

# Lecture Notes on Software Engineering

## r3304

Tim Storer  
[timothy.storer@glasgow.ac.uk](mailto:timothy.storer@glasgow.ac.uk)  
Room 304 SAWB

2012-13



# Acknowledgments

These notes are a mixture of my own material on Software Engineering, that provided by Ray Welland and Phil Gray and material derived (well co-opted) from the cited sources. The case study is derived from the coursework topic set for the PSD3 class in 2009-10 and so is also partly influenced by the solutions proposed by students in that cohort.

The notes are typeset in L<sup>A</sup>T<sub>E</sub>X, using the beamer package.<sup>1</sup> Diagrams were produced using MetaPost<sup>2</sup>. The MetaPost module metauml<sup>3</sup> was used to for the UML diagrams, and the Expressg<sup>4</sup> module was used for flow charts.

---

<sup>1</sup><http://latex-beamer.sourceforge.net/>

<sup>2</sup><http://www.tex.ac.uk/ctan/graphics/metapost/>

<sup>3</sup><http://metauml.sourceforge.net/>

<sup>4</sup><http://www.ctan.org/tex-archive/graphics/metapost/contrib/macros/expressg>

# Some Administration

Like this – Some administration (2)

This booklet of lecture notes incorporates material from the Object Oriented Software Engineering 2, Professional Software Development 3, Requirements Engineering M and Object Oriented Software Engineering M courses. The material covered in each of the courses varies.

- The labels that appear in the margin roughly correspond to the title of a slide used in the lecture course.
- There is an electronic version of the notes available online on each of the course moodle sites.
- The electronic version of the lecture notes may be updated as the lecture course progresses.

Each chapter of notes starts with pointers for further reading called *Recommended Reading*, which means 'like fresh green vegetables' - they may not taste very nice, but will benefit you in the long run. The notes also reference a small bibliography, which you should also consider reviewing, particularly if you are studying more advanced topics.

These notes *should not* be considered as an adequate replacement for a good text book on Software Engineering and your own notes from the lectures. Each of the courses covered by these notes is accompanied by a recommended text book.

Questions during the course? (3)

If you have questions during the course, you should:

- interrupt during the lecture - please feel free to stop the lecture if you feel there is something I haven't explained properly, got wrong or contradicts something else you have read. Someone else might be thinking the same thing. Remember, you are not assessed in anyway based on your contributions to lectures. Please do not wait until the end of the lecture to ask your question, time is often tight to get the next lecture started and for you and other students to get to the next class. Also, it means that no one else will get the benefit of the answer.;
- comment on the Moodle page for the relevant course;

- open door policy - this means you can come and see me whenever you have a question. I **will not respond to emailed questions** unless they pertain to personal circumstances. This is because it is easier to work out what the real problem is through an in-person conversation so that we can work through examples etc. Also, verbal communication has the benefit of deniability...



# Contents

<b>I The Software Engineering Process</b>	<b>1</b>
<b>1 Introduction to Software Engineering</b>	<b>3</b>
1.1 Software Failures . . . . .	3
1.2 What Makes Software Development Hard? . . . . .	6
1.3 What is Software Engineering? . . . . .	7
1.4 Project Value and Quality . . . . .	8
1.5 Software Engineering in the Small . . . . .	11
<b>2 The Object Oriented Paradigm and Java</b>	<b>13</b>
2.1 Introduction . . . . .	13
2.2 Objects and Classes . . . . .	13
2.3 Types and Members . . . . .	15
2.3.1 Types . . . . .	15
2.3.2 Instance Members . . . . .	16
2.3.3 Class Members . . . . .	18
2.4 Polymorphism, Inheritance and Abstraction . . . . .	19
2.4.1 Polymorphism in the Collections Framework . . . . .	22
2.4.2 Enumerations . . . . .	28
2.5 Annotations . . . . .	29
2.6 Packages . . . . .	29
2.7 Exceptions . . . . .	30
2.7.1 The Exceptions Framework in Java . . . . .	31
2.7.2 Managing Exceptions . . . . .	33
2.7.3 The Throwable Class Hierarchy . . . . .	36
2.8 Programming Style and Documentation . . . . .	37
2.9 Exercises . . . . .	43
2.10 Workshop: Generic Types and Auto-Boxing in Java . . . . .	46
2.10.1 Runtime Type-cast Defects . . . . .	46
2.10.2 Using Generics . . . . .	47
2.10.3 Scoping Generic Types . . . . .	48
2.10.4 Behind the Scenes . . . . .	51
<b>3 The Software Lifecycle</b>	<b>53</b>
3.1 Software Engineering in the Large . . . . .	54

3.2	Software Development Activities . . . . .	63
3.3	Software Development Process Models . . . . .	65
3.4	Reusable Software . . . . .	65
3.5	Object Oriented Software Development . . . . .	67
<b>4</b>	<b>Software Process Models</b>	<b>71</b>
4.1	Introduction . . . . .	71
4.2	V-Model . . . . .	74
4.3	Spiral Model . . . . .	75
4.4	Iterative Models . . . . .	78
4.5	The Rational Unified Process . . . . .	79
4.6	Agile Methods . . . . .	87
4.7	Extreme Programming . . . . .	89
4.7.1	Planning and Specification . . . . .	89
4.7.2	Testing and Coding . . . . .	93
4.7.3	Prototyping, Design and Architecture . . . . .	96
<b>5</b>	<b>Project Management</b>	<b>99</b>
5.1	Working in Groups . . . . .	99
5.1.1	Models of Team Organisation . . . . .	100
5.1.2	Roles in Teams . . . . .	101
5.1.3	Group Communication . . . . .	103
5.2	Project Planning and Scheduling . . . . .	104
5.2.1	Documenting Project Plans . . . . .	105
5.2.2	Estimating Task Duration . . . . .	108
5.2.3	Visualising Project Schedules . . . . .	113
5.2.4	Calculating Critical Paths in a Schedule . . . . .	117
5.3	Exercises . . . . .	121
5.4	Workshop: Using Trac for Project Planning . . . . .	122
5.4.1	Logging into the Trac Server . . . . .	122
5.4.2	Setting up the Site . . . . .	123
5.4.3	Creating and Viewing Tickets . . . . .	125
5.4.4	Customising Trac to Fit your Project . . . . .	126
5.4.5	Extending Trac . . . . .	128
<b>6</b>	<b>Risk Management</b>	<b>131</b>
6.1	The Risk Management Process . . . . .	131
6.2	Identifying Risks . . . . .	133
6.3	Analysing Risks . . . . .	135
6.4	Controlling Risk . . . . .	137
6.5	Documentation and Risk Management . . . . .	138
<b>7</b>	<b>Quality Assurance</b>	<b>141</b>

<b>8 Change Management</b>	<b>143</b>
8.1 Software and Change . . . . .	143
8.2 Change Management Process . . . . .	146
8.3 Communicating Change . . . . .	152
8.4 Automated Version Management . . . . .	155
8.5 Workshop: Version Control using Subversion . . . . .	159
8.5.1 Creating a Repository . . . . .	160
8.5.2 Working with Versions . . . . .	162
8.5.3 Using Properties . . . . .	164
8.5.4 Branches, Tags, Conflicts and Merges . . . . .	166
8.5.5 Going Further . . . . .	169
<b>9 Software Metrics</b>	<b>171</b>
<b>10 Documentation and Technical Writing</b>	<b>173</b>
10.1 General Considerations . . . . .	173
10.2 Structuring Documentation . . . . .	174
10.3 Quality Assurance and Documentation . . . . .	178
<b>II Requirements Engineering</b>	<b>185</b>
<b>11 Requirements Engineering</b>	<b>187</b>
11.1 Requirements Risk . . . . .	187
11.2 The Requirements Process . . . . .	188
11.3 Requirements Elicitation . . . . .	191
11.3.1 Questionnaires . . . . .	191
11.3.2 Observations . . . . .	192
11.3.3 Interviews . . . . .	192
11.3.4 Walk throughs . . . . .	193
11.4 Documenting Requirements as Use Cases . . . . .	194
11.5 Requirements Validation and Analysis . . . . .	199
11.6 Exercises . . . . .	203
<b>12 Requirements Refinement and Planning</b>	<b>205</b>
12.1 Recap - Branch Library . . . . .	205
12.2 Planning . . . . .	209
12.3 Describing Use Cases . . . . .	210
12.4 Activity Diagrams . . . . .	214
12.5 Advanced Features . . . . .	217
12.6 Completing Use Case Descriptions . . . . .	220
12.7 Exercises . . . . .	223
<b>13 Object Oriented Analysis with UML</b>	<b>225</b>
13.1 Introduction - The Unified Modelling Language . . . . .	225

13.2	Developing Object Oriented Domain Models . . . . .	227
13.3	Class Diagrams . . . . .	228
13.3.1	Associations and Constraints . . . . .	230
13.3.2	Validating Class Associations . . . . .	233
13.3.3	Decorating Associations . . . . .	234
13.3.4	Polymorphism during analysis . . . . .	239
13.3.5	Extending Class Diagrams . . . . .	245
13.4	Exercises . . . . .	250
<b>14</b>	<b>From Domains to Designs</b>	<b>255</b>
14.1	Introduction . . . . .	255
14.2	Class Responsibility Collaboration . . . . .	257
14.3	CRC Cards to UML Class Diagrams . . . . .	259
14.4	Exercises . . . . .	262
<b>15</b>	<b>Modelling Dynamic Behaviour</b>	<b>263</b>
15.1	Introduction . . . . .	263
15.2	Sequence Diagrams . . . . .	264
15.3	Communication Diagrams . . . . .	270
15.4	From Use Cases to Architecture . . . . .	274
15.5	Exercises . . . . .	286
<b>16</b>	<b>Prototyping</b>	<b>287</b>
16.1	Introduction . . . . .	287
16.2	Why prototype? . . . . .	287
16.3	The Prototyping Process . . . . .	289
16.4	Approaches to Prototyping . . . . .	290
16.4.1	Throw-Away Prototypes . . . . .	290
16.4.2	Evaluating New Technology . . . . .	292
16.4.3	Proof of Concept . . . . .	294
16.4.4	User Interface Prototyping . . . . .	295
16.4.5	Rapid Application Development (RAD) . . . . .	296
16.4.6	Incremental Prototypes . . . . .	298
<b>III</b>	<b>Software Design, Implementation and Testing</b>	<b>301</b>
<b>17</b>	<b>Software Re-use and Frameworks</b>	<b>303</b>
17.1	Re-usable Software Components . . . . .	304
17.2	Choosing and Deploying Re-usable Software . . . . .	306
17.3	Developing re-usable software . . . . .	309
17.4	The Java Swing GUI Framework . . . . .	311
17.4.1	Basics . . . . .	312
17.4.2	Menu Bars and Tool Bars . . . . .	316

17.4.3	Controlling Container Layout . . . . .	317
17.4.4	Events . . . . .	322
17.4.5	Dialogs . . . . .	332
17.5	The Apache Axis Web Service Framework . . . . .	335
17.5.1	Deploying Web Services . . . . .	336
17.5.2	Distributed System/Web Service Engineering . . . . .	341
17.5.3	Serializing and De-serializing JavaBeans . . . . .	344
<b>18</b>	<b>Using Design Patterns</b>	<b>349</b>
18.1	Describing and Choosing Pattern . . . . .	349
18.2	Structural Patterns . . . . .	351
18.2.1	Abstraction – Occurrence . . . . .	351
18.2.2	Player – Role . . . . .	354
18.2.3	Adapter . . . . .	356
18.2.4	Proxy . . . . .	358
18.2.5	Façade . . . . .	363
18.2.6	Composite Pattern . . . . .	366
18.2.7	Read-only Interface . . . . .	368
18.3	Creational Patterns . . . . .	369
18.3.1	Singleton . . . . .	369
18.3.2	Factory . . . . .	372
18.4	Behavioural Patterns . . . . .	375
18.4.1	Iterable – Iterator . . . . .	375
18.4.2	Delegation . . . . .	376
18.4.3	Observer – Observable . . . . .	378
18.4.4	Other Design Patterns . . . . .	380
18.5	Workshop: Inner Classes . . . . .	382
18.6	Exercises . . . . .	383
<b>19</b>	<b>Design Principles and Practice</b>	<b>387</b>
19.1	Introduction . . . . .	387
19.2	Design Principles . . . . .	391
19.2.1	Maintain a Separation of Concerns . . . . .	392
19.2.2	Maintain Sufficient Abstraction . . . . .	394
19.2.3	Reuse and Reusability . . . . .	397
19.2.4	Design for Testing . . . . .	397
19.2.5	Increasing Cohesion . . . . .	398
19.2.6	Reduce Coupling . . . . .	406
19.3	Documenting and Evaluating Designs . . . . .	418
19.4	Exercises . . . . .	420
<b>20</b>	<b>Software Architecture</b>	<b>425</b>
20.1	Exercises . . . . .	426

<b>21 Software Testing</b>	<b>427</b>
21.1 Aims of Testing and Basic Concepts . . . . .	427
21.2 Black box Testing . . . . .	431
21.3 White box Testing . . . . .	435
21.4 Categorising Defects . . . . .	438
21.4.1 Control Defects . . . . .	438
21.4.2 Numerical defects . . . . .	441
21.4.3 Documentation and External dependency defects . . . . .	443
21.5 Testing Strategies and Integration . . . . .	444
21.6 Managing Test Cases with JUnit . . . . .	448
21.7 Exercises . . . . .	452
21.8 Workshop: Agile Practices for Testing . . . . .	455
21.8.1 Agile Methods . . . . .	455
21.8.2 Scanners . . . . .	456
21.8.3 Procedure . . . . .	456
21.8.4 The Problem . . . . .	456
<b>22 Testing and Assurance of Large Scale, Complex Software Based Systems</b>	<b>459</b>
22.1 Introduction . . . . .	459
22.2 Testing Non-functional Requirements . . . . .	463
22.2.1 Reliability, Safety and Security Testing . . . . .	464
22.2.2 Performance Testing . . . . .	467
22.2.3 Threshold Testing . . . . .	467
22.3 Managing Testing within Teams . . . . .	468
22.3.1 Developing Acceptance Tests . . . . .	468
22.3.2 Regression Testing . . . . .	471
22.3.3 Conducting Acceptance Test Demonstrations . . . . .	472
22.3.4 Automating Acceptance Tests . . . . .	473
22.3.5 Evaluating Test Plans . . . . .	474
22.4 Workshop: Parameterized Tests and Theories in JUnit . . . . .	474
<b>23 Modelling System State</b>	<b>475</b>
23.1 Introduction . . . . .	475
23.2 State Transition Networks . . . . .	476
23.3 The State Machine Diagrams Notation . . . . .	480
23.3.1 Concurrency and Message Broadcast . . . . .	483
23.3.2 Conditions on Transitions . . . . .	484
23.3.3 Internal Actions . . . . .	485
23.3.4 State Persistence . . . . .	486
23.4 Using State Machine Diagrams in the Design Process . . . . .	486
23.5 Exercises . . . . .	495
23.6 Workshop: Using Threads . . . . .	497
23.6.1 Implementation in Java . . . . .	497

23.6.2 Timing and Coordination Defects . . . . .	502
23.6.3 Synchronization . . . . .	504
23.6.4 Synchronization Defects . . . . .	506
23.6.5 Summary . . . . .	508
<b>24 Aspect Oriented Development</b>	<b>509</b>
24.1 Summary . . . . .	509
24.2 Exercises . . . . .	510
<b>25 Formal Methods in Software Engineering</b>	<b>511</b>
25.1 Introduction . . . . .	512
25.2 Formal Specification in Software Development . . . . .	513
25.3 Using Formal Specifications . . . . .	515
25.4 The Object Constraint Language . . . . .	516
25.4.1 Writing Constraints . . . . .	517
25.5 Collections . . . . .	524
25.6 Constraints on Operations . . . . .	527
25.7 Model Driven Development . . . . .	529
25.8 Exercises . . . . .	532
25.9 Workshop: Model Driven Development . . . . .	537
25.9.1 Setup . . . . .	537
25.9.2 Exploring a Project . . . . .	538
25.9.3 Checking Constraints . . . . .	541
25.9.4 Modifying the Snapshot . . . . .	544
25.9.5 Adding More Constraints . . . . .	544
25.9.6 Generating Source Code . . . . .	545
25.9.7 Extensions . . . . .	545
<b>IV Software Maintenance</b>	<b>546</b>
<b>26 Software Evolution and Maintenance</b>	<b>547</b>
26.1 Maintenance Costs in Software Engineering . . . . .	547
26.2 Lehman's Laws . . . . .	549
<b>27 Software Comprehension</b>	<b>551</b>
27.1 Source Code Comprehension . . . . .	553
27.2 Analysing Program Design . . . . .	555
27.3 Source Code Obfuscation . . . . .	559
27.4 Workshop: Software Comprehension with Eclipse . . . . .	561
27.5 Exercises . . . . .	562
<b>28 Software Refactoring</b>	<b>563</b>
28.1 The Refactoring Process . . . . .	564
28.2 Bad Smells . . . . .	565

28.3 Applying Refactorings . . . . .	568
28.4 Limits of Refactoring . . . . .	575
28.5 Workshop: Refactoring with Eclipse . . . . .	578
<b>29 Re-engineering Legacy Systems</b>	<b>585</b>
29.1 Legacy Systems . . . . .	585
29.2 Reverse Engineering . . . . .	589
29.3 Source Code Translation . . . . .	590
29.4 Architectural Re-structuring . . . . .	591
29.5 Platform Migration . . . . .	592
29.6 Workshop: Using Logging Frameworks (log4j) . . . . .	597
<b>A Use Case Description Template</b>	<b>609</b>

