centre: http://msdn.microsoft.com/en-us/architecture/bb871630.aspx

Microsoft. (2008). *Product Family Life-Cycle Guidelines for Visual Basic 6.0*. Retrieved September 2008, from Visual Basic 6.0 Resource Center: http://msdn.microsoft.com/en-us/vbrun/ms788707.aspx

Microsoft. (2008). *Visual Studio 2008 Extensibility*. Retrieved June 2008, from Visual Studio 2008 Extensibility: http://msdn.microsoft.com/en-us/vsx2008/products/bb933751.aspx

Naba kumar. (2007). Retrieved July 2008, from Anjuta DevStudio: http://anjuta.sourceforge.net/

Object Management Group. (2008, January). Retrieved June 2008, from OMG's MetaObject Facility (MOF): http://www.omg.org/mof/

Pelechano, V., Albert, M., Muñoz, J., & Cetina, C. (2006). *Building Tools for Model Driven Development. Comparing Microsoft DSL Tools and Eclipse Modeling Plug-ins.* Valencia: Technical University of Valencia.

Prism Group. (2008). *Prism - Better information. Better business.* Retrieved October 2008, from Prism - Management Information System (MIS) Software for the Print Industry: http://www.prism-world.com/

Prism New Zealand. (2005). *Prism WIN - Report Writer Handbook.* Auckland: Prism Group.

Roddick, J. F. (1995). *A survey of schema versioning issues for database systems.* University of South Australia, School of Computer and Information Science. Australia: University of South Australia.

Sorensen, R. (n.d.). *Comparison of Software Development Methodologies*. Retrieved June 2008, from A Comparison of Software Development Methodologies: http://www.stsc.hill.af.mil/crosstalk/1995/01/Comparis.asp

Spolsky, J. (2001). *User Interface Design for Programmers.* California, United States of America: Apress.

Stahl, T., Völter, M., Bettin, J., Haase, A., & Helsen, S. (2003). *Model-Driven Software Development.* Heidelberg: Wiley.

Subramaniam, V., & Hunt, A. (2006). *Practices of an Agile Developer.* The Pragmatic Bookshelf.

University of Geneva. (n.d.). Retrieved July 2008, from What is BNF notation?: http://cui.unige.ch/db-research/Enseignement/analyseinfo/AboutBNF.html

Wikipedia. (2007). *Domain-Specific programming language*. Retrieved March 2008, from Wikipedia: http://en.wikipedia.org/wiki/Domain-Specific_programming_language

Appendices

Appendix A Report Writer Factsheet by (Prism Group, 2008)