# List of Figures

1.1	failing in the small . . . . .	4
1.2	failing medium . . . . .	5
1.3	failing big . . . . .	5
1.4	quality as cost increases . . . . .	10
2.1	an initial problem description for sports league management system . . . . .	14
2.2	Rateable interface of the sportser application. . . . .	20
2.3	fruit inheritance relationships . . . . .	21
2.4	sports inheritance hierarchy . . . . .	22
2.5	the Collection inheritance hierarchy . . . . .	23
2.6	the Override annotation . . . . .	29
2.7	general scheme for handling abnormal conditions . . . . .	31
2.8	the throwable class hierarchy . . . . .	37
2.9	listing of a typical project directory layout . . . . .	38
2.10	university problem description . . . . .	44
3.1	rates of project failure . . . . .	54
3.2	causes of software project failure . . . . .	58
3.3	Scottish elections e-counting system . . . . .	60
3.4	Accenture share price, 6 <sup>th</sup> May 2010 . . . . .	62
3.5	the waterfall software development process model . . . . .	63
3.6	example problem definition . . . . .	68
3.7	initial requirements modelling for the branch library . . . . .	69
4.1	the waterfall SDP with feedback . . . . .	73
4.2	a close up view of the software evolution activity . . . . .	74
4.3	the V software development model . . . . .	75
4.4	the spiral model showing activities . . . . .	76
4.5	the phases of a software development organised into a spiral . . . . .	77
4.6	RUP phases . . . . .	81
4.7	process effort in different RUP phases, adapted from Jacobson et al. [1999] . . . . .	81
4.8	inception phase of the RUP . . . . .	83
4.9	elaboration phase of the RUP . . . . .	83

4.10	construction phase of the RUP . . . . .	84
4.11	transition phase of the RUP . . . . .	85
4.12	milestones and releases within the Rational Unified Process model . . . . .	85
4.13	subsequent cycles of the Rational Unified Process model . . . . .	86
4.14	overview of the XP process (re-drawn from Wells <sup>3</sup> ) . . . . .	90
4.15	an example XP user story . . . . .	90
4.16	an example project velocity . . . . .	92
4.17	test first development . . . . .	94
4.18	cost of fixing defects . . . . .	95
5.1	summary of milestones and work packages . . . . .	107
5.2	example tasks from the branch library management system project . . . . .	109
5.3	example task estimation and allocation for the branch library system . . . . .	110
5.4	the effect of adding additional team members to a late project (redrawn from Brooks [1995]) . . . . .	112
5.5	example Gantt chart . . . . .	115
5.6	example PERT chart . . . . .	116
5.7	introduction of dummy tasks into a project schedule . . . . .	116
5.8	critical paths on PERT charts . . . . .	118
5.9	An example project plan . . . . .	121
5.10	the front page of Trac . . . . .	124
5.11	adding a milestone to the Trac . . . . .	127
5.12	configuring Trac with the MoSCoW rules. . . . .	128
5.13	embedding a Gantt chart in Trac . . . . .	130
6.1	relationship between development and risk management processes . . . . .	132
6.2	a risk management process, adapted from Freimut et al. . . . .	132
6.3	categorising risks by probability and impact . . . . .	138
6.4	example risk in a risk register . . . . .	139
6.5	example control register . . . . .	140
6.6	risks and controls class diagram . . . . .	140
8.1	conflicts caused by uncontrolled software changes . . . . .	144
8.2	the ripple effect caused by a single change to the requirements specification. . . . .	146
8.3	a generic change management process . . . . .	147
8.4	branching and merging during change management . . . . .	149
8.5	working bases and working copies . . . . .	156
8.6	the update, resolve, commit cycle . . . . .	157
10.1	sample structures for two types of technical document . . . . .	176

11.1	the requirements engineering process . . . . .	188
11.2	user requirements for the request book use case . . . . .	189
11.3	example system requirements for the branch library . . . . .	189
11.4	requirements and design . . . . .	190
11.5	methods for requirements elicitation . . . . .	191
11.6	different approaches to structuring interviews . . . . .	193
11.7	a report of an interview with a department administrator . . . . .	194
11.8	revised set of actors for the system . . . . .	195
11.9	actors and the system boundary . . . . .	195
11.10	a use case diagram for some of the the branch library use cases	196
11.11	external systems as actors . . . . .	196
11.12	a timed notification use case . . . . .	197
11.13	request book use case non-functional requirements . . . . .	199
11.14	the new cancel book request use case . . . . .	200
11.15	a pre-mature design decision expressed as a use case . . . . .	201
11.16	museum problem description . . . . .	203
12.1	interview with the Head of Department (HoD) . . . . .	206
12.2	the use cases related to transferring a book to the branch . . . . .	207
12.3	use cases for lending books in the branch . . . . .	208
12.4	use cases for keeper administration . . . . .	208
12.5	non-functional requirements for the branch library system . . . . .	209
12.6	all use cases identified for the branch library . . . . .	209
12.7	moscow rules categorisation of the use cases . . . . .	210
12.8	the primary scenario for the request book use case . . . . .	211
12.9	an alternative scenarios for the request book use case . . . . .	211
12.10	the steps of the primary scenario for the request book use case . . . . .	212
12.11	the steps of the request book use case as pseudo code . . . . .	212
12.12	denoting alternatives in pseudo code . . . . .	213
12.13	denoting repetition in pseudo code . . . . .	213
12.14	the request book use case description in pseudo-code . . . . .	213
12.15	notation for activity diagrams . . . . .	214
12.16	request book use case activity diagram . . . . .	215
12.17	the login use case . . . . .	216
12.18	including login in the request use case . . . . .	217
12.19	extending the view status use case . . . . .	218
12.20	the lend book use case and actor inheritance . . . . .	219
12.21	revised actors for the branch library system . . . . .	220
12.22	pre and post conditions for use cases . . . . .	221
12.23	the component parts of a use case document . . . . .	222
13.1	UML diagram type hierarchy . . . . .	227
13.2	UML classes with un-typed attributes . . . . .	229
13.3	extending classes with types . . . . .	230

13.4	associations and constraints in UML . . . . .	231
13.5	associations and constraints in UML . . . . .	232
13.6	denoting class instances in UML . . . . .	233
13.7	using UML instance diagrams to understand class relationships	234
13.8	refining attribute types during software design . . . . .	234
13.9	using association classes for many-many associations . . . . .	235
13.10	directed associations in UML . . . . .	235
13.11	reflexive associations in UML . . . . .	236
13.12	aggregation decorations of associations in UML . . . . .	236
13.13	composition decoration of associations in UML . . . . .	237
13.14	using compositions for functional propagation . . . . .	238
13.15	using compositions for functional delegation . . . . .	239
13.16	grouping common operations and attributes in abstract classes	240
13.17	inheritance discriminators . . . . .	241
13.18	a bad generalization violating the <i>is a</i> rule . . . . .	242
13.19	making specializations explicit with abstract operations . . . . .	243
13.20	merging excessive specializations . . . . .	244
13.21	using the player-role design pattern to avoid multiple inheritance	245
13.22	denoting interfaces in UML . . . . .	245
13.23	realizing interfaces in UML . . . . .	246
13.24	uses of notes in UML . . . . .	247
13.25	using dependencies and stereotypes in UML . . . . .	248
13.26	mobile forensics application problem description . . . . .	250
13.27	the Sock and Piddle micro-brewery domain description . . . . .	251
13.28	class diagram of a music player . . . . .	251
13.29	an instance diagram showing, newspapers, magazines, cities and countries . . . . .	253
14.1	Object oriented analysis, design and implementation . . . . .	256
14.2	identifying entities in the request book for branch use case . . . . .	257
14.3	branch class CRC card . . . . .	258
14.4	class collaborations . . . . .	259
14.5	UML class diagram . . . . .	260
14.6	revised UML class diagram . . . . .	260
15.1	taxonomy of dynamic UML diagram types . . . . .	264
15.2	sequence diagram notation . . . . .	265
15.3	invoking sequences . . . . .	265
15.4	object construction and deletion . . . . .	266
15.5	self-messaging in sequence diagrams . . . . .	267
15.6	control structures using guards on sequence diagrams . . . . .	267
15.7	references to other sequence diagrams . . . . .	268
15.8	asynchronous messaging on sequence diagrams . . . . .	269
15.9	timing constraints on sequence diagrams . . . . .	271

15.10	communication diagram for the request book use case . . . . .	272
15.11	classes developed from a communication diagram . . . . .	273
15.12	moving from use cases to architecture . . . . .	274
15.13	the lend book use case . . . . .	275
15.14	initial conceptual class for the lend book use case . . . . .	275
15.15	the initial lend book sequence diagram . . . . .	276
15.16	the selectLoan message . . . . .	277
15.17	the login sequence diagram . . . . .	278
15.18	the revised selectLoan message . . . . .	279
15.19	the revised login sequence diagram . . . . .	280
15.20	the selectBook sequence . . . . .	280
15.21	the revised selectBook sequence . . . . .	281
15.22	the complete lend book sequence diagram . . . . .	282
15.23	extended communication diagram . . . . .	283
15.24	revised class diagram extended with method names . . . . .	284
16.1	overview of the prototyping process . . . . .	290
16.2	overview of the throw away prototyping process . . . . .	291
16.3	using prototypes during testing . . . . .	293
16.4	prototyping process for evaluating new technology . . . . .	295
16.5	the netbeans GUI designer . . . . .	296
16.6	throw-away vs incremental prototyping . . . . .	299
16.7	longevity of different types of prototype . . . . .	299
17.1	slots and hooks in frameworks . . . . .	305
17.2	packages in the Java Swing GUI framework . . . . .	311
17.3	a basic swing window . . . . .	312
17.4	a window containing two buttons . . . . .	314
17.5	attaching a tool tip to a widget . . . . .	315
17.6	a simple menu bar . . . . .	316
17.7	the AWT and Swing class hierarchy . . . . .	317
17.8	example of using the FlowLayout layout manager . . . . .	318
17.9	using the grid layout manager . . . . .	319
17.10	the BorderLayout layout manager with a tool bar . . . . .	321
17.11	example of the swing event model . . . . .	323
17.12	the event object information GUI . . . . .	324
17.13	action listeners as inner classes . . . . .	324
17.14	using interface . . . . .	328
17.15	derived widget classes with self contained listeners . . . . .	328
17.16	capturing events from multiple sources . . . . .	330
17.17	instance diagram of MultipleSources . . . . .	331
17.18	a simple dialog . . . . .	333
17.19	the service oriented architecture pattern . . . . .	335
17.20	setting up Eclipse with Axis . . . . .	338

17.21	a generic remote service accessor . . . . .	341
18.1	repeated data values in instance attributes . . . . .	351
18.2	the abstraction occurrence design pattern . . . . .	352
18.3	applying abstraction occurrence to lectures . . . . .	353
18.4	extending the abstraction occurrence pattern . . . . .	353
18.5	a complex inheritance hierarchy . . . . .	354
18.6	the player-role design pattern . . . . .	355
18.7	applying the player-role pattern to mortgages . . . . .	356
18.8	a software re-use problem . . . . .	356
18.9	the adapter design pattern . . . . .	357
18.10	applying the adapter pattern to support re-use . . . . .	357
18.11	transparently inserting method call logging . . . . .	358
18.12	the proxy design pattern . . . . .	359
18.13	applying the proxy pattern . . . . .	359
18.14	the Java reflected proxy mechanism . . . . .	360
18.15	high coupling in a vehicle control system . . . . .	364
18.16	the facade design pattern . . . . .	364
18.17	applying the facade pattern . . . . .	365
18.18	a highly coupled design with potential for generalization . . . . .	366
18.19	the composite (or general hierarchy) pattern . . . . .	367
18.20	applying the composite pattern to the calendar problem . . . . .	367
18.21	a user management class . . . . .	369
18.22	the read-only interface design pattern . . . . .	369
18.23	separating privileges on the user class . . . . .	370
18.24	two example singleton problems . . . . .	371
18.25	the singleton design pattern . . . . .	372
18.26	enforcing singleton behaviour on the university and glossary . . . . .	372
18.27	problem . . . . .	373
18.28	the factory design pattern . . . . .	374
18.29	applying the factory pattern . . . . .	374
18.30	the iterable-iterator design pattern . . . . .	376
18.31	the Hashtable-HashIterator relationship . . . . .	376
18.32	re-using functionality from another class . . . . .	377
18.33	the delegation design pattern . . . . .	377
18.34	applying the delegation pattern to the winner problem . . . . .	378
18.35	chaining delegation across several classes . . . . .	378
18.36	the evacuation alert system problem . . . . .	379
18.37	the observer-observable design pattern . . . . .	379
18.38	applying the observer-observable pattern to the alert system . . . . .	380
18.39	a domain model for the mobile forensics application . . . . .	385
19.1	the relationship between a system and its environment . . . . .	387
19.2	software system design terminology . . . . .	388

19.3	the cost-functionality-quality compromise in design . . . . .	389
19.4	the design process . . . . .	390
19.5	design strategies . . . . .	391
19.6	separation of concerns in an SVN log parsing framework . . . . .	393
19.7	extending the alert system . . . . .	396
19.8	re-organising a design to support testing . . . . .	398
19.9	generic layered information system architecture . . . . .	400
19.10	a layered design for a messaging system . . . . .	401
19.11	instance diagram of a layered architecture for a messaging system	402
19.12	data cohesion . . . . .	403
19.13	the CaptureConfigPanel widget . . . . .	404
19.14	the ImageIO utility class . . . . .	406
19.15	reducing coupling in a software design . . . . .	407
19.16	incomplete delegation in an abstraction-inheritance pattern . .	408
19.17	decoupling Club from ScoreBoard . . . . .	408
19.18	content coupling when selecting a side . . . . .	409
19.19	placing responsibility for side selection in the Team class . .	410
19.20	common coupling in the sportster application . . . . .	410
19.21	caching in the mobile forensics application . . . . .	411
19.22	the bi-plane bomber video game – very retro . . . . .	413
19.23	using a lookup table to de-couple commands, operations and state . . . . .	414
19.24	using a Mailable interface to abstract mailing information . .	417
19.25	food order system . . . . .	421
21.1	effectiveness vs. efficiency in testing . . . . .	428
21.2	slip, defect, failure model . . . . .	429
21.3	the software testing process . . . . .	429
21.4	an implementation of the BinaryToDecimal operation . . . . .	436
21.5	unit testing and stubs . . . . .	444
21.6	example integration sequence . . . . .	446
21.7	the test-fix cycle . . . . .	447
21.8	integration testing exercise . . . . .	453
22.1	combinatorial explosion in software testing . . . . .	460
22.2	documenting acceptance tests . . . . .	470
22.3	documentation test cases for non-functional requirements . .	471
23.1	a simple state transition network for a light switch . . . . .	477
23.2	the TCP connection establishment sequence . . . . .	477
23.3	TCP connection establishment STN . . . . .	478
23.4	state explosion in an STN . . . . .	479
23.5	using local variables in STNs . . . . .	479
23.6	STN with recursion . . . . .	480

23.7	the state machine diagram notation . . . . .	481
23.8	a state machine diagram for a DVD player tray . . . . .	482
23.9	comparing STNs and state machine diagrams . . . . .	482
23.10	concurrent states and broadcast events . . . . .	483
23.11	TCP connection establishment state model . . . . .	484
23.12	an extended DVDTray state machine diagram . . . . .	485
23.13	domain model for Book and BookCopy . . . . .	487
23.14	Book use cases . . . . .	487
23.15	a sequence diagram for the request book for branch use case . . . . .	488
23.16	a sequence diagram for the process request use case . . . . .	488
23.17	an initial state machine diagram for BookCopy . . . . .	489
23.18	the record receipt of book sequence diagram . . . . .	490
23.19	the final BookCopy state machine diagram . . . . .	491
23.20	Revised BookCopy class incorporating state behaviour . . . . .	492
23.21	Harel's watch . . . . .	494
23.22	the Booking class . . . . .	495
23.23	the Booking class . . . . .	495
23.24	thread architecture in Java . . . . .	497
23.25	extending the thread class . . . . .	498
23.26	implementing the Runnable interface . . . . .	500
23.27	creating an anonymous thread class . . . . .	501
23.28	deadlock defects . . . . .	506
25.1	the formal methods development process . . . . .	513
25.2	running example – coursework management system. . . . .	519
25.3	placing constraints directly on the class diagram . . . . .	520
25.4	navigating the class diagram . . . . .	522
25.5	model driven development . . . . .	530
25.6	library system . . . . .	532
25.7	project proposals . . . . .	533
25.8	voting system . . . . .	534
25.9	mountain walks system domain model . . . . .	535
25.10	the OCLE application after start up . . . . .	538
25.11	the OCLE environment after loading the StudentLifeElipse project	539
25.12	the StudentLifeElipse class diagram . . . . .	540
25.13	the UoStGlasburgh instance model . . . . .	542
25.14	the StudentLifeElipse constraints file . . . . .	543
25.15	an error reported in the UoStGlasburgh model . . . . .	544
26.1	cost estimates of software maintenance between 1980 and 2006	548
27.1	recovering structure using StarUML . . . . .	556
27.2	Eclipse TPTP Profiler summary view . . . . .	557
27.3	Eclipse TPTP Profiler call tree view . . . . .	557

27.4	Eclipse TPTP Profiler call tree view . . . . .	558
28.1	the refactoring process . . . . .	564
28.2	Brooks on self-documenting code . . . . .	569
28.3	the Book and Author classes . . . . .	570
28.4	the revised book class and companions . . . . .	574
29.1	Deciding how to change a legacy system . . . . .	587
29.2	risk vs value spectrum in software maintenance . . . . .	588
29.3	A poorly structured legacy system architecture. . . . .	591
29.4	A revised structure for the legacy system architecture. . . . .	592
29.5	software wrappers during migration . . . . .	593
29.6	software gateways during migration . . . . .	594
29.7	multiple gateways in a layered architecture . . . . .	595



# **Part I**

# **The Software Engineering Process**



# Chapter 1

# Introduction to Software Engineering

## Recommended Reading

PLACE HOLDER FOR lethbridge05object

### 1.1 Software Failures

Software development is hard. Or at least, the development of high quality, defect free software, which really solves a customer's problem is hard.

To illustrate this bold statement, consider the following examples. Figure 1.1 is a montage of small scale software application failures manifested at the respective application's user interface. The email application on the left hand side is reporting that the user has 2,147,483,625 unread messages. That's a rather heavy work load for the day. The bottom middle picture is taken from a cash machine, giving the user the opportunity to close the user facing application. The middle picture, a classic Windows blue screen of death was taken in an airport departures lounge. The top picture was captured from Firefox - I had no idea that a browser could be too new for the security suite. The picture on the bottom right (Thunderbird) is helpfully informing us that undefined, is well, undefined.

The examples were taken from a diverse range of bespoke, commercial off the shelf and open source applications. You can find plenty more on the Daily Worse Than Failure website.<sup>1</sup> None of the failures are particularly critical, but they give an unfavourable impression of the quality assurance procedures for the different applications (whether fair or not).

The picture in Figure 1.2 was taken in a Scottish railway station. I don't know the full back story to this, but it would seem that the newly installed

Failing small (6)

---

<sup>1</sup><http://thedailywtf.com/>



Figure 1.1: failing in the small

booking software was causing the staff some problems. The sign suggests that making bookings with the new system actually takes longer than with the old one. This could be one of several problems. For example, staff training requirements may not have been properly established before the new software was installed. Alternatively, the performance requirements for the system may not have been properly specified, or were not tested and satisfied prior to deployment. Understanding the real requirements for a software system is crucial to building the right system.

#### Failing medium (7)

Finally, Figure 1.3 illustrates one of the user interface screens for the e-counting system that was used to process paper ballots for the general election in Scotland in 2007. The figure shows the part of the system used to validate optical character recognition results from the ballot scanning software. The user was required to indicate in the boxes running down the outside of the paper image, which party had received a vote. In this case, the vote is for the Scottish National Party.

#### Failing big (8)

The figure shows a small failure: when the ballot paper was scanned the crease where it was folded was picked up as a dark horizontal line. The inset picture shows the line more clearly. The optical mark recognition software has treated the fold line as the top of the ballot paper and aligned the selection boxes on the screen accordingly. It isn't clear whether the boxes presented on the screen still represent all the possible candidates, or whether it isn't possible to confirm the correct vote in this case.

This particular problem represents a far larger range of problems which were encountered on election night. Throughput of ballots was much less than expected, causing delays into the following morning. Ballot scanning machines frequently jammed or rejected ballot papers and had to be taken apart and cleaned. The database used to store electronic ballot images and vote records was not correctly configured, so that officials were unable to announce results of



Figure 1.2: failing medium

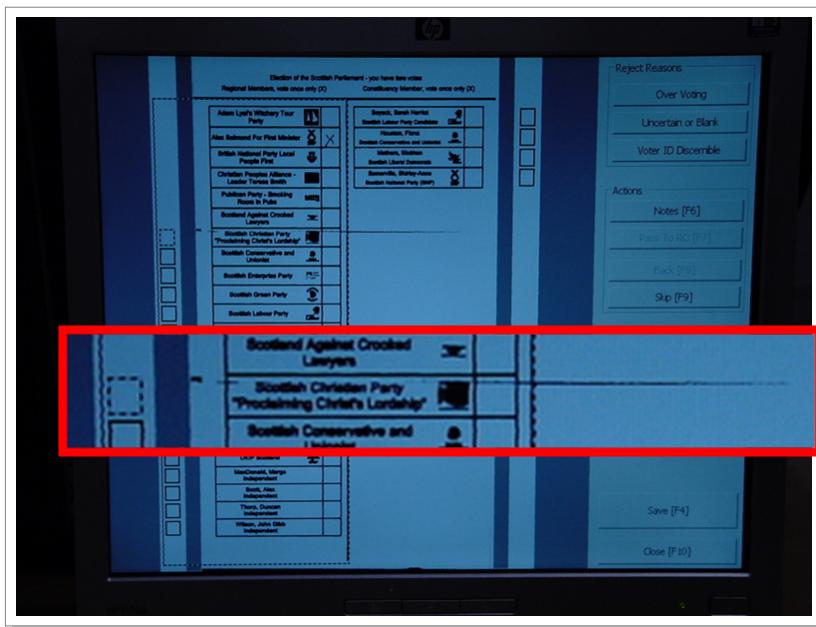


Figure 1.3: failing big

the election until the following day. Candidates and their representatives were sidelined in the vote auditing process and began to question the accuracy of results.

## 1.2 What Makes Software Development Hard?

Writing software seems to be fraught with difficulties and insurmountable challenges. Even projects that can be characterised as successful have residual defects that users are required to cope with. For any non-trivial program (and for many trivial ones) we seem unable to build defect free software programs, or even fully understand the original problem. Even worse, the problem can often change while we are attempting to address it.

How can we explain this situation? In all the illustrations above, the development teams were almost certainly concerned with ensuring the success of their product. However, the complexity of software and software development makes the task extremely difficult to get right first time, particularly for very large systems such as the e-counting system. Brooks [1995] divided these obstacles into accidental and intrinsic characteristics of software development:

- accidental characteristics can be overcome over time through the development of better software development practices or tools. They are a feature of how software development was done in the 1970s, rather than how it could be done;
- intrinsic characteristics can only ever be mitigated by the development of new tools or practices, but can never be completely overcome.

Essential  
characteristics of  
software development  
(9)

Software development is intrinsically challenging because:

- software is intangible, complex and consequently has many different representations. A software developer never 'sees' the program they are working on in the same way as the builder of a house, or an architect of a bridge. Whereas these engineers utilise models to assist their construction efforts, software developers rely on different representations of software for all their observations of and interactions with a program;
- software is malleable and can be readily changed to implement new features or fix defects. This makes it tempting for customers and developers to constantly change the specification for software as it is developed;
- many software projects address large scale, or even societal problems. Such projects have to be completed by multiple developers organised into a development team or teams because the work is too much for one person to complete in a realistic time scale. Large software projects require coordination between individual developers, teams and even whole organisations; and

- software development is constrained by multiple pre-existing development efforts which have resulted in legacy systems; and co-existing development efforts that change the environment even while the development team works. These development efforts are often hard to identify and manage.

Despite these challenges, many software projects are successful, delivering considerable value to the end customer. Consequently, we need to understand the more successful approaches to software development in order to reduce the risk of failure in our own projects.

### 1.3 What is Software Engineering?

An engineer is a professional who has the tools and methods necessary to identify and address the real needs of a customer. The approach adopted by an engineer will depend on the nature of the problem, but should be based on well founded scientific principles. Fundamentally, engineering is concerned with implementing a development process that has low risk of failure, about which confident process schedule and costs predictions can be made. Engineers in different disciplines are often regulated by a professional body, such as the IEEE<sup>2</sup> for electrical and electronic engineers.

The notion of professionalising the role of the software developer stems from two NATO sponsored conferences in the late 1960s Naur and Randell [1968]. The attendees were concerned with what they called the '*software crisis*': the inability of software development teams to deliver functionally correct, high quality software systems on time and within budget. Many of the challenges facing software developers then are still characteristic of software projects today.

Software engineering is concerned with developing the tools and methods necessary for the professional development of software systems. The British Computer Society<sup>3</sup> is the professional body for software engineers in the UK, however, there is no requirement for practicing software developers to be licenced by it. One way to think of software engineering as a profession is as a work in progress.

Just as for many other engineering disciplines there are many types of software projects:

- information systems are used by organisations to manage business processes. They comprise a database, business processing logic and a user interface for presenting information to the user. Information systems can often be built by configuring existing frameworks or platforms to suit individual needs;

Types of software project and engineering specialisms (10)

---

<sup>2</sup><http://www.ieee.org/>

<sup>3</sup><http://www.bcs.org/>

- data processing systems used in scientific computation applications, for example. These application handle very large datasets to produce complex analyses.
- real time control systems are required to process inputs from sensors embedded in their environment and then effect a change in a timely manner. Real time systems are often needed when the reactions of a human user would be too slow to safely control the system. Examples include auto pilot systems in aircraft, the assisted braking systems in many cars and flight control software in space craft;
- embedded systems operate on micro-processors with restricted memory, battery and processing power. Such systems are present in many household appliances (fridges, washing machines, microwave ovens), cars, industrial plant machinery, watches, mobile phones, mp3 players and wireless sensor network nodes. An embedded system developer is highly constrained by the resource limitations of the host device; and
- distributed systems utilise multiple processors connected by a network to perform computations. The output from sub-processes must be coordinated, gathered and reconciled with varying network reliability.

Of course, many software projects have characteristics of more than one of these domains. Characterising project type helps in thinking about the risks associated with the project.

## 1.4 Project Value and Quality

All software projects are constrained finite resources of time and money, and the benefit of the project to the customer must, in the end, outweigh these costs. The benefits to customers, depends on the nature of the software market for which the project is developed, for example:

- bespoke software is developed for a single customer to address a particular need. Historically, bespoke projects were the dominant form of project development and were largely developed from scratch each time;
- Enterprise Resource Planning (ERP) developments utilise frameworks and platforms designed for a particular domain of information system (e.g. university management or accounting) for much of the functionality required. The framework is configured and sometimes customised to meet the specific needs of the organisation. In addition, the business practices of the organisation are often adapted to fit the new system;
- Commercial off the Shelf (CotS) software products are developed for distribution and/or sale on the open market in order to satisfy an unmet

demand. COTs software may also be used as components in larger software projects, where it is cheaper to acquire pre-developed components, rather than attempt to re-implement them. Typically, the developer will release the binary version of the software for sale, but withhold the source code and other artifacts; and

- Open Source Software (OSS), which is developed in a public environment. The source code for the project is published by the developers and available for further development and alteration by others. Development teams are often a distributed mixture of volunteers and professionals seconded to the project by a software company. Value is often extracted from the project through alternative mechanisms, such as offering support services or selling 'premium' versions of the software.

Regardless of the particular business model for a software project, one of the most important tasks for an engineer is to ensure that the products they develop meet appropriate standards of quality within the constraints of the available resources. Higher quality delivers greater value to the customer (and is thus a more attractive investment) because they exert less effort 'working around' defects and coping with failures.

For software development, high quality means software that has been:

High quality software (12)

- well specified, to meet the *real* needs of the customer. Establishing requirements means that the software development team understand the problem domain for the proposed project and the constraints imposed on the system;
- well designed, demonstrable if the software exhibits high functional cohesion and low coupling. This means that the dependencies between components in the system have been minimised;
- well documented both in the source code and associated documentation, such as requirements specification, design document and test plan;
- well tested through the development of a comprehensive test plan and accompanying automated test suite; and
- well managed, through the use of management tools such as version control, configuration and build management, and project tracking.

Software quality can be measured in a variety of ways, from observations of the development process and of the resulting software system. For example:

Some software quality metrics (13)

- mean time to failure;
- defects per line of code (LOC);

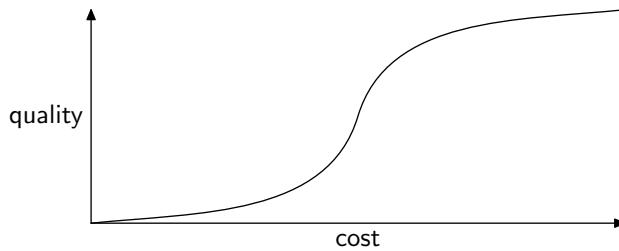


Figure 1.4: quality as cost increases

- documentation coverage;
- test suite coverage;
- rate of coupling;
- throughput; and
- response time.

There is, of course, a necessary trade off between resource constraints and product quality: the higher the quality demanded for a software product the longer it will take to deliver and the more it will cost. Figure 1.4 illustrates a general relationship between software quality and cost. Initial improvements in cost can be achieved relatively cheaply. However, further improvements in quality require exponentially increasing resources, reducing the return on investment.

Quality-cost trade off  
(14)

The task of a professional software developer is to set appropriate quality standards for the project. However, this can be a difficult judgement to make. A key aspect of this is the expected life time of the software in the market place. The longer a software product is returning value to customers, the easier it is to obtain and justify investment in quality. In the long term, the value of the system may be reduced if quality standards are not maintained, because the software will become increasingly difficult to manage and evolve. However, short term projects (sometimes called throw-away projects) are generally not worth investing substantial effort into software quality, because there will not be enough time to recover the invested resources over the system life time.

short vs. long term  
quality (15)

The problem is telling the difference. Projects that begin as short, throw-away efforts may have an unexpectedly long life time. Initially, it may not be perceived as beneficial to invest in quality for a project with a short lifetime. However, if the system development continues beyond its expected lifespan, a failure to invest in quality early can make later investment more expensive.

Most software projects are now evolutionary developments of pre-existing systems, rather than 'clean start' and fragments of software can have an extremely long shelf life. There are numerous software 'war stories' of developers

returning to projects after a long absence, only to discover their temporary ‘hacks’ are now tightly integrated into the development.

## 1.5 Software Engineering in the Small

In many of the chapters in these notes, will be looking at the software engineering as it is practiced ‘in the small’. There are several themes that will run throughout (and are inter-linked):

- graphical modelling and analysis in order to understand the problem domain using the Unified Modelling Language (UML) notation;
- software design and quality assurance to improve the reliability of the software we produce;
- computer assisted software engineering (CASE) tools for automating software development tasks wherever possible; and
- professional practice.

Themes in this Course  
(16)

We will look at the disciplined methods, practices and tools that can be adopted by an individual software engineer, working by themselves or as part of a larger team in order to develop and assemble components into high quality systems.



## Chapter 2

# The Object Oriented Paradigm and Java

### Recommended Reading

PLACE HOLDER FOR lethbridge05object

#### 2.1 Introduction

The history of software engineering has been one of increasing abstraction coupled to the increasing capabilities of computers to arrange and perform tasks for themselves.

An early development in software engineering was the idea of structured programming, which partitioned the behaviour of programs into discrete procedures or functions. The development meant that programmers could rely on a function to exhibit some effect on program data, without knowing precisely how it was implemented. The object oriented paradigm extends this notion by encapsulating the data together with corresponding behaviour that will manipulate it.

In the object oriented paradigm, a software system is structured as a collection of encapsulated objects - everything in the program at runtime is an object. The overall behaviour of the system arises out of the interactions between the individual objects. In this chapter we will explore the object oriented paradigm in detail, and its realization in the Java object oriented programming language.

#### 2.2 Objects and Classes

An *object* is a self contained entity in an *object oriented environment*. Each object has identity, state and services. The specification of an object is described by its *class*, including the state attributes maintained by every object and the

objects	classes
an identity for <i>addressing</i> itself and other objects	type name shared by all objects of the same class
space for storing the <i>attribute values</i> (the object's <i>state</i> ), which are either simple primitive values or references to other objects	attributes which specify what values and references an object can store. The attribute label allows the attribute value to be addressed. The <i>attribute type</i> specifies what sort of data can be stored by the attribute
a prescribed set of <i>services</i> that give rise to the object's behaviour. State changes in an object occur as a result of communication between objects, via <i>messages</i> to the services.	operations which implement the behaviour of objects. An <i>operation signature</i> specifies how a message must be formatted to invoke the operation on the object.

Table 2.1: objects vs classes

"a company sells a system for managing sports leagues. Each league has a number of teams (e.g. football), or individual players (e.g. singles tennis). Teams consist of a number of players. Each team plays other teams in matches during a season."

Figure 2.1: an initial problem description for sports league management system

Objects vs classes (18) services they will offer via operations. Classes can be used to represent and reason about the behaviour of a large number of objects. The relationship between the features of objects and classes is shown in Table 2.1.

How to decide when something is a class or an instance? The course text book, Lethbridge and Laganière [2005], has a rather circular definition that comes down to classes can have instances and objects are members of a class: not very helpful!

An alternative way to think about things is that objects are representations of the 'actual' things in the real world with state attributes that change over time. Classes are abstract descriptions (with a label) which capture the things you want to record about a collection of objects. An object could actually belong to several different classes, depending on what you want to say about it, but it will always 'be' an object.

Categorising things as objects and classes then depends on what you want to model in the *problem domain*. A good way to pick out classes to model for a problem is to identify the nouns. Consider the example problem description in Figure 2.1.

The following classes can be identified from the problem (with corresponding examples of instances):

- football, cricket, rugby:Sport

- premierLeague, ashesTour, sixNations:SportsLeague
- stJohnston, england, scotland:Team
- gerrard, cook, thompson:Player
- englandVsAustralia:Match
- winterTour2011:Season

Notice the convention for naming classes. The first letter of each word in the classes is capitalised. The first letter of each word except the first is capitalised in instance names. A colon is used to separate instance names (on the left) from class names (on the right). So the first bullet point can be read as: “football, cricket and rugby are all instances of sport.”

## 2.3 Types and Members

### 2.3.1 Types

Every data value in an object oriented environment has a type. Types are categorised as either primitive for simple values such as booleans, numbers or character, or *classes* for things which reference objects instances.

Some types occupy a ‘fuzzy’ middle ground between primitives and references, in that they are implemented as classes, but typically represented as primitives in object oriented models. Examples include String and Date.

Types in an OO environment (21)

- primitive types
  - **booleans**, numbers (**int, double...**) and characters (**char**)
- reference types
  - Student, Match
- ‘quasi’ primitive types
  - String, Date, List

The array type is another ‘quasi’ primitive type in Java. Arrays are sequences of primitives or object references that can be referred to by an index. For example:

Array examples (22)

```
// an array of integer values
int [] someIntegers =
  new int []{1,1,2,3,5,8,13,21,34};
```

```

//a String array of length 4
String[] someStrings = new String[4];
someStrings[2] = "Charlie";

//an array of arrays of Objects (2D array)
Object[][] someObjects = new Object[5][];
someObjects[0] = new Object[6];
someObjects[0][0] = "the";
someObjects[0][1] = new Integer(2);
someObjects[0][3] = 4.0;

```

Features of arrays include:

- a fixed length, accessible via the **final** attribute `length` attribute set when the array is instantiated;
- either literal declaration of initial values, or declaration by length;
- multi-dimensional extensions.

Where possible, it is preferable to use the Collections framework to manage aggregations of object references in Java. The array type is a concrete implementation of aggregation. Using a combination of Collection classes and frameworks allows the implementation specifics of a store to be abstracted away from an application program. We will look at the features of the Collections framework later on in Section 18.4.1.