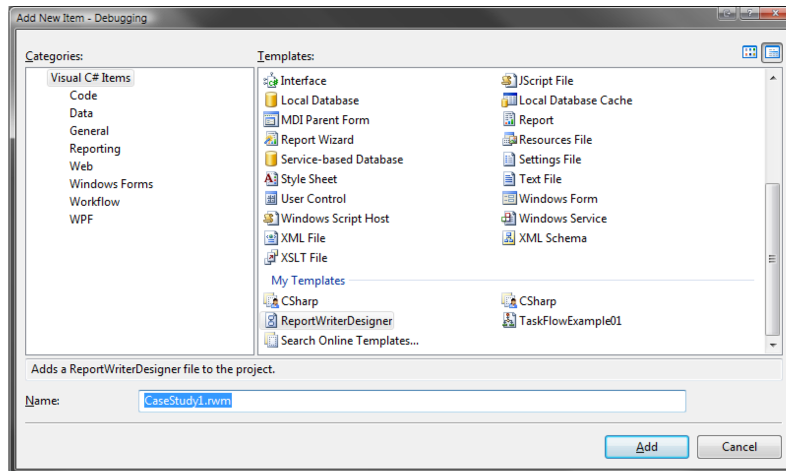
Attached

Appendix B Core RWL Constraints

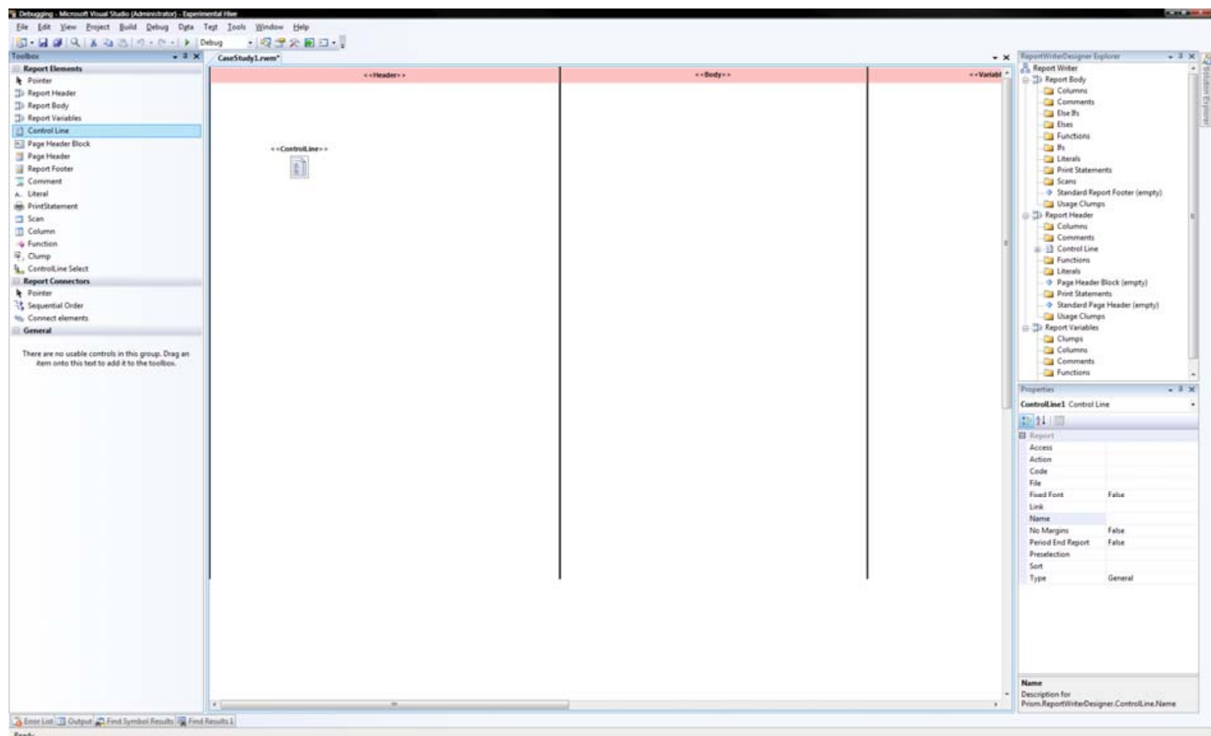
RWL Construct (Model Element)	Constraints
HeaderSection	• Has to contain on and only one ControlLine • Can contain a PageHeaderBlock and StandardPageHeader
ControlLine	• Can only be added to the header section • Can contain a Select model element
Select	• Can only be part of the ControlLine model element
	• Can only be dropped on the HeaderSection
StandardPageHeader	• Can only be dropped on the HeaderSection
PageHeaderBlock	• Can only be dropped on the HeaderSection • Can contain PrintStatement model element • Can contain one and only one StandardPageHeader
PrintStatement	• Can be dropped anywhere on the report
Scan	• Can only be dropped on the BodySection
StandardReportFooter	• Can only be dropped on the BodySection
Literal	• Needs a parent PrintStatement model element
Column	• Needs a parent PrintStatement or Clump model element • Can be dropped anywhere on the report
Function	• Needs a parent Column or Clump/ClumpUsage or Literal • Can be dropped anywhere on the report
Clump	• Can only be dropped on the VariablesSection • If dropped on any other section, an instantiation will be added in the VariablesSection and a usage will be added in the specified section
Comment	• Can be dropped anywhere on the report