### 2.3.2 Instance Members

Classes specify the *attributes* and *operations* of a class, which are collectively known as the class' *members*.

Instance attributes define the possible state values for instances of a class, and have an *identifier* label and a *type*. The label for attributes are denoted in a similar way to classes, except that the first letter is not capitalised. The Player attributes (23) attributes of the Player class might be:

- `surname:String`
- `forenames:String`
- `age:int`
- `team:Team`
- `fitness:Boolean`

Variables vs. objects  
(24)

Notice that variables are not the same as objects. Variables are *references* to objects:

- a variable may refer to many different objects during the life time of an application; and
- several different variables may refer to the same object.

Try sketching the state of this program after each line:

```
String s1 = new String();
String s2 = s1;
String s3 = s2;
s1 = new String();
s1 = s1.toString();
```

Operations specify the behaviour of object instances of a class. Invoking an operation means that the state (the attributes) of an object instance could be altered.

Operations (25)

- an *identifier* label. Several different operations may have the same identifier, however each one must have a unique list of parameters;
- a *return type* of the value to be passed back to the invoking instance when the operation is complete. The return type can be any of the types described above, as well as the special **void** type, which means nothing is returned; and
- a list of *parameters*, each of which is a variable with an identifier and a type. When an operation is invoked, a set of *argument values* is passed to the target object. The type of the argument values must match the type of the specified parameters.

Some of the operations of the Player class might be:

```
isFit():Boolean
setRanking(int:Ranking):void
getMatchRecord(sport:Sport):List<Match>
```

All members of a class have a *visibility* which describes whether they can be accessed from other members outside of instances of the class. The Java member visibility modifiers are:

Member visibility (26)

- **private** members which are only accessible from other members of the same class;
- default members (no explicit modifier) are accessible by members of classes in the same package;
- **protected** are accessible by members of classes in the same package and by members of sub-classes; and

- **public** members are globally accessible. The public members of a class provide its *application programming interface* (API), which specifies how other objects can interact with instances of the class.

It is often tempting to make all members of a class public in order to get a program to compile. However, this can often lead to software with complex dependencies and violates the object oriented principle of *information hiding*. That is, as much of the internal implementation of an instance class as possible should be *encapsulated*.

Encapsulation hides implementation detail from other software developers, allowing them to concentrate on what behaviour an object provides, not how it does it. Encapsulation also makes future changes to a class easier, since other parts of the program do not depend on the internal implementation details. Information hiding is a form of *loose coupling*, which we will return to in Chapter 19.

Some members of a class have to be visible in some form, otherwise it wouldn't do much at all! However, as a general rule, all members of a class should initially be **private**. Visibility modifiers can be relaxed for members once it is established that they have to be accessible to other classes as part of the class' API.

### 2.3.3 Class Members

We have seen how instance members can be used in a class to define the state and behaviour of class instances. *Class members* are attributes and operations that are shared between all instances of a class. In Java, a class member is denoted by the **static** keyword after the visibility modifier and before the type, for example:

```
public static String aStaticString =
    "a static string";

private static void aStaticMethod(){
    /*method body*/
}
```

Class attributes are often used to define *constants* related to that class. For example:

- the `java.awt.Color` class defines a number of standard colors as constants (of type `Color`, including:

`Color.red, Color.RED and Color.BLUE;`

- the `java.util.Locale` class, which is used to contextualise applications use of things like language and currency. The class has a number of pre-defined `Locale` class attributes, including:

Class members (27)

Example class members (28)

Locale.CANADA, Locale.FRANCE and Locale.UK.

Notice the naming convention for class attributes (all capital letters) is the same as for constants in many other programming languages.

Class operations are used to provide behaviour that is independent of the state of class instances. For example:

- the String class has a number of methods for converting values of other types into strings:

```
String b = String.valueOf(true);  
//b has value "true";
```

```
String i = String.valueOf(-45);  
//i has value "-45"
```

- the java.util.Collections class contains methods for manipulating instances that implement different types of Collection. For example:

```
public static sort(list>List<?>):void
```

```
public static shuffle(list>List<?>):void
```

provide methods for sorting and shuffling a list of objects.

It can be difficult to decide whether an attribute or an operation belongs to a class or its instances. A good rule of thumb is that:

- if an attribute is shared between objects it is a member of the class; and
- if an operation does not need to access or alter an individual object's state, then it is a member of the class.

Distinguishing class  
and instance members  
(29)

One thing to be aware of is class operations that are passed instance attributes of the same class as an argument when they are invoked. It may be better to associate the operation with the class instances.

## 2.4 Polymorphism, Inheritance and Abstraction

In Java, operations are implemented as *methods*, which contain the code to implement the operation's behaviour. Several different methods may implement the same operation.

Figure 2.2 illustrates this concept for the Rateable interface. The interface solves the problem of implementing ratings for different types of participants in sports. The class diagram (see Chapter 13) illustrates the Rateable interface and two implementing classes Player and Team. The realization relationship between the classes and the interface is shown by dashed line with a solid arrow

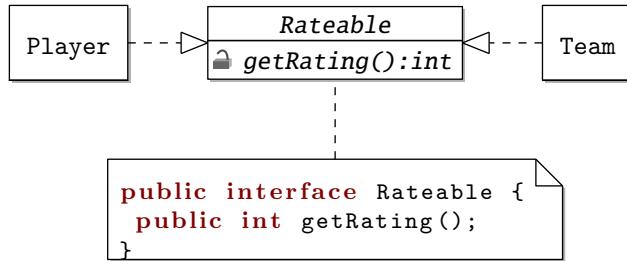


Figure 2.2: Rateable interface of the sportser application.

head in the direction of realization. The Java code specifying the interface is also shown in a note.

The implementation of the `getRating()` method for `Player` and `Team` is shown below. The `Player.getRating()` method is an atomic method that simply returns the current value of the `rating` attribute for a `Player` instance:

```

public class Player
implements Rateable{
private int rating;

@Override
public int getRating(){
    return rating;
}
  
```

However, the implementation of `Team.getRating()` aggregates the rating of all the players in the team.

```

public class Team
implements Rateable{
private Set<Player> players;

@Override
public int getRating(){
    int rating = 0;
    for (Player player : players){
        rating += player.getRating();
    }
    return rating/players.size();
}

protected Club club;
protected Set<Side> sides;
}
  
```

In Java, interfaces are a special type that specify the operations that must be implemented as methods in one or more classes that *realize* the interface.

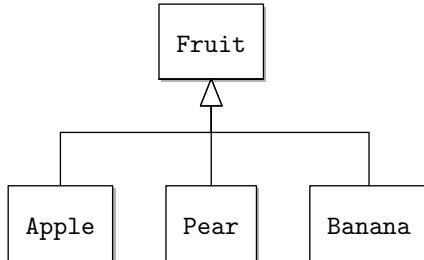


Figure 2.3: fruit inheritance relationships

'Realizing an interface' also means that instances of that class can also be treated as instances of the interface. Multiple implementations of the same interface can have different non-functional characteristics while still providing the same functionality.

For example, the `java.util.Map` interface specifies *operations* for:

`Map.put(key:Object,value:Object):void`

`Map.get(key:Object):Object`

Interface polymorphism:  
example (32)

These are implemented as *methods* in the `java.util` package `HashMap` and `TreeMap` classes. The code below shows the use of the `Map` interface with the two different concrete implementations.

```

Object key = new Integer(4);
Object value = new String("hello!");

Map map = new HashMap();
map.put(key, value);

map = new TreeMap();
map.put(key, value);

String val = (String)map.get(key);
//val is the "hello" string (notice the casting)

```

Notice that interaction with the concrete implementations occurs via a single variable of the `Map` interface type.

Another type of polymorphism is caused by establishing *inheritance relationships* between classes. Figure 2.3 illustrates three inheritance relationships between concrete classes representing different types of fruits and an abstract, general fruit class.

Inheritance relationships denote that the *sub* class inherits the properties and behaviours of the *super* class. When drawing relationships between classes, make sure that the sub-class can be described as the super class (this is called the *is-a* rule). For example, the statement "an apple *is-a* fruit" makes sense.

Abstraction and Inheritance/Generalization  
(33)

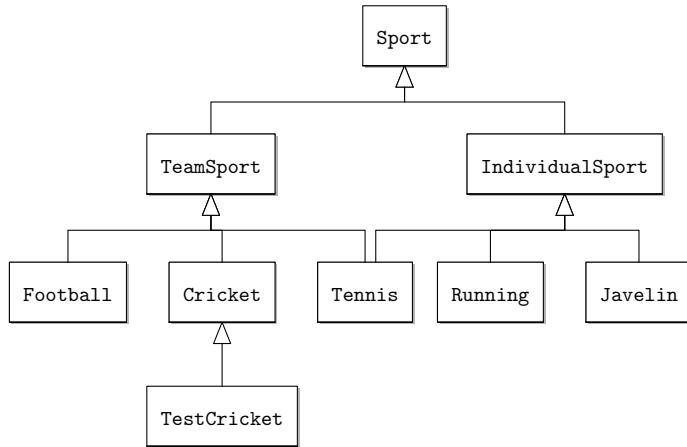


Figure 2.4: sports inheritance hierarchy

Classes can be organised into inheritance *hierarchies*. For example, we might decide to partition the fruit class into different categories of fruit. Also, the Sport class for the sportster system might be refined into a larger hierarchy as shown in Figure 2.4. Notice that the is-a rule is *transitive*. This means that it must hold across chains of inheritance relations in a hierarchy. For example, the is-a relationship must hold between TestCricket and TeamSport, and between TestCricket and Sport.

#### Sport Hierarchy (34)

Some of the operations in a super-class can be over-ridden by sub-classes. This can be useful in providing more specific behaviour to more concrete classes. However, this means we need rules to specify the precedence of method implementations to be selected for execution in a inheritance hierarchy. When a method name is invoked for execution on an instance:

#### Inherited methods, dynamic binding and over-riding (35)

- by default, the concrete method in the lowest sub-class in a hierarchy is executed at runtime;
- **protected** and **public** operations are inherited by sub-classes;
- **abstract** operations must be implemented by concrete classes;
- inherited concrete operations can be *over-ridden* with more specific behaviour in sub-classes; and
- over-ridden operations can be accessed in java using the **super** keyword.

### 2.4.1 Polymorphism in the Collections Framework

The collections framework provides classes which model the properties of different types of abstract data structure. The data structures are used for storing

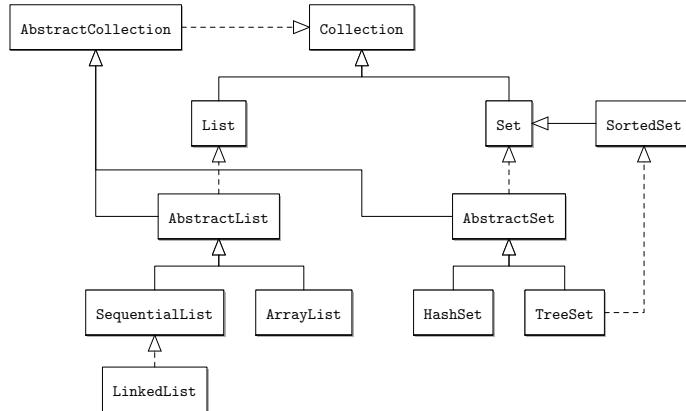


Figure 2.5: the Collection inheritance hierarchy

references to the collection's *elements*. Different types of abstract data structure have different functional properties, and different concrete data structures store elements in different ways and have different non-functional properties. The framework is organised as an inheritance hierarchy, as shown in Figure 2.5.

A *Collection* (also referred to as *multi-sets* or *bags*) is the most general type of abstract data structure:

Example inheritance hierarchy: collections (36)

- any number of objects (of a particular type) can be stored in a *Collection*; and
- the same object can occur more than once in a *Collection*.

The operations that can be invoked on a *Collection* include:

Collection API (37)

```

public void add(Object o)

public void addAll(Collection c)

public boolean remove(Object o)

public boolean removeAll(Collection c)

public void clear()

public boolean retainAll(Collection c)

public boolean contains(Object o)

public boolean containsAll(Collection c)

public boolean isEmpty()

public int size()
  
```

```

public int toArray()

public int toArray(Object[] a)

```

The `AbstractCollection` class provides default implementations for most of these operations. The `AbstractCollection` class can be completed by adding a concrete data structure for storing elements and implementing the `size()` and `iterator()` methods. The code below shows the implementation of a collection (or bag) of sweets (using an array).

Example  
Implementation:  
`BagOfSweets` (38)

```

public class BagOfSweets extends AbstractCollection{

    private String[] sweets =
    {"CHOCOLATE", "COCONUT", "APPLE",
     "CARAMEL", "CARAMEL", "CHOCOLATE", "CARAMEL"};

    @Override
    public Iterator iterator() {
        return new BagIterator();
    }

    @Override
    public int size() {
        return sweets.length;
    }
}

```

Example usage (39)

The `BagOfSweets` class is an *unmodifiable* collection. We can also implement the `add()` and (optionally) `remove()` methods to complete a modifiable collection. The `BagOfSweets` class can now be used just like any other collection.

```

Collection c = new BagOfSweets();
System.out.println(c);
System.out.println(c.size());
Iterator i = c.iterator();
System.out.println(i.hasNext());
System.out.println(i.next());

```

Sample output:

```

[CARAMEL, CHOCOLATE, CARAMEL, APPLE, CARAMEL, CHOCOLATE,
CARAMEL]
7
true
APPLE

```

Different properties of different types of collection are represented by the way the abstract Collection interface is extended, implemented and over-ridden by the different sub-classes.

The `List` interface models a collection of elements with an index. The API specifies additional operations for inserting elements into a list in a particular position and also specifies where elements are to be added to the list for the over-ridden methods from `Collection`.

Extending and  
over-riding  
Collection for List  
(40)

```

public void add(Object o)

public void add(int index, Object o)

public boolean remove(Object o)

public boolean remove(int index)

public List subList(int fromIndex, int toIndex)

...

```

The code below shows an example usage of a list.

Example usage of  
List (41)

```

List list = new ArrayList();
list.add("hello");
list.add("oose2");
list.add("class");
System.out.println(list);
list.add(1,"there");
System.out.println(list);
list = new LinkedList();
list.add("yourself");
list.add(0,"hello");

```

In contrast to a collection, the ordering of elements in a list is significant.  
Sample output:

```

[hello, oose2, class]
[hello, there, oose2, class]
[hello, yourself]

```

Notice again that interaction with the list implementations (the `ArrayList` and `LinkedList` class instances) is done via the `List` interface. The two list implementations are *functionally* equivalent. However, they may have very different non-functional characteristics, that become apparent when using large data sets.

Sets are collections that only contain unique elements. Two elements are considered equal if the `equals(Object e)` method (inherited, or overridden from the `Object` class) returns true.

For example:

- the `String` class checks that the characters at each index in two `Strings` are equal; and
- the `Point` class below overrides the `equals()` method to compare the `x` and `y` coordinates of two points.

Comparing elements  
in a Collection (42)

```

public class Point {
    private int x;
    private int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    @Override
    public boolean equals(Object o){
        if (o == null || !(o instanceof Point))
            return false;
        Point p = (Point)o;
        return p.x == this.x && p.y == this.y;
    }
}

```

Sets are collections that, unlike lists, have at most one element of a given value, and have no prescribed ordering. The code below illustrates the use of the Set interface and its implementations.

```

Set set = new HashSet();
set.add("hello!");
set.add("I");
set.add("said");
set.add("hello!");
System.out.println(set);
System.out.println(set.contains("hello!"));

```

The output from the application might be:

```
[said, hello!, I]
true
```

Notice that the output shows that set implementations don't respect the order of inserts like a list does, and the repeated entries are not added to the set.

All Collection classes in Java implement the Iterable interface. A class which implements Iterable must implement the Iterable.iterator():Iterator operation, which returns a Iterator instance over the members of the class. Any Iterator has the following methods:

```

hasNext():boolean
next():Object
remove():void

```

The remove() method can be optionally implemented, and invoking it may throw an UnsupportedOperationException if the underlying collection is not modifiable.

Example use of the Set interface (43)

Iterating over collections:  
java.util.Iterator (44)

This is an implementation of the *Iterator-Iterable design pattern*, which allows each element in the collection to be inspected in a consistent way, regardless of the underlying collection implementation (see Section 18.4.1 for more information).

The code below shows how to implement an iterator for the `BagOfSweets` class.

```
class BagIterator implements Iterator {
    Random r = new Random(); int count = 0;

    @Override
    public boolean hasNext() {
        return count < sweets.length;
    }

    @Override
    public Object next() {
        if (count >= sweets.length)
            throw new NoSuchElementException(
                "You've eaten them all, you greedy so and so!");
        else {
            int nextSweet = 0;
            do {
                nextSweet = r.nextInt(7);
            } while (sweets[nextSweet] == null);
            count++;
            return sweets[nextSweet];
        }
    }
}
```

Example  
Implementation:  
`BagOfSweets iterator`  
(45)

The iterator can be used in two different loops, either explicitly in a `while` loop, or in a `for-each` loop. The code below illustrates the two different loop styles for the `BagOfSweets` collection.

Iterating over a  
collection (46)

```
Collection bag = new BagOfSweets();

Iterator i = bag.iterator();
while(i.hasNext())
    System.out.print(i.next());

//since Java 5
for (Object sweet: bag)
    System.out.print(sweet);
```

The second style was introduced into the Java language in version 5. The `foreach` loop is similar to the equivalent construct in Python. The approach avoids the programmer explicitly accessing the iterator of the collection.

Example enumeration  
Shape (47)

## 2.4.2 Enumerations

Many sets of objects don't change over the lifetime of an application. This is often the case, for example, with *flags*, which are parameters that have discrete meanings for the behaviour of methods.

In this situation, it may be better to declare the collection as an *enumeration* of constant objects, rather than as a collection. You can think of enumerations as a short hand way of defining an immutable class with a number of static variables assigned to pre-defined (named) instances. The code below shows the implementation of a enumeration for a number of pre-defined shapes:

```
public enum Shape {  
    CIRCLE, TRIANGLE, SQUARE, PENTAGON,  
    HEXAGON, HEPTAGON, OCTOGON, NONAGON, DECAGON  
}
```

The members of an enumeration can be iterated over like a collection:

```
for (Shape s: Shape.values())  
    System.out.println(  
        s.name() + ":" + s.ordinal());
```

All elements of an enumeration have an `ordinal()`, indicating the order in which they are declared in the source code:

```
System.out.println(  
    Shape.valueOf("CIRCLE").ordinal());
```

It may be tempting to use enumerations for all of your classes, by just adding a new member to the enumeration for each new object you need. For example, we could implement the `Color` class with a set number of pre-defined colors (`Color.Blue`, for example) or the `Locale` class as the number of different locales (`Locale.UNITED_KINGDOM`) an application might have to deal with. However, this quickly becomes unwieldy, since in many cases, we don't need to explicitly name all the objects used in an OO program. A good rule of thumb is decide whether a particular class has a finite number of constant, named entities. In this case, it is probably suitable for implementation as an enumeration.

Just like other classes, enumeration types can be given attributes and operations. However, all the instances of the enumeration must be pre-defined and assigned to static variables of the enumeration class. The code below shows how the pre-defined capital cities of different countries are assigned to the respective enumeration instances:

```
public enum Country {  
    UNITED_KINGDOM("London"),  
    IRELAND("Dublin"),  
    FRANCE("Paris")  
    // ...  
};
```

Enumerations with  
attributes (48)

```

private String capital;
public String getCapital(){return capital;}

private Country (String capital) {
    this.capital = capital;
}
}

```

A variant of the *singleton* design pattern is used to instantiate each instance with its attribute values using a private constructor. The operations defined for the enumeration can be invoked in the normal way, for example:

```

for (Country c: Country.values())
    System.out.println(c.getCapital());

```

## 2.5 Annotations

Annotations provide a means of adding semantic meaning to ‘boiler plate’ parts of the code. Annotations are useful for improving software engineering quality because they can be used during the compilation process to statically detect defects and prevent them from occurring at runtime.

There has already been an example of a Java annotation in the sample code for collections above - the `@Override` annotation. The `@Override` annotation stipulates that the labelled method must override a method or operation in a super class. If the method has been mis-named during development, for example, then the compiler will flag an error. Figure 2.6 shows a compile time error reported by Eclipse.

Annotations can also be used at runtime by applications that use *reflection*. We will see more examples of the use of annotations in the JUnit framework later in Chapter 21.

Using annotations to improve software quality (49)

## 2.6 Packages

Packages are used to:

Using packages and imports (50)

- organise classes into sub-systems;

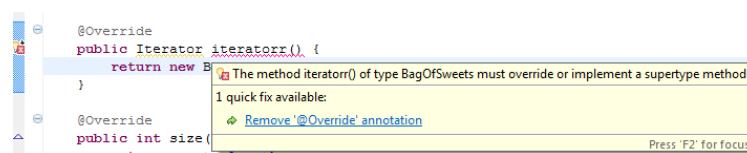


Figure 2.6: a defect in the `BagOfSweets` class statically detected by the `Override` annotation.

- provide a unique name-space for an application and its classes; and
- make the coupling between sub-systems explicit through import statements.

Packages are stored in a directory hierarchy that matches the package name (relative to the source code base), for example:

```
package uk.ac.glasgow.oose2.sportster.cricket;

import uk.ac.glasgow.oose2.sportster.TeamSport;

public class Cricket extends TeamSport{/**/}
```

Package names are often prefixed with the organisational owners of the software. For example, all packages in the sample code in these notes is prefixed uk.ac.glasgow. The JUnit packages are prefixed with org.junit.

Packages are used to improve the cohesion of a software application. We will look at appropriate uses of packages in Chapter 19.

## 2.7 Exceptions

To be useful, software programs must interact with their environment in some way. This interaction can be in the monitoring of sensors and control of motors in a real time system, or something as simple as reading data from a file or network connection. This means that the program must be able to cope with abnormal situations resulting from this interaction, for example:

- ill-formed input data;
- unexpected (e.g. out-of-range) values in input data;
- shortage of memory; and
- unreachable network nodes.

Robust software behaves defensively by anticipating and explicitly accounting for these unexpected events. Figure 2.7 illustrates a general mechanism for managing abnormal situations. The figure illustrates the flow of a program that is monitored by a detector. The detector is used to *trap* or abnormal situations when they arise in the program by monitoring for a particular condition. When the condition is satisfied, the program is temporarily halted and a signal is raised and passed to a handler. The handler manages the abnormal situation before allowing normal program flow to resume (if the situation was not fatal to the program).

*Exceptions* are used in many programming languages to signal when something has gone wrong in a computation. An exception is said to be *raised* or

Abnormal situations  
(51)

handling abnormal  
situations (52)

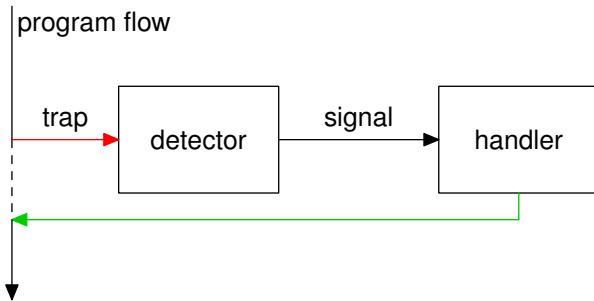


Figure 2.7: general scheme for handling abnormal conditions

*thrown* when an error occurs. Exception mechanisms separate the signalling of errors from the return value of a method or function.<sup>1</sup> Many older languages, such as C, lack an exception mechanism and so can only signal errors in return values (the actual return value is often copied into a buffer specified in the function input, with the return value used to indicate success or failure of a computation).

There are several different strategies that a handler may adopt, depending on the abnormal situation encountered.

- correct the error and resume;
- retry computation (risky);
- try an alternative method;
- record/warn about the error and resume; or
- halt the program (or thread).

Strategies for handling abnormal situations  
(53)

### 2.7.1 The Exceptions Framework in Java

In Java, abnormal situations are managed using the **try/throw/catch/ finally** construct.

- a statement with potential for an abnormal condition is attempted in a **try** block;
- when an abnormal condition is detected:
  - an instance of an particular `Throwable` class is created and is *thrown* (or raised);
  - no further statements in the **try** block are executed;

Java runtime -  
**try/throw/catch/ finally**  
(54)

---

<sup>1</sup>I remember reading a criticism of exception mechanisms which suggest that they have re-introduced the `goto` statement into programming languages, famously considered harmful by Dijkstra[Dijkstra, 1968].

- the exception is *caught* somewhere in a **catch** block somewhere in the method stack; and
- any clean up (whether exception was thrown or not) is then performed in a **finally** block.

Exceptions are objects that are constructed from sub-classes of the Throwable class. The Java code below shows how to implement an exception class by extending Exception.

#### Exception classes (55)

```
public class BarException extends Exception {

    /** supports serialisation */
    private static final long serialVersionUID =
        2781434210711607020L;

    public BarException(String msg/*...*/){
        super(msg);
    }

    public BarException(String msg, Throwable cause){
        super(msg,cause);
    }
}
```

The class defines two constructors. The first constructor takes a String argument which is used to add a message to the exception, describing its cause. This message can be accessed when the exception is caught. The second constructor accepts a second argument for a Throwable instance. This can be used to wrap the underlying cause of an exception. In both cases, the arguments are passed to a constructor in the Exception super class.

If a methods can throw an exception, that must be indicated in the method's signature (except for a small number of, er, exceptions (geddit?!)!). The Java code shown below illustrates how an exception is thrown when an abnormal situation arises in the barThrower() method of the GreatFoo class.

#### Throwing an exception (56)

```
public class GreatFoo {

    private boolean fooBarred;

    public GreatFoo(boolean fooBarred){
        this.fooBarred = fooBarred;
    }

    public void barThrower () throws BarException {
        if (fooBarred)
            throw new
            BarException("FOOBARRED!");
    }
}
```

```

    else System.out.println("Phew, I'm okay");
}
}

```

The `barThrower()` method is declared to potentially throw exceptions of type `BarException`. This forces any client code to explicitly handle for the potential exception in a **try-catch** block. If the `fooBarred` exception is satisfied when the `barThrower()` method is invoked, a new instance of `BarException` is constructed and thrown.

The Java code below shows how to catch the exception.

```

GreatFoo greatFoo1 = new GreatFoo(false);
GreatFoo greatFoo2 = new GreatFoo(true);

try {
    greatFoo1.barThrower();
    greatFoo2.barThrower();
    greatFoo1.barThrower();
} catch (BarException e) {
    e.printStackTrace(System.err);
} finally {
    System.out.println(
        "I always get executed after a try-catch.");
}

```

Catching the exception (57)

Two instances of `GreatFoo` are created, one of which is foobarred, and the other not. The code `barThrower()` method is attempted on `greatFoo1`, `greatFoo2`, and then `greatFoo1` again within a **try** block. If an exception is thrown, it is caught in the **catch** block. In this case, the exception is handled by printing out the method call stack at the time the exception was thrown. Since `greatFoo2` is foobarred, an exception will be thrown at this line. Consequently, the final line of the **try** block will never be executed. The **finally** block is executed regardless of whether an exception was thrown.

The output from the application is shown below.

Output (58)

```

Phew, I'm okay
uk.ac.glasgow.oose2.exceptions.BarException: FOOBARRED!
    at uk.ac.glasgow.oose2.exceptions.GreatFoo.barThrower(
        GreatFoo.java:13)
    at uk.ac.glasgow.oose2.exceptions.Main.main(Main.java:21)
I always get executed after a try-catch.

```

## 2.7.2 Managing Exceptions

A thrown exception is caught in the nearest matching catch block. A match occurs if the argument type to the catch block is the same class or a super class of the thrown exception type. Once a suitable catch block has been found, all further catch blocks are ignored. If an exception is not caught by the application,

it is caught by the virtual machine, which by default kills the thread and prints out the exception's stack trace on `System.err`. This has implications:

- an exception may be caught some distance away from where it was thrown. In the worst case, the exception may be caught by the virtual machine. Some commentators have called the **try-catch-finally** structure a re-introduction of the **goto** statement, famously considered harmful by Dijkstra.
- **try-catch** blocks may trap exceptions thrown from more than one location. This is particularly the case for try catch blocks handling very general or commonly occurring exceptions;
- the code in the **finally** block may cause further exceptions to be thrown, for example, if the block is used to close an I/O stream; and
- catch blocks with more specific exception types should be placed closer to an exception than more abstract types, else the more specific catch block will be unreachable.

## Ordering catch blocks (59)

Consider the following example:

```
public void update(String filePath, String key,
    Serializable value){

    Map<String,Serializable> values = null;
    try {
        File file = new File(filePath);

        if (file.exists()){
            ObjectInputStream ois = new ObjectInputStream(new
                FileInputStream(filePath));
            values = (Map<String,Serializable>)ois.readObject();
        } else
            values = new HashMap<String,Serializable>();

        values.put(key, value);
        ObjectOutputStream oos = new ObjectOutputStream(new
            FileOutputStream(filePath));
        oos.writeObject(values);

    } catch (FileNotFoundException e) {
        System.out.println(
            "Couldn't find file ["+filePath+"].");
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
}
```

```
    }
}
}
```

The following guidelines should be followed when managing exceptions:

- exceptions should be dealt with as close to the source of the **throw** as possible;
- avoid passing the same exception up through multiple stack frames. Instead, wrap lower level exceptions in higher level of abstractions.

For example, the persistent object code could be improved by:

Better (60)

```
public void update(
    String filePath, String key, Serializable value)
throws PersistenceException{
    Map<String,Serializable> values = getPersistenceMap(
        filePath);
    values.put(key, value);
    writePersistenceMap(filePath,values);
}

protected void writePersistenceMap(
    String filePath, Map<String, Serializable> values)
throws PersistenceException{
    ObjectOutputStream oos = null;

    try {
        oos = new ObjectOutputStream(
            new FileOutputStream(filePath));
    } catch (IOException e) {
        throw new PersistenceException(
            "While creating object output stream to file [" +
            filePath+"].",e);
    }

    if (oos != null)
        try {
            oos.writeObject(values);
        } catch (IOException e) {
            throw new PersistenceException(
                "While writing ["+values+"] to ["+filePath+"].",e);
        }
    }

protected Map<String, Serializable> getPersistenceMap(
    String filePath)
throws PersistenceException {
```

```

File file = new File(filePath);
if (file.exists())
    try {
        ObjectInputStream ois =
            new ObjectInputStream(
                new FileInputStream(filePath));

        return (Map<String,Serializable>)ois.readObject();

    } catch (FileNotFoundException e) {
        throw new PersistenceException(
            "Couldn't find file ["+filePath+"].",e);

    } catch (IOException e) {
        throw new PersistenceException(
            "While reading map from file ["+filePath+"].",e);

    } catch (ClassNotFoundException e) {
        throw new PersistenceException(
            "While de-serializing map from file ["+filePath+"].
            ",e);
    }
    return new HashMap<String,Serializable>();
}

```

(Partial) Throwable  
Type Hierarchy (61)

### 2.7.3 The Throwable Class Hierarchy

Although most exceptions *must* be handled for by a Java application, there are some types of abnormal situation that an application is not expected to cope with. These situations either arise from problems in the virtual machine itself (and should in principle never occur), or are the result of programming defects in the application. Figure 2.8 illustrates the throwable class hierarchy which is used to categorise the different types of low level abnormal situation.

The `Throwable` class is sub-classed by application `Exception` classes and virtual machine `Errors`. The `RunTimeException` and the `Error` class hierarchies are shaded to indicate that these classes do not have to be explicitly managed in a `try-catch` block. `RunTimeException` exceptions encompass programming defects that have resulted in an abnormal situation, such as attempting to de-reference a null field or variable, or using an array index of less than zero or greater than the length of the array. `VirtualMachineError` classes encompass fatal situations which will result in a program halting, due to problems in the virtual machine, if, for example, the virtual machine runs out of memory.

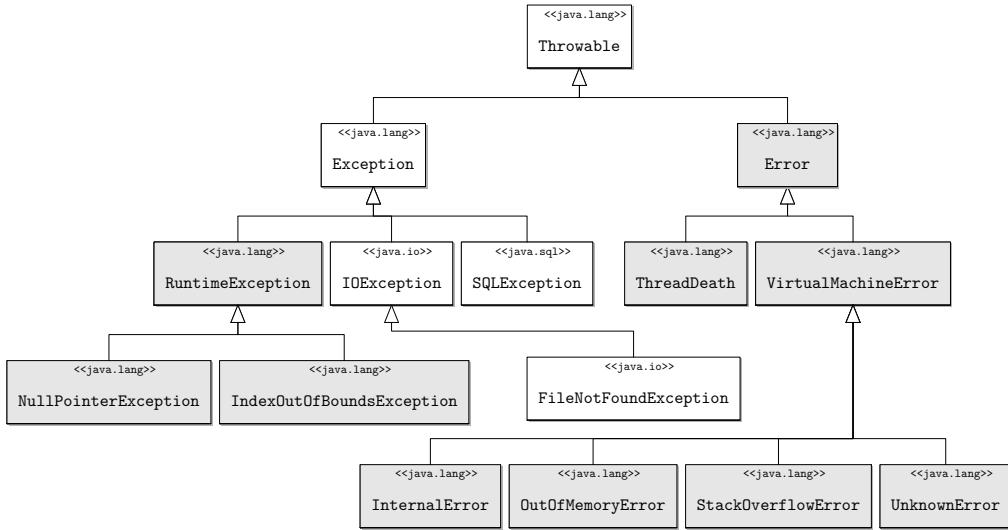


Figure 2.8: the throwable class hierarchy

## 2.8 Programming Style and Documentation

This section describes various good practices for managing source code and documentation for a software project. The most important practice you must adopt is to exploit the benefits of automation when organising and documenting software. Computers are excellent tools for managing boring repetitive tasks (they really enjoy it).

If you find yourself re-doing a task more than twice that involves more than one action, find a software tool that will do it for you. Software engineers love building applications that will help them to their job better, and they often distribute them for free<sup>2</sup>. Examples include:

Automation in source  
code management  
(62)

- preparing a software project for deployment outside of the development environment;
- running tests repetitively to ensure that old defects aren't re-introduced;
- formatting logging statements so that the log files can be automatically processed and analysed;
- making continual backups so that previous versions can be recovered;
- preparing user documentation from source code documentation; and
- SmallTalk (geddit?).

---

<sup>2</sup>This is often called Yacc (Yet Another Compiler, Compiler) shaving, or procrastination by managers.

```

bin
build.xml
config
    l4j.properties, build.properties
doc
    javadoc
    manual
    readme.txt
    uml
lib
    javax.jar, log4j.jar, servlet.jar
log
    20110104-1215.log, 20110104-1244.log
src

```

Figure 2.9: listing of a typical project directory layout

If you can't find something that does exactly what you need, you can always write a script in Python, Perl, ant, make, TCL, Jacl, or any other scripting language suitable for the task.

Software projects have multiple artefacts that need to be managed in order to preserve and improve quality. Careful and consistent layout of the artefacts in a project repository can help the software management process. Specifically:

- adopt a standard directory organisation for every project you start. The organisation you work for may have rules about this, but your development environment should also be configured to do it automatically; and
- separate user interface, test harness and functional code into different modules. For example:
  - when you create a unit test for a class, place it in a sub-package of the target class' package called 'test'
  - collect the user interface components of your application into a separate high level package
  - you should have a separate Main class entry point for invoking your program, the Main class should parse any command line arguments and pass these programmatically to the 'real' program entry point.

## Organising a software project (64)

## Example layout (65)

Remember, tidiness is next to noodily goodness.<sup>3</sup>. A directory listing showing a typical layout for a project is shown in Figure 2.9.

A consistent and organised approach should also be adopted for programming. This is often referred to informally as writing 'clean code' (not to be

---

<sup>3</sup>See [http://www.wikipedia.org/wiki/Flying\\_Spaghetti\\_Monster](http://www.wikipedia.org/wiki/Flying_Spaghetti_Monster)

confused with the Clean programming language<sup>4)</sup> because the code is free of clutter or obscurity that hinders understanding.