Appendix C Case Study 1 Screenshots

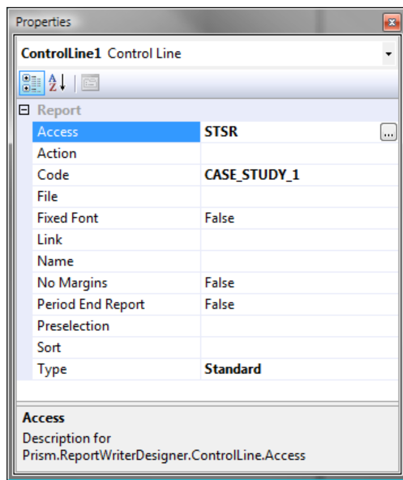
Step 1:



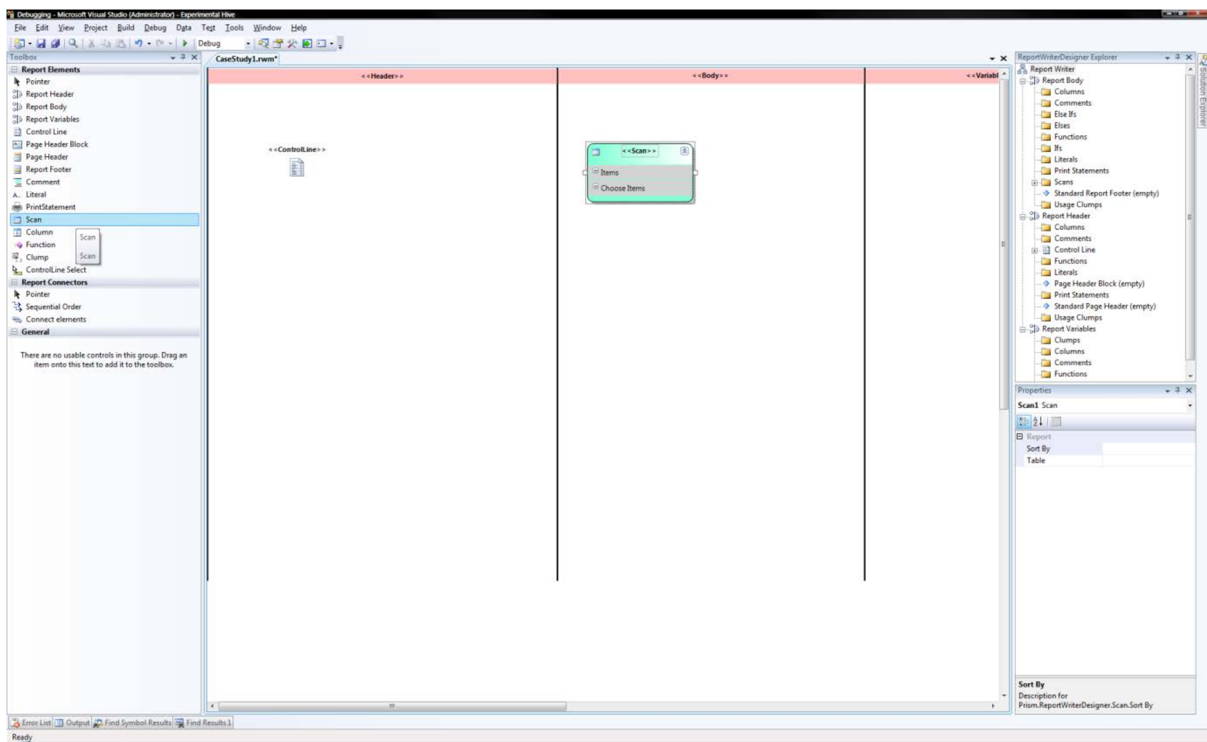
Step 3:



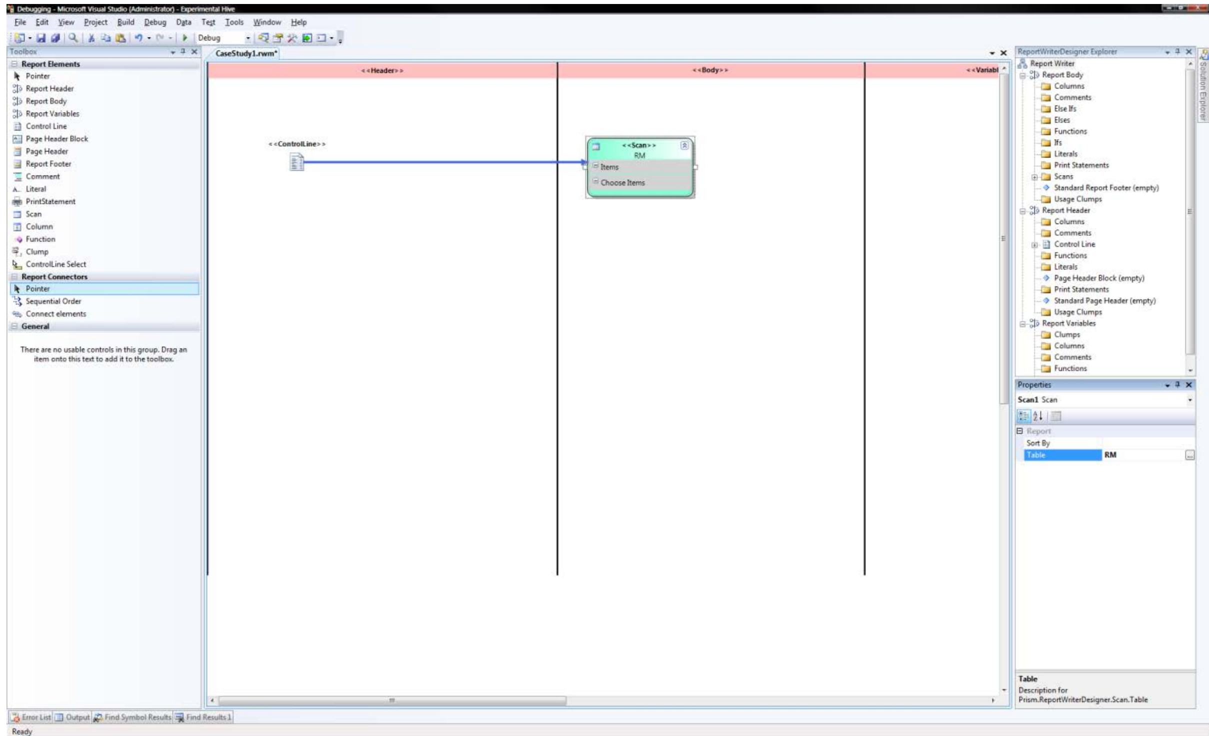
Step 4:



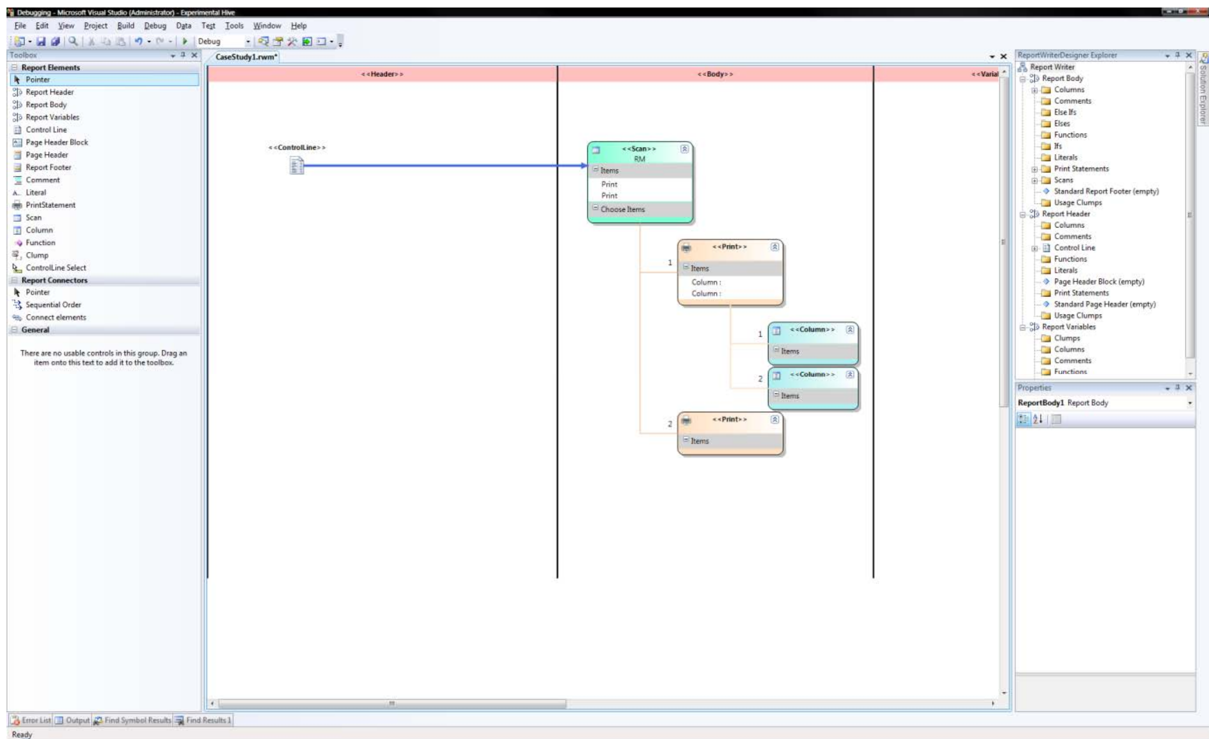
Step 5:



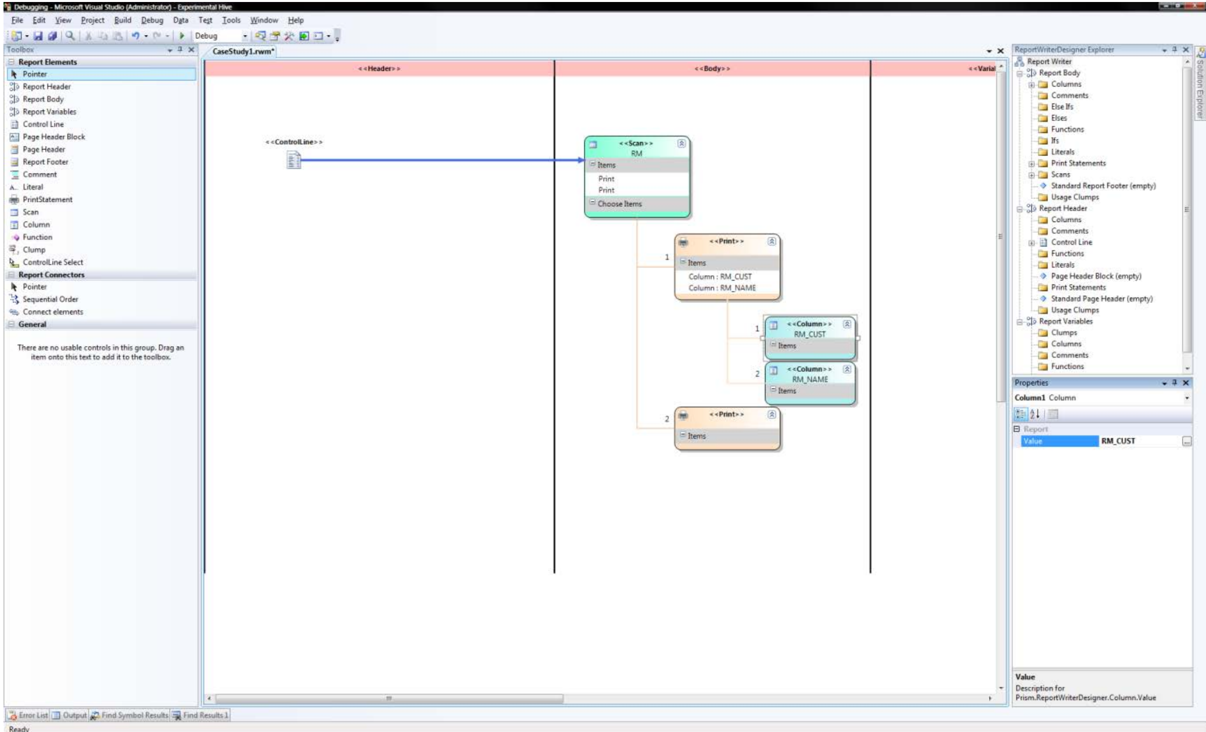
Step 6:



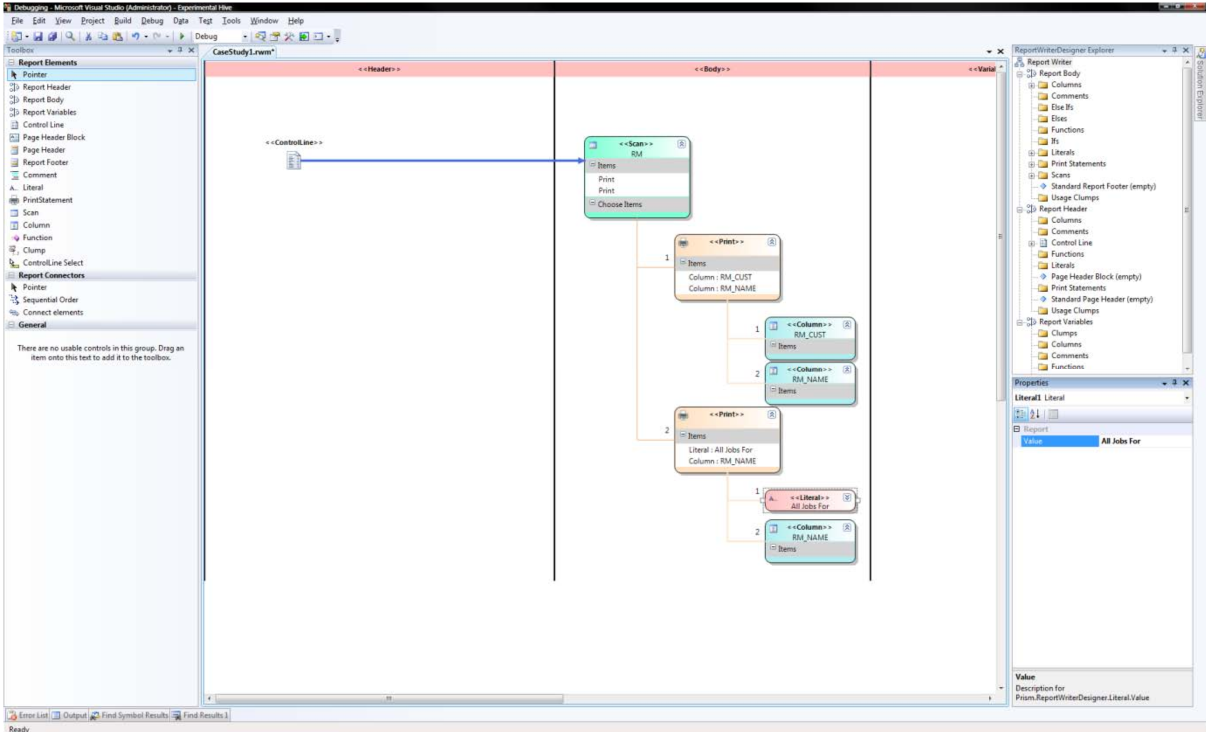
Step 8:



Step 9:



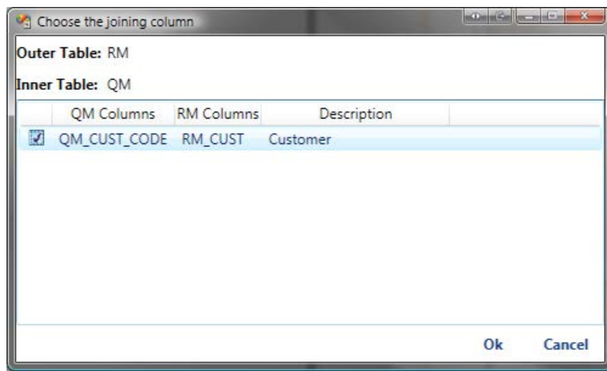
Step 10:



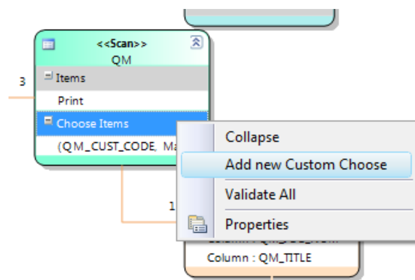
Step 11:

Step 12:

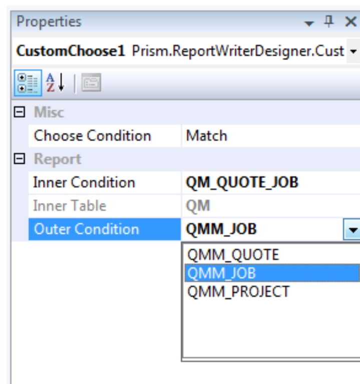
Step 14:



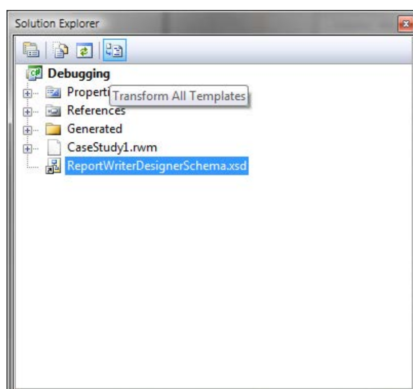
Step 15:

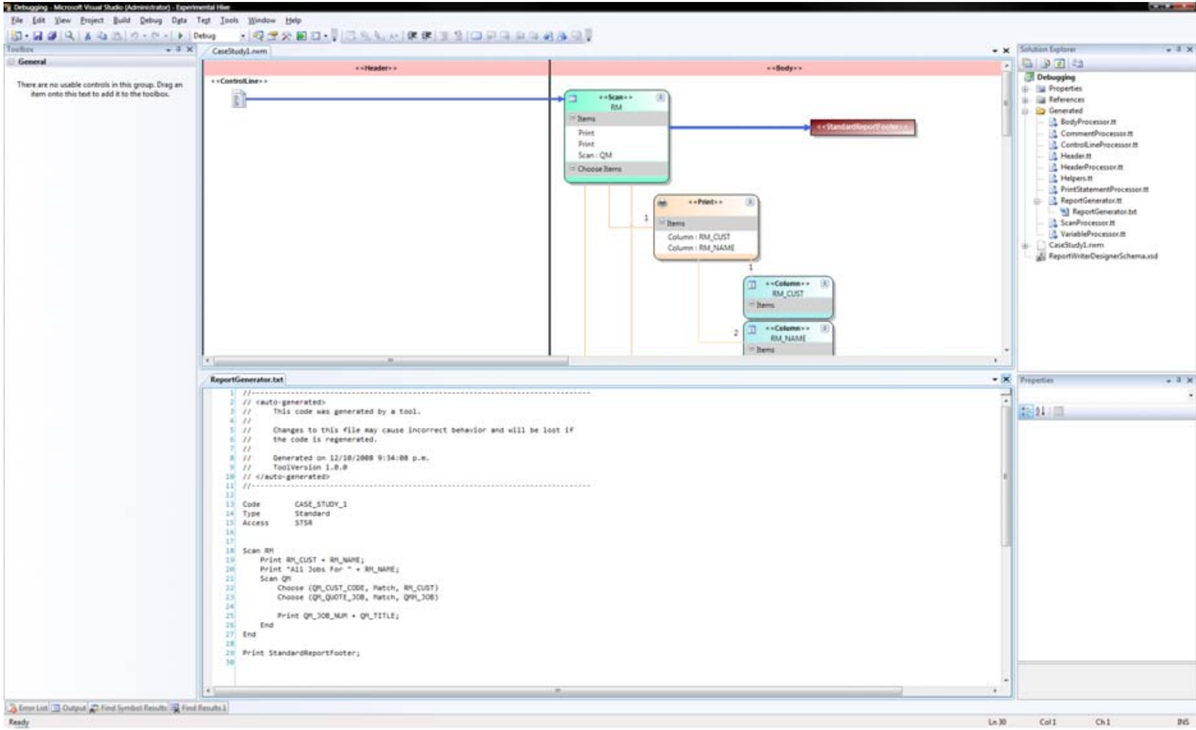


Step 16:



Step 18:





Appendix D *ExpandCollapseBase* Hide/Show children

```
public partial class ExpandCollapseBase
{
        protected override void Expand()
        {
                base.Expand();

                if (this.ModelElement is NamedElement)
                {
                        NamedElement namedElement = this.ModelElement as NamedElement;

                        CollapseExpandHelper.HideShowModels(namedElement.Targets, HideShow.Show);
                }

                this.Diagram.Invalidate();
        }

        protected override void Collapse()
        {
                base.Collapse();

                if (this.ModelElement is NamedElement)
                {
                        NamedElement namedElement = this.ModelElement as NamedElement;

                        CollapseExpandHelper.HideShowModels(namedElement.Targets, HideShow.Hide);
                }

                this.Diagram.Invalidate();
        }
}
```

Appendix E Compartment Child Ordering

```
public static class FilterHelper
{
        public static List<NamedElement> OrderedList(ReadOnlyLinkedElementCollection<NamedElement>
                                                        namedElements)
        {
                List<NamedElement> sortedList = new List<NamedElement>();

                List<NamedElementReferencesNamedElements> namedElementReferences = new
                                                        List<NamedElementReferencesNamedElements>();
                foreach (var item in namedElements)
                        foreach (var innerItem in
                                        NamedElementReferencesNamedElements.GetLinksToSources(item))
                                namedElementReferences.Add(innerItem);

                IOrderedEnumerable<NamedElementReferencesNamedElements> orderedNamedElementReferences
                                = namedElementReferences.OrderBy(p => p.Order);
                foreach (var item in orderedNamedElementReferences)
                        sortedList.Add(item.Target);

                return sortedList;
        }
}
```

Appendix F Automatic Layout of Child Shapes

```
public static class ILayoutPolicyHelper
{
        public static void LayoutPolicy(NodeShape parentShape, ref PointD startPoint, ref RectangleD
                                        lastChildBounds)
        {
                // do not do this while undo/redo
                if (parentShape.Store.InUndoRedoOrRollback) return;

                IOrderedEnumerable<NamedElementReferencesNamedElements> orderedNamedElements =
                        NamedElementReferencesNamedElements.GetLinksToTargets(parentShape.Subject as
                        NamedElement).OrderBy(f => f.Order);
                foreach (NamedElementReferencesNamedElements namedElementReference in
                                                                orderedNamedElements)
                {
                        NamedElement namedElement = namedElementReference.Target;

                        using (Transaction t =  parentShape.Store.TransactionManager.
                                                BeginTransaction("AutoLayoutShapes", true))
                        {
                                #region Shape location
                                LinkedElementCollection<PresentationElement> shapes =
                                        PresentationViewsSubject.GetPresentation(namedElement);
                                LayoutHelper.LayoutChildShape(ref startPoint, parentShape, shapes,
                                        ref lastChildBounds);
                                #endregion

                                t.Commit();
                        }

                        using (Transaction t = parentShape.Diagram.Store.TransactionManager.
                                        BeginTransaction("AutoLayoutLinks", true))
                        {
                                #region Links location
                                LinkedElementCollection<PresentationElement> linksShapes =
                                        PresentationViewsSubject.
                                        GetPresentation(namedElementReference);
                                LayoutHelper.LayoutLinksShapes(false, parentShape, linksShapes);
                                #endregion

                                t.Commit();
                        }
                }

                parentShape.Diagram.Invalidate();
        }
}
```

Appendix G Survey Request Letter

Attached

Appendix H Survey Participation Information Sheet

Attached

Appendix I Consent Form

Attached

Appendix J Survey

Attached