High quality software is often described as self documenting, because the source code expresses the functionality sufficiently well that extensive comments explaining what the code does and how it does it is unnecessary. Although it is usually not possible to write code that doesn't require *any* documentation, being able to reduce the amount of documentation means that:

- the effort required to maintain documentation is reduced (and therefore the documentation is more likely to be current); and
- the clarity of the source code is not hindered by extensive comments.

Features of self-documenting code include:

- consistent identifier selection following language and organisational conventions;
- consistent layout of program constructs, including:
  - placement and use of braces,
  - indentation,
  - for and if loops, and
  - variable declaration;
- simplest (least obscure) implementation;
- single definitions of literal values and algorithms.

Avoid all *cloning* of code if at all possible. The cut-and-paste function in your editor should be considered harmful and treated as deprecated. Instead, encapsulate the code you want to re-use in a parameterised method definition.

Another way of improving clarity of source code is to organize the class members consistently, for example, declaring class members in this order:

- static variables,
- instance variables,
- constructors,
- public (API) methods,
- private methods and finally
- static methods.

Programming practices: features of self documenting code (66)

Organising classes (67)

---

<sup>4)</sup><http://wiki.clean.cs.ru.nl/Clean>

By doing this, the methods used to instantiate and control the class' objects are placed near the top of the method (these typically form the API for the class). Utility private and static methods are placed lower down, but also grouped together.

Some software engineers adopt a rather extreme position on self-documenting code, arguing that users should rely *only* on the code as written for their understanding of a software application - after all this *is* a form of documentation as to what the program will do. The argument is that maintaining supplementary documentation explaining what the high level language source code does violates the requirement not to repeat yourself (i.e. source files containing both code and comments is a form of cloning). In addition, the supplementary documentation will inevitably become outdated as the source code evolves. This approach is flawed for two reasons.

- source code documentation explains what a program does, as well as how it works. The 'what' of source code documentation changes less frequently than the how, because interfaces between modules need to remain fairly constant, whereas the details of how a program works change as defects are corrected and performance improvements are made. Without the 'what' documentation, software engineers who are uninterested in the 'how' will need to read the source code in order to understand how the program works; and
- very few program implementations are sufficiently clear to not require further documentation to assist understanding in places. Quite apart from the fact that few developers are truly rigorous in their approach to writing code, system development often requires imperfect 'hacks' or 'cludges' because of the design decisions made for components on which the project depends. In this situation, a source code comment is useful in explaining why the implementation is less than 'ideal'.

The code below shows example documentation for an operation specified in an interface (the operation converts a binary string into its decimal equivalent). The documentation includes an overview of the functionality of the operation, the required arguments and the value to be returned. The documentation also describes the circumstances in which an exception will be raised.

Why source comments  
are necessary (68)

```
/**  
 * Converts a String of binary symbols (0,1)* in two's  
 * complement format, into the equivalent (signed)  
 * integer  
 * value.  
 *  
 * @param binString  
 *        the string of binary symbols (0,1)*  
 * @returns a decimal representation of the binary input  
 */
```

```

* @throws NumberFormatException
*         if the input string contains any
*         character
*         other than 0 or 1
*/
public int binaryToDecimal(String binString)
    throws NumberFormatException;

```

Once the specification for an operation has been documented, it is unnecessary to re-document the specification in implementing classes, unless the implementation overrides the expected behaviour described in the interface. For example, the `List.add(o:Object)` method overrides the more general behaviour of the `Collection.add(o:Object)` method. Where the behaviour is unchanged, a reference to the documentation is sufficient, as shown in the code below.

```

/**
 * @see uk.ac.glasgow.oose2.documentation.
    BinaryToDecimalUtil#binaryToDecimal(java.lang.String
)
*/
@Override
public int binaryToDecimal(String binString) throws
    NumberFormatException {
}

```

Comment code:  
Javadoc for the  
implementation (70)

Inline documentation should explain the purpose of *chunks* of code. It is usually unnecessary (and counter-productive) to provide an explanation of each line of code. The code below shows the documentation of the implementation of the `binaryToDecimal()` method.

```

//check input first.
for (char c: binString.toCharArray())
    if (c != '0' && c != '1')
        throw new NumberFormatException(
            "Only binary symbols [0,1] permitted");

//set up the accumulators
int result = 0;
int twos = 1;

//reverse the input so that numerals are dealt with in
//increasing order
StringBuffer buf =
    new StringBuffer(binString).reverse();
String revBinString = buf.toString();

//process each numeral of the prepared string
for (char c: revBinString.toCharArray()){
    if (c == '1')
        result += twos;
}

```

```

    twos *= 2;
}

if (isTwosComplement){
    //apply two's complement rule
    char twosComplement =
        revBinString.charAt(revBinString.length()-1);

    if (twosComplement == '1') result = result - twos;
}
return result;

```

## Summary

This chapter has reviewed object oriented software development concepts. In parallel the chapter reviews good practice in software development, which improves abstraction, organisation and software quality.

An object oriented approach to software development partitions data and behaviour into objects. The overall behaviour of the system is a result of the interactions between the collection of object in the system. Object oriented modelling and development is characterised by two features useful for managing system complexity:

Features of object oriented development  
(71)

- **encapsulation and information hiding** of the state and internal behaviour of an object so that is not directly accessible to external service clients. Access to internal state can be regulated by the specification of an object's operations. Operations can be provided to create a new instance (a *constructor*), alter an object's state (a *mutator*) or access the state (an *accessor*). System development is concentrated on what the services an object provides can do, not how it does it.
- **inheritance and polymorphism** encourages re-use of higher level class specifications among more detailed implementations. Polymorphism enables multiple implementations to handle the same message from a client in different ways, in a manner which is transparent to the client.

## 2.9 Exercises

1. What are the main features of objects and classes in the object oriented paradigm?
2. Explain, with an example, how you would distinguish between an object and a class in a problem description.
3. Which of the following items do you think should be a class, and which should be an instance?

(a) Falkirk	(k) TWS's laptop
(b) West Highland Way	(l) Linux Operating System
(c) Schiehallion	(m) University Course Object Oriented Software Engineering 2
(d) pint of beer	(n) The Restaurant at the End of the Universe
(e) web server	(o) Donald Knuth
(f) UML class diagram	(p) Cricket
(g) train	(q) Cricket ball
(h) 18:11 Glasgow Queen Street - Perth train service	(r) off-side rule
(i) laptop	(s) Jupiter
(j) Sony VGN-TT2 laptop	

For any item that should be an instance, name a suitable class for it. If you think an item could be either a class or an instance, depending on circumstances, explain why.

4. Which of the following would not form good sub-class → super-class pairs (generalizations), and why?

(a) snow → weather	ticket → theatre ticket
(b) country → continent	(g) golf club → team
(c) bicycle → vehicle	(h) lecture → university course
(d) Charlie the Cat → Marmalade Tom	(i) lecturer → course coordinator
(e) table → furniture	(j) football magazine → magazine
(f) London West End theatre	(k) carrot → vegetable

Look for:

- violations of the *isa* rule;

“a university has developed a system for managing courses and assessed work. A course may have several coursework and deliverables and/or an exam. A course is delivered by a lecturer and a number of tutors and is taken by students.”

Figure 2.10: an initial problem description for the university course management system

- lack of behaviour specialization; and
  - other problems.
5. Consider the description for a system for managing assessed work in a university in Figure 2.10.
- Identify the main classes in the problem description and give some example instances;
  - add attributes and operations to the classes where possible; and
  - consider what assumptions you would need to make to proceed further.
6. Improve the readability and documentation of this method (it converts a color image into greyscale):

```
public static BufferedImage RGBtoGreyScale(BufferedImage bi
)
throws Exception {

if (bi.getColorModel().getPixelSize() != 24) throw new
Exception();

BufferedImage myResult = new BufferedImage(bi.getWidth(),
bi.getHeight(), BufferedImage.TYPE_INT_RGB);
Graphics2D graphicsObj = myResult.createGraphics();

for (int x = 0; x < bi.getWidth(); x++)
for (int Z = 0; Z < bi.getHeight(); Z++) {
int PIXEL = bi.getRGB(x, Z);
Color color = new Color(PIXEL);

int average = (color.getBlue() + color.getGreen() +
color
.getRed()) / 3;
graphicsObj.setColor(new Color(average, average,
average));
graphicsObj.drawLine(x, Z, x, Z);
}
}
```

```
    return myResult;  
}
```

## 2.10 Workshop: Generic Types and Auto-Boxing in Java

In general, it is desirable to detect as many defects caused by programming errors as possible at compile time rather than runtime.

A prevalent problem when working with programming languages which support polymorphic types is the risk of a runtime cast error. This tutorial introduces the use of generics to *statically* detect and prevent type cast defects in Java.

### 2.10.1 Runtime Type-cast Defects

Consider the following code for a class that is used to hold a single object reference value:

```
public class Box {  
    private Object contained;  
  
    public void add(Object contained) {  
        this.contained = contained;  
    }  
  
    public Object get() {  
        return this.contained;  
    }  
}
```

The code could, for example, be extended to provide an implementation of a linked list. However, we are interested in the storage and retrieval of objects from the box. A limitation of the Box class is that the contained attribute must be declared to be the most abstract type that the box might be required to store (in this case the Object class parent of all other classes).

The code below shows how values can be stored in the box and subsequently retrieved:

```
public static void main(String[] args) {  
    // ONLY place Integer objects into this box!  
    Box integerBox = new Box();  
    integerBox.add(new Integer(10));  
  
    Integer someInteger = (Integer) integerBox.get();  
  
    System.out.println(someInteger);  
}
```

Note that:

- the author has helpfully documented that this particular use of box should only be used to store Integer type variables;

- the `Integer` instance is implicitly cast to the more abstract type `Object` when it is passed as an argument to the `add()` method;
- the primitive `int` type that we actually wish to store in the box has had to be wrapped in an `Integer` class instance; and
- that when the value is retrieved from the box it must be explicitly cast as an `Integer` type if it is to be assigned to an `Integer` type variable.

All these aspects of the use of `Box` are rather awkward and unnecessarily clutter in the code. In addition, there is no guarantee that all future uses of the `Box` instance assigned to `integerBox` will respect the original author's instructions. Consider for example, the following code:

```
// ONLY place Integer objects into this box!
Box integerBox = new Box();
// Imagine this is one part of a large application
// modified by one programmer.
integerBox.add("10"); // note how the type is now
    String
// ... and this is another, perhaps written
// by a different programmer
Integer someInteger = (Integer) integerBox.get();
System.out.println(someInteger);
```

The code will successfully compile, however when the program is executed, the following exception will be caused:

```
Exception in thread "main" java.lang.ClassCastException: java.
lang.String cannot be cast to java.lang.Integer
```

The inappropriate cast of a `String` to an `Integer` is only manifested at runtime. Ideally, since the appropriate type for the `integerBox` instance is known at compile time, it would be better to catch the defect introduced then.

Runtime detection of casting defects (80)

## 2.10.2 Using Generics

The generics extension introduced in Java 5 provides features for performing some static type-checking of variables at compile time. Generic classes are declared by introducing a generic *type variable* into their class definition. Consider the revised version of the `Box` class shown below.

```
public class Box<T> {
    // T stands for "Type"
    private T t;

    public void add(T t) {
        this.t = t;
    }
}
```

Creating a generic class (81)

```

public T get() {
    return this.t;
}
}

```

The type variable T is placed in angle brackets after the Box type identifier is declared. After that, the type variable can be used in the rest of the class definition as if it was a normal type. In particular, the T type can be used to define the type of the contained attribute and the associated add() and get() methods.

Whenever the generic Box<T> is used in a variable declaration, we can specify the particular type that should be used in place of T. The code below gives an example of how this is done.

### Using the Box Class (82)

```

Box<Integer> integerBox = new Box<Integer>();
integerBox.add(new Integer(10));

Integer someInteger = integerBox.get(); // no cast!
System.out.println(someInteger);

// Can only add a String.
Box<String> strBox = new Box<String>();
strBox.add("ah, a box that fits just right!");
System.out.println(strBox.get());

```

Notice that:

- the Integer and String types are used to replace the T type variable with a concrete type;
- the result of the get() method no longer needs to be explicitly cast to an Integer or String since the type of objects to be held in the box is specified at compile time; and
- the int value is not wrapped in an explicit Integer object before being passed to the add() method. This is a related feature also introduced in Java 5 called auto-boxing. Primitive types are automatically wrapped in their equivalent reference types when mapped to a reference type parameter.

The use of generics means that attempts to perform inappropriate casts at runtime are caught during compilation - go ahead, try and add a String instance to a Box<Integer> instance.

#### 2.10.3 Scoping Generic Types

Sometimes it can be useful to restrict the scope of generic types, so that they can only be instantiated with a particular set of Java classes. The code below

shows how this can be done for a class which can be used to provide persistent behaviour for objects.

```
public class PersistentObject<T extends Serializable> {

    private final Class<? extends T> _clazz;
    private T inner;

    private OutputStream os;
    private InputStream is;

    public PersistentObject(Class<? extends T> _clazz,
                           InputStream is, OutputStream os) {
        this._clazz = _clazz;
        this.is = is;
        this.os = os;

        retrieve();
    }

    public T get() {
        return inner;
    }
}
```

The Java code shows a class for persisting Java objects in a `OutputStream`/`InputStream` pair. When a new instance of the `PersistentObject` class is created, it is passed the I/O stream pair and a class object which defines the type of object that this `PersistentObject` is to persist. The constructor assigns these parameters to the appropriate attributes and then invokes the `retrieve()` method, whichs recovers any stored state information for the `inner` instance.

The code below shows how the `retrieve()` method works.

```
@SuppressWarnings("unchecked")
private void retrieve() {
    try {
        ObjectInputStream ois = null;
        try {
            ois = new ObjectInputStream(is);
        } catch (IOException e) {
            System.err.println("Couldn't open input stream [" +
                               + is + "]");
        }

        inner = (T) ois.readObject();

    } catch (ClassNotFoundException e) {
        System.err
            .println("Couldn't instantiate inner object of type
                    ["

```

```

        + _clazz + "]");
        e.printStackTrace();
    } catch (ClassCastException e) {
        System.err.println("Wrong type for inner object: ["
            + _clazz + "]");
    }

    } catch (IOException e) {
        System.err.println("Couldn't read object of type ["
            + _clazz + "]");
    }

    if (inner == null)
        try {
            inner = _clazz.newInstance();
        } catch (InstantiationException e) {
            System.err
                .println("Couldn't instantiate new instance of ["
                    + _clazz.getName() + "].");
        } catch (IllegalAccessException e) {
            System.err
                .println("Couldn't access nullary constructor of ["
                    + _clazz.getName()
                    + "] to instantiate new inner object.");
        }
    }
}

```

The nested try-catch block opens the provided input stream and wraps it in a `ObjectInputStream`. The nested try-catch is necessary to distinguish the two potential causes of an `IOException` - creating the stream and reading an object. The next statement attempts to read an object of the generic type from the stream. Notice that the method has to be annotated with a `@SuppressWarnings` annotation because the cast to generic type `T` is unchecked. Finally, The condition checks if the object was successfully read from the stream. If the object was not read, the third try-catch block attempts to initialise a new object of the specified type. The `update()` works in a similar way, but with an `ObjectOutputStream` to write the object to store.

So how does all this relate to scoped generics? Look at the class signature and the definition of the `_clazz` attribute:

```

public class PersistentObject<T extends Serializable> {

    private final Class<? extends T> _clazz;

```

which are both examples of scoped generic types. The class signature specifies that the `PersistentObject` class can be used with any class which implements the `Serializable` interface. This is crucial for the functionality, since only `Serializable` instances can be read and written from object streams.

The `_clazz` attribute is used to initialize the `inner` instance attribute when an object cannot be read from store. The type declaration says that any

class which is a sub-class of T can be used for this purpose. This allows the PersistentObject class to be declared to be of type Map say, but use the HashMap class to provide concrete instances of the interface.

#### 2.10.4 Behind the Scenes

So how does all this work? When the code is compiled, the compiler performs the necessary compile time checks to make sure that the *erased* types do not violate the type checking rules. The concrete type specified in a generic variable declaration is *erased* and a cast annotation for that type is added to any statements where a specific variable type is assigned a generic instance.

So, the checks are made *statically* during compilation, but the generic types themselves don't appear in the compiled code. The generic types and type-safety guarantees are not present in the compiled code. This means that it is possible to 'fool' the generics mechanism in Java and generate a ClassCastException, even though no explicit casting takes place. The code below shows how this can be done.

```
package uk.ac.glasgow.oose2.generics.revised;

public class BoxDemo4 {
    public static void main(String[] args) {
        Box<Integer> testBox = new Box<Integer>();
        testBox.add(new Integer(10));
        //testBox.add("");

        Box otherBox = testBox;

        otherBox.add("A thing");

        String x = (String)otherBox.get();
        System.out.println(x);
    }
}
```

The output from this program is as shown below.

```
Exception in thread "main" java.lang.ClassCastException: java.lang.String cannot be cast to java.lang.Integer
at uk.ac.glasgow.oose2.generics.revised.BoxDemo4.main
(BoxDemo4.java:11)
```

The declaration of the raw typed otherBox allows it to be assigned a Box of any concrete type, such as in this case, Box<Integer>. This means that a String type can be added to a supposed Box<Integer> via a raw reference type.

Behind the Scenes  
(86)

Breaking the generics  
type-safety  
mechanism (87)



## Chapter 3

# The Software Lifecycle

### Recommended Reading

**PLACE HOLDER FOR sommerville10software**

Recommended reading  
(88)

**PLACE HOLDER FOR naur68report**

**PLACE HOLDER FOR glass97b-software**

The RISKS digest <http://catless.ncl.ac.uk/Risks>

Joel on Software. <http://www.joelonsoftware.com/>

Daily Worse Than Failure. <http://www.dailywtf.com>

**PLACE HOLDER FOR rosenberg07dreaming**

Naur and Randell [1968] is the report on the first conference on software engineering sponsored by NATO in 1968. Many of the problems addressed by the conference still be-devil software projects today.

Glass [1997b] is a compendium of software project failures dating from the 1970s, 80s and 90s. Much of the technological discussion is now (unsurprisingly) rather dated. However, the problems faced by the project teams in terms of project scale and complexity, management of political conflict and uncertain requirements are as applicable today as they were in their own time.

Rosenberg [2007] is a personal account of the Open Source Application Foundation's Chandler<sup>1</sup> personal organiser project. The book describes the challenges faced by the project team, including uncertain requirements, changing software base and external political pressures.

'Joel on Software' is a very practical blog on software development and software project management issues, written by Joel Spolsky. The 'Daily Worse

---

<sup>1</sup><http://www.chandlerprxobject.org>

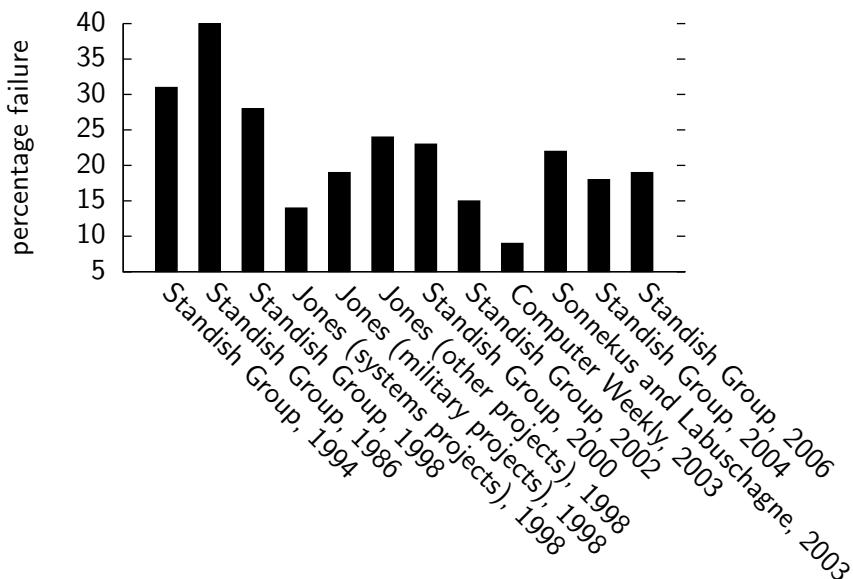


Figure 3.1: surveyed rates of software project failures, selected studies, adapted from Emam and Koru [2008]

'Than Failure (DailyWTF)' is an informal blog of 'curious perversions in information technology' (amusing war stories of software engineering in the small, medium and large cock-ups) maintained by Alex Papadimoulis. The RISKS digest is a mailing list for reporting and discussing software failures (some of the supposed failures are a little hysterical).

### 3.1 Software Engineering in the Large

The concept of software engineering was introduced in Chapter 1. We examined the reasons why software development *in the small* is hard and how the principles, practices, methods and tools of software engineering can contribute to the production of high quality software. However, most software projects are a collaborative effort amongst a team of software engineers, working with a heterogeneous group of customers, managers and users, all of which will have their own preferences for the functionality of the software system.

The scale of many software projects and the challenges of coordinating work in an already complex discipline makes such software projects even more prone to failure. Figure 3.1 illustrates a selection of estimates of software project failure rates, gathered by Emam and Koru [2008] between 1994 and 2006. Note that despite some decline in reported failures, several recent estimates put the rate of project failure at above 20%. Emam and Koru's own prediction was that software failures were between ten and fifteen percent between 2005 and 2007.

Consider the following case study, which illustrate the challenges of large-scale system development.

**The London Ambulance Service's Computer Aided System** (CAD), was conceived as a replacement for a largely paper based system, but which would also substantially alter the working practices of the service's staff. The failures associated with the project are well documented in an official inquiry [South West Thames Regional Health Authority, 1993], as well as by several other authors [Robinson, 1996, Dalcher, 1999, Finkelstein and Dowell, 1996]. The following description is largely based on these sources. The context for the project in 1991 was:

LAS CAD – overview (90)

- an emergency service with ~500 front line emergency staff operating ~200 emergency vehicles ~ 40 control room staff;
- between 2000 and 2500 calls to the emergency service every day, resulting in 1400 emergency journeys (the LAS also provides non-emergency transport for patients);
- a largely manual system for managing and allocating service resources to emergencies, that was widely perceived as under strain;
- a previous attempt at computerisation that had been abandoned due to project budget and schedule over-runs;
- an atmosphere of *mistrust* between ambulance crews, dispatch centre staff and management, following an acrimonious pay dispute and a major re-organization; and
- a political and management desire to meet standards for response times without major additional investment. Indeed, the LAS budget was reduced during the project.

The project to deliver the system, supplied by a consortium of Apricot Systems, Mitsubishi, Datatrak and System Options, was divided into three phases. The first two phases were concerned with improving the call taking and resource allocation activities. During the final phase, the allocation system would be fully automated, making the role of resource allocation redundant.

The project had experienced on-going problems during phase one and two. However, the new system suffered serious overload following a deployment on the 26<sup>th</sup> October, 1992, during the final phase three launch. Evidence of severe problems with the system throughout the project include:

LAS CAD – anecdotes of the failures (91)

- management ignored early warnings of problems, such as from experts on safety critical systems;

- calls to the service took 11 hours to receive a response;
- new calls to the system sometimes wiped old calls before they had been dealt with;
- vehicle location information was inaccurate and sometimes the nearest vehicle to an emergency could not be identified;
- the vehicle location system failed to identify every 53<sup>rd</sup> vehicle;
- sometimes, two ambulances would be sent to the same emergency;
- several sources allege that approximately 30 people died as a consequence of the new system (these figures are disputed and difficult to verify);
- the service received 900 complaints in the days following initial deployment;
- the chief executive of the LAS resigned on the 28<sup>th</sup> of October (two days after the phase three go live). The chairman of the LAS resigned five months later;
- Systems Options lost their contracts with other customers;
- the new system was abandoned at a cost of £43 million (an initial cost estimate was £1.5 million); and
- the introduction of successful computerisation in the LAS was delayed by a further four years, contributing to a number of unnecessary deaths under the manual system.

A complete system failure on the 4<sup>th</sup> of November, resulting in a reversion to the full manual system. The *proximate* cause of the *system* failure was a *memory leak*. A memory leak occurs when memory that has been allocated to temporarily to a process is not subsequently released, meaning that the amount of free memory available for use gradually declines until the system is un-useable. However, the causes of the *project* failure were much more widespread than this. Typically, we can classify the causes of software project failure as:

1. building a system for the wrong reason;
2. building the wrong system; and
3. building the system wrong.

The official inquiry into the LAS CAD project identified numerous causes of the system failure. The documented causes of failure can be categorised using the scheme described above.

The system was built for the wrong reasons, because there was:

Causes of software project failure (92)

The LAS disaster – causes (93)

- a political imperative to make budget savings across the service, creating a preference for the lowest tender;
- pressure to complete the system within a (eventually extended) timeframe of less than a year: the original specification, completed in February 1991 required complete system implementation by January 1992; and
- a desire to automate the decision of which ambulances to allocate to which emergencies, which had previously been taken by emergency medical dispatchers.

The wrong system was built because there was:

- little consultation with prospective system users (emergency or control room) about the nature of their work;
- a ‘gross’ under-estimation of the costs of the software development part of the project;
- a dependence on ‘perfect’ information and communication channels, causing a build of error reports;

The system was built wrongly because:

- there was inadequate training of staff for the new system, creating an environment of hostility and suspicion;
- parts of the system was developed using Visual Basic, a (then) un-proven programming language;
- code changes were made ‘on-the-fly’ in response to user requests and in many cases undocumented;
- test coverage (particularly of throughput) was limited due to the constrained nature of the project plan;
- deployment was disorganized and haphazard; and
- proscribed standards were allowed to slip (again, due to the constrained project time frame).

The causes of project failure should usually be addressed in the order listed: if you are building a system for the wrong reason, there is little point trying to ensure the system meets those reasons (i.e. trying to get the requirements right); if you are building the wrong system, then trying to ensure the system meets (incorrect) requirements is not much use.

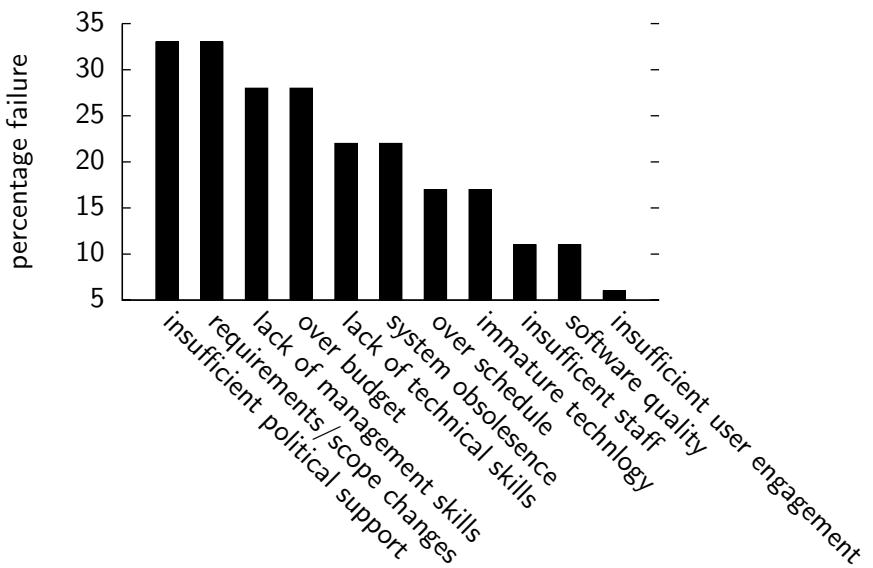


Figure 3.2: (non exclusive) causes of software project failure [Emam and Koru, 2008]

More generally, Emam and Koru [2008] have surveyed the causes of project cancellation, as shown in Figure 3.2. The causes are not exclusive, so a project cancellation could be attributed to more than one cause.

Of particular interest is that the two biggest causes of project cancellation can be categorised as building the system for the wrong reason (insufficient management support) and building the wrong system (requirements/scope changes).

An easy question to ask, but often difficult to answer is: who is to blame for a project failure? In the case of LAS CAD, the consortium that provided the system was overly optimistic about their ability to deliver. Consortiums are almost always difficult organisational structures to manage, such that they have clear purpose and direction.<sup>2</sup>

However, the LAS managers who specified the project imposed impossible constraints on the delivery of the system, which made delivery within the expected timescale and budget allowed highly unlikely. The design of the tendering process meant that the cheapest bidder (regardless of their capacity to deliver) would be awarded the contract. Consequently it was seems unlikely that the LAS managers ever questioned why the lowest bid was dramatically cheaper than any other.

A political scientist might, of course, note the pressures that the LAS management were under in the immediate short term to simultaneously improve performance and cut budgets was highly detrimental to the chances of project success. Had the project proceeded in a more relaxed atmosphere, then per-

Causes of software project cancellation (94)

---

<sup>2</sup>Ask any academic about collaborative research projects.

haps more realistic decisions might have been made. Consequently, it could be argued, the project was doomed to failure by high level decisions far removed from the management of the project of itself.

And yet, this would seem to allow any of the software and system development professionals involved to abdicate any responsibility for the project's consequences by 'referring complaints to higher powers.' Consider the British Computer Society's code of conduct<sup>3</sup>:

You shall:

1. (a) have regard for the public health, safety and environment...
2. (a) only offer to do work or provide a service that is within your professional competence.  
(b) not claim any level of competence that you do not possess...

The British Computer Society code of conduct (95)

Clearly, the code of conduct implies that IT professionals hold responsibility for the consequences of projects they work on. Bear in mind that although you are not legally bound by the code of conduct, you may face disciplinary action as a member of the BCS and you might be legally and or criminally liable for your actions if you dis-regard it.

Software engineering as a discipline in its own right was proposed because of the perceived failure of software development projects during the 1950s and 1960s. The challenges of producing software of sufficient quality to meet the project objectives, within the anticipated schedule and budget were considered so severe that commentators referred to the problem as the *software crisis*.

Numerous definitions of the discipline have been proposed, for example:

the principles, methods, techniques and tools for the specification, development, management and evolution of software systems

Defining software engineering (96)

[Sommerville, 2010]

The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software

[Bourque and Dupuis, 2005]

the process of solving customer's problems by the systematic development of large, high-quality software systems within cost, time and other constraints

---

<sup>3</sup><http://www.bcs.org/upload/pdf/conduct.pdf>



Figure 3.3: the e-counting system used for the Scottish General Election in 2007

[Lethbridge and Laganière, 2005]

In summary, software engineering is concerned with the production of software of sufficient quality which meets the *real* needs of the customer. This is achieved through the methodical application of computing science (and other scientific disciplines) to the practice of software development.

Software engineering has made considerable progress over the previous half-century in reducing the problems associated with developing large scale software systems. Emam and Koru [2008], for example, as we have already seen, found a general decline in software project failures since 1994. Despite this, it's still fairly easy to draw on examples of software project failures in the 21<sup>st</sup> Century. Consider the following two recent case studies.

**Scottish Elections e-Count System, May 2007** Figure 3.3 illustrates the e-counting system used by the Scottish Executive for the Scottish General Election in 2007 Lock et al. [2008]. The system was used to electronically count the paper ballots marked by voters. Despite successful acceptance test (demonstrating that the system met the customer's requirements), the system suffered extensive problems on the night of the election count, including:

- loss of voting privacy for voters due to the requirement that ballot papers be placed into a ballot box unfolded;
- slower ballot processing than expected due to repeated paper jams in the scanners;
- a higher rate of rejected or unscanned ballots (approximately 140,000 ballot papers were rejected across Scotland; and
- the inability of monitoring systems to display count progress to election monitors. This was symptomatic of a problem later in the night, when

Scottish Elections,  
2007 (97)

election officials were unable to obtain the raw vote data from the system database in order to produce a final result.

The problems experienced, caused a loss of *trust* in the results that were finally produced, because the system had not satisfied the implicit requirement to be transparent, as well as accurate and efficient. The official inquiry into the failure Gould [2007] of the system concluded (among other problems) that:

- management of the project had been confused and split between the Scottish Executive, the local authority Returning Officers (chief election officials), the Scottish Office and the vendor;
- the timetable for the project had been compressed due to uncertainty over legislation. Legislation permitting the use of e-counting was not passed until very close to the election; and
- testing of the system was limited and not realistic.

In summary, the *scale* of the project was beyond the capabilities of the consortium formed to manage it. The resources available (both budgetary and schedule) were insufficient to deliver a system of sufficient quality, resulting in severe problems on election night.

**The New York Stock Exchange ‘Flash Crash’, 6<sup>th</sup> May, 2010** occurred during a short period of time in the late afternoon and is described in an official report released later in the year [U.S. CFTC/SEC, 2010]. During a matter of minutes, the quoted value of securities on the exchange began to fluctuate dramatically, with some trades occurring at extreme dollar values (as low as a penny, or as high as \$100,000). Several indices dropped by nearly 6% in a matter of minutes, before re-bounding. In an extremely short time \$800 billion dollars of value disappeared from the market before re-appearing. This volatility is illustrated by the dramatic decline and then recovery of the Accenture share price, as shown in Figure 3.4.

The official report into the events, suggested that the proximate cause of the crash to a large trader who initiated an automated sell program for approximately \$ 4.1 billion of contracts [U.S. CFTC/SEC, 2010]. The program was configured to sell the contracts when the volume of activity in the market was high, one indicator that there are ready buyers for the contracts. Unfortunately, the large number of sells on offer caused sharp declines in the number of traders willing to buy. Consequently, this caused a large number of sells to be initiated as price values declined. This triggered more sells by the automated program as volume increased, because the program did not monitor prices as well as volume. Only a short pause in trading by several of the larger traders eventually allowed the exchange to recover; however, the situation could have been much worse. This

New York Stock  
Exchange ‘Flash  
Crash’, May 2010 (98)

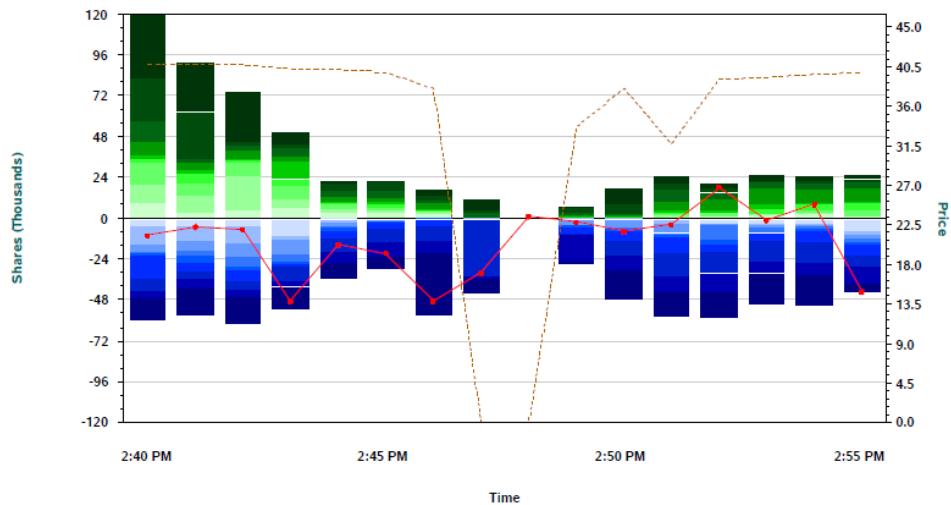


Figure 3.4: decline and recovery in Accenture share price value 2.40pm - 2.55pm, 6<sup>th</sup> May, 2010. Extracted from U.S. CFTC/SEC [2010].

explanation has been contested by several other sources commenting on the events.

All three case studies presented in this chapter can be characterised as large, complex systems. Such systems (in any discipline) are prone to 'failures', characterised by budget or deadline over-runs, negative political or social impact, reduction in functionality, project cancellation or outright system collapse. Software projects are often compared to large, complex and unique projects in civil engineering, which are similarly prone to failure (think of the Tay bridge disaster, for example).

However, such failures are often followed by successful development projects once the lessons from the initial attempts have been learnt. In the LAS case, a CAD system was eventually successfully implemented by the end of the decade; and substantial revisions have been made to the e-counting system used for Scottish Elections in preparation for 2012.

Comparing the flash crash software failure with the e-counting and LAS CAD systems reveals quite different characteristics. This scale of system, (sometimes dubbed *systems of systems*) where behaviour is manifested by the interactions between numerous independently managed software programs is an increasing facet of software engineering. Their domains of control extend across numerous organisations and legal jurisdictions and are near on impossible for any one developer, or even a development team to fully understand or manage. Such systems become extremely long lived, and consequently are never developed or de-commissioned in their entirety. Rather, the overall system evolves gradually as sub-systems and components are added and replaced.

Failures in such systems are likely to be quite different from those in tradi-

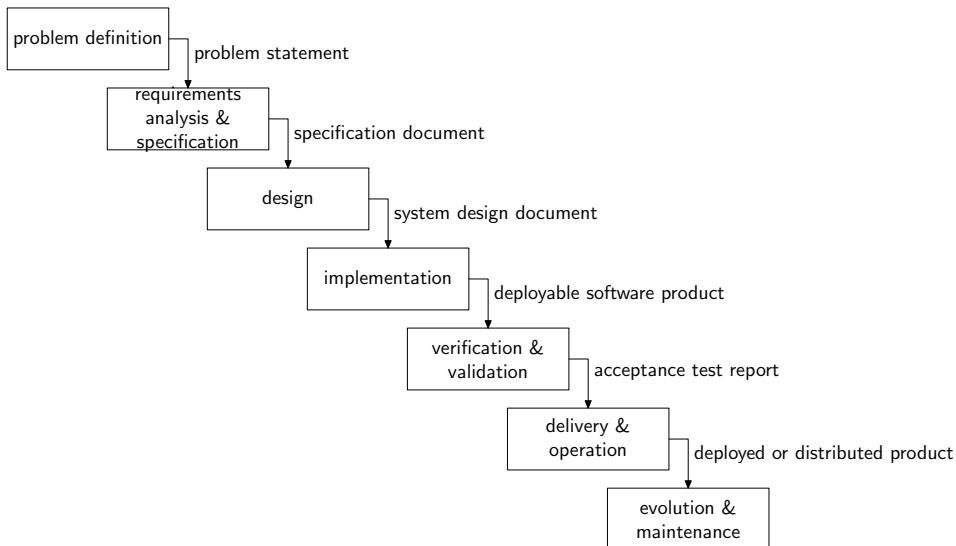


Figure 3.5: the waterfall software development process model

tional development projects, as the ‘Flash Crash’ illustrates. In addition, software development processes will need to reflect the inter-connection of systems-of-systems. Much greater emphasis will need to be placed on software development as a maintenance and evolution activity, rather than as a development model.

The purpose of these notes is to introduce you to some of the *risks* of large scale, collaborative software development, and some of the methods, techniques and tools that have been developed to address them. During the course, you will have the opportunity to practice software development on a software team project. The techniques introduced do not necessarily address the challenges of ultra-large scale, or system-of-systems development. The ‘Flash Crash’ case study was introduced to illustrate the limits of our current understanding of the capability of software engineering.

## 3.2 Software Development Activities

Software is often said to have a *life cycle*, in which a number of basic activities recur. Figure 3.5 illustrates the basic phases common to many software engineering process models (in different guises and with different emphasis). The figure also shows the key outputs from each of the phases:

Software development activities (100)

- Establishing a *problem definition* initiates a period of software development by providing a high level outline of the problem to be addressed.  
**output:** an initial *problem statement*

- During the *requirements analysis and specification* phase the initial problem statement is refined and extended. The *scope* of the project is established. Two questions in particular must be answered during this phase:
  1. “*what is the problem to be solved?*” concerns what is inadequate about the world that the development project should put right. In addition, it is necessary to establish the conditions under which the project can be considered successful, and the metrics used to gauge this. Note that the outcome from this activity may be a decision to not build a software system at all.
  2. “*what is needed to solve the problem?*” identifies what should be developed to solve the identified problem, such as the *functional* capabilities of a system and its non-functional characteristics.

**outputs:** a *requirements specification document* detailing *what* the system to be built is to do and a *test plan* containing acceptance test cases for the proposed system.

- The *design* of a system describes *how* the ‘*what*’ the system should do from the requirements specification is to be achieved.  
**output:** a system design document containing an architectural description of the system components and their relationships; and a description of how the components interact with each other (dynamic behaviour).
- During *implementation* the proposed software system is written, typically in a high level programming language. The product must also be prepared for testing and delivery at this time. In addition, any accompanying documentation such as user manuals must also be produced.  
**output:** a deployable *software product*
- The *verification and validation* phase is concerned with ensuring that the developed system satisfies the customer’s requirements. Verification is concerned with whether the software product is being built correctly, typically whether it satisfies its documented requirements. Validation considers the fundamental question of whether the software satisfies the customers real needs, irrespective or documented requirements.  
**output:** an *acceptance test report*, describing the extent to which the software product meets the customer’s requirements
- During *delivery* the software product is handed over to the customer. Depending on the type of project, this may be a *bespoke* deployment, or the preparation of the product for distribution to a large customer base.  
**output:** a deployed or distributed product

- The *operation and evolution* of the deployed system represents the period of time when the customer's users interact with the system. As a result, the customer may identify defects in the new system or potential new features. As a consequence, the system must be further developed to meet the customer's needs.

**output:** further versions of the system

The activities in the model are organised into a *waterfall* Benington [1983]; the outputs from higher activities flow downwards into the next activity until the system is deployed. We will begin this course by examining the early stage activities, concerning requirements elicitation.

### 3.3 Software Development Process Models

Software development process models are frameworks for organising the phases and activities that occur during a software project life cycle. Different process models have been proposed that have advantages and disadvantages for different types of project. Process models prescribe when and how activities should occur, and often incorporates particular software practices and standards. There are a number of different types of process model:

- linear (waterfall, which we have reviewed above);
- iterative (multiple releases);
- concurrent (multiple activities occurring at the same time); and
- configurable or adaptive.

Software development process models (101)

All process models are approximations of the reality in a real project, as accommodations are made for particular needs or events. Chapter 4 describes a number of different software process models, including the Rational Unified Process, and the XP agile methodology.

### 3.4 Reusable Software

Much of software development now depend on previously developed components to provide substantial parts of their functionality. Examples of software components include:

- platforms such as operating systems for desktop computers and mobile phones;
- middleware for standardising communication between distributed components, such as ERP or service oriented architectures;

Software components (102)

## Component based engineering (103)

- frameworks for providing, for example, standardised logging facilities and automated test harnesses; and
- libraries for implementing, for example, communication protocols, data format parsers, and encryption algorithms.

The practice of re-using software components has led to the development of object and component based software engineering.

The features of component based software engineering include:

- availability of utility components for re-use encouraging greater maturity of standardized software;
- emphasis on assembly and configuration rather than component design changing the role a system developer to one of composition and orchestration rather than implementation;
- service provision across organisational boundaries extending the notion of commercial off the shelf (COTS) applications to the software development process, improving competition between software providers; and
- runtime composition where the choice of component to be utilised in a software process is made dynamically, just before the component is required. Typically, components are selected from a registry of services available at the time of need. As improved components are made available, these need only to be added to the registry.

Re-using previously developed components can benefit a software development in several ways:

- there is a cost saving in terms of both time and money, since the re-used components do not have to be specified, designed and implemented and tested for each new project;
- re-using software expands the size of the user population, increasing the likelihood that defects will be discovered and corrected. This means that the overall software base for a system is more mature and stable;
- re-using high quality components improves the quality of the new system; and
- re-using software encourages developers to design their own software for re-use, creating a virtuous cycle.

Unfortunately, the development of re-usable components also constrains future development efforts, often in unforeseen ways. This has led to much of modern software development being characterised as *brownfield*, a metaphor

type	<i>analysis</i>	<i>design</i>	<i>structure</i>	<i>behaviour</i>
use case	✓			✓
activity	✓			✓
class	✓	✓	✓	
communication	✓		✓	✓
sequence	✓	✓		✓
state chart	✓			✓
package		✓		
component	✓	✓		
deployment	✓	✓		

Table 3.1: a subset of UML diagram types

from civil engineering and architecture, in which construction takes place on top of the remains of previous buildings Hopkins and Jenkins [2008]. Brownfield construction means that an architect has to conduct a site survey, looking for pre-existing foundations, pipelines, cables and so on.

For software engineering, a brownfield site survey means looking for the pre-existing IT infrastructure and dependencies that will constrain the development options for a new project. The survey allows engineers to discard some infeasible options for solving the development problem before too many resources are committed.

Brownfield  
development (104)

### 3.5 Object Oriented Software Development

Object oriented *modelling* developed in the 1980s as three separate methods: Object-Oriented Design Booch [1982], Object Modelling Technique Beck and Cunningham [1989] and the Objectory Method Kruchten [1997]. The Unified Modelling Language (UML) is a graphical notation for expressing aspects of a software development project. The language resulted from their combination during the 1990's. Table 3.1 lists a sub-set of UML diagram types that will be used during the course. The table categorizes the diagram types in two ways:

- a diagram type can be used to show the *structure* of a software artifact, illustrating the components and relationships between them; or the dynamic behaviour a system during execution.
- a diagram type is suitable for either analysis (understanding the world as it is) or construction (proposing a new software artifact), or both.

The Unified Modelling  
Language (UML)  
(105)

## Example problem definition (106)

There are numerous implementations of the UML in Computer Aided Software Engineering (CASE) tools, such as Borland Eclipse<sup>4</sup>, Umbrello<sup>5</sup>, MetaUML<sup>6</sup> and Microsoft Visio<sup>7</sup>, to name just a few. Many of these tools integrate directly with object oriented *programming languages*, such as Java and C++.

The UML notation is managed as an Object Management Group (OMG)<sup>8</sup> open standard. This has the advantage of establishing a common understanding of concepts from different perspectives of a project (requirements analyst, customer, software architect, developer and so on). Despite this, the UML notation is *not* a software development process in itself, but can be used to document the outputs from the activities of a process.

The *Rational Unified Process (RUP)* is a software development process model developed by the Rational Corporation Rational, that emphasises an object oriented approach to software development. We will discuss the model in more detail later. For now, we can begin to explore the first phase of the RUP process: project *inception*. RUP projects are initiated by the development of a *problem definition*.

Figure 3.6 illustrates an example problem definition input to the inception phase of a RUP project. The project concerns the development of a system for managing library books within a federated branch library system. The description is relatively short, and could lead to considerable uncertainty concerning the scope and duration of the project. The purpose of the inception phase is to resolve some of these issues in order to establish a sound business case. The requirements for the project will be established and refined in later phases.

*Use cases* are descriptions of interactions *between a user and the system* in order to achieve some purpose. I stress the middle part of the last sentence, because there is a tendency for those new to use cases to attempt to use them to describe activities *around* the system, which is not their purpose.

---

<sup>4</sup><http://www.eclipse.org>

<sup>5</sup><http://uml.sourceforge.net>

<sup>6</sup><http://metauml.sourceforge.net>

<sup>7</sup><http://office.microsoft.com/en-us/visio>

<sup>8</sup><http://www.omg.org>

### Managing a branch library

A university is considering the introduction of a branch library system in its departments to support staff and students in their work. A computer based system is needed for managing the movement of books from the central library and within the branch. Each branch will receive a collection of books from the main library, typically as a result of a request from a member of staff. These will then be available to loan out to members of staff and students.

Figure 3.6: example problem definition

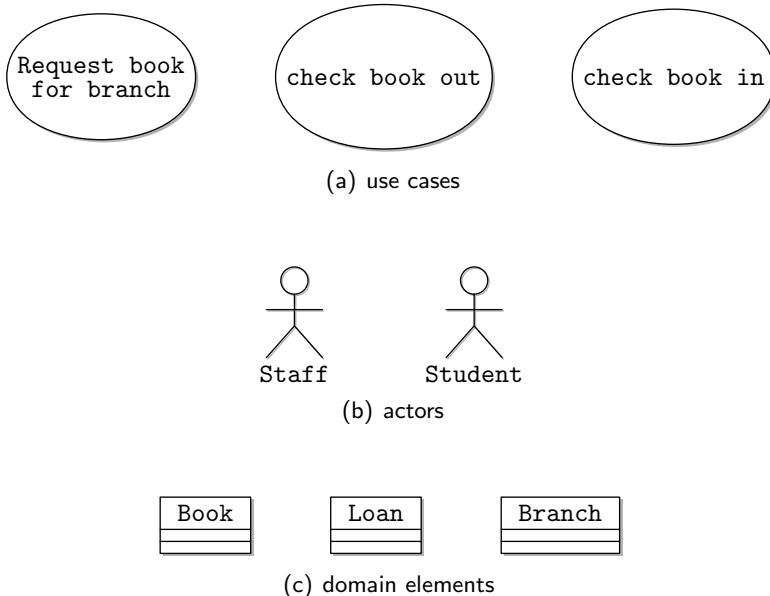


Figure 3.7: initial requirements modelling for the branch library

Figure 3.7(a) illustrates some initial use cases that can be gathered from the problem definition in Figure 3.6. The diagram shows that the system will be used to request books to be transferred to a branch, to record that books have been checked out by a user and that books have been returned.

A system will typically have a large number of small use cases. It may prove useful to refine complex use cases into several smaller related ones. The relationships between use cases and system users (organised into *roles*) define the system's *scope*. Figure 3.7(b) identifies some initial actors for the system: the staff and students who will use the branch library system.

A domain model is a high level representation of the system (data and operations) to be built and its context (environment). The model identifies the key elements which are of interest in the domain and represent them as classes. Creating a domain model is the initial step in providing functionality to realise the use cases. Figure 3.7(c) illustrates some initial classes in the domain identified from the problem definition.

All software development projects are subject to risks, potential future events which may adversely affect the success of the project. Risk planning is concerned with identifying, mitigating and managing perceived project risks. Perhaps the most common form of project failure is to build the wrong system (requirements risk). This may be for several reasons:

- a lack of understanding of the business or social context of the system by the engineers;
- uncertainty on behalf of the customer as to what the system should do

Initial modelling - use cases, actors and the domain (107)

Requirements risks (108)

- cultural differences between engineers and customers;
- ambiguous or vague requirements specifications, for example “the system needs to have features to support members of staff in processing applications”;
- ambiguities over domain items. A member of staff in a university, for example, may mean academic staff, ancillary staff, post-graduates (who often support teaching, or any combination of the above.

From the engineer’s perspective, the risk is that we do not know what the customer really wants. The solution that we will use on this course involves:

- methods for eliciting the *real* requirements from the customer;
- a semi-formal notation for describing what the system should do (use case diagrams) in a way that is understandable to both system engineers, customers and users;
- a process for tracking, managing, updating and validating use cases and other requirements ; and
- a means of investigating poorly understood requirements (prototyping) with the customer

We will begin to look at gathering and documenting requirements using use cases in Chapter 11.

## Summary

Software engineering is concerned with methods, tools and practices for reducing uncertainty (particularly requirements risk) in the software development process. Software process models are used to structure the development process, so that developers know what to do next. The UML and RUP are a object oriented modelling language and development model, respectively.

## Chapter 4

# Software Process Models

### Recommended Reading

**PLACE HOLDER FOR sommerville10software**

**PLACE HOLDER FOR jacobson99unified**

**PLACE HOLDER FOR beck05xp**

<http://www.extremeprogramming.org>

### 4.1 Introduction

Recall the waterfall model of software development introduced in Chapter 3 and illustrated in Figure 3.5. The model describes a collection of activities undertaken successively by a software development team that collectively result in a software product. The key activities during software development are the specification of requirements, development of the product, validation of the product against the requirements and then subsequent evolution. The features of a software development process can be discerned from the associated artefacts that are produced along the way. These include:

- the problem definition and requirements specification document;
- design documentation, including architectural models, behaviour descriptions and prototyping reports;
- test plans and schedules;
- software components and the complete system;
- acceptance test reports; and
- user and maintenance documentation.

Waterfall: risk management (111)

Adopting a software process model helps a project manager to control development efforts by their team and therefore manage the risks affecting the project. The waterfall model addresses risk through:

- monitoring progress to project milestones;
- planning the entire project schedule before resources are committed;
- top-down specification, design and implementation; and
- compartmentalised work efforts.

What's wrong with the waterfall model? (112)

However, there are limitations to approaching software development in this way:

- requirements elicitation must be idealised;
- high risk problems are deferred until late in the life cycle;
- later identification of mistakes in earlier activities causes *feedback*; and
- the effort and resources required for evolution are under-emphasised.

We have already seen how it is often easier to make progress during requirements gathering if clients are presented with an early possible solution in the form of a prototype. Strictly following the waterfall model makes this difficult, because design and development of solutions should not begin until the requirements document is complete and signed off.

Adopting the waterfall model also means that risks associated with the later stages of the life-cycle cannot be addressed at all until earlier activities are complete. A considerable amount of effort may be wasted on establishing a mature requirements specification that cannot be implemented, for example. Acceptance testing late in a project may uncover mismatches between the requirements specification document and what the client really wanted.

Feedback occurs because of attempts to address the two problems described above. Changes must be made to the project documentation as mistakes are uncovered during design, development and acceptance testing. In principle, further work will need to be suspended until the changes have worked their way through the process to the latest activities and the revised milestones 'signed off'. Figure 4.1 illustrates the waterfall model with this effect. Each activity now has an upward flowing arrow, illustrating the potential for later development to require changes to earlier artefacts.

For example, should acceptance testing reveal a missing 'must have' feature for the system, this will need to be added to the requirements document. The design and implementation will then need to be updated to contain the feature and the acceptance test re-started. In practice this either means that the project

Feedback in the waterfall process (113)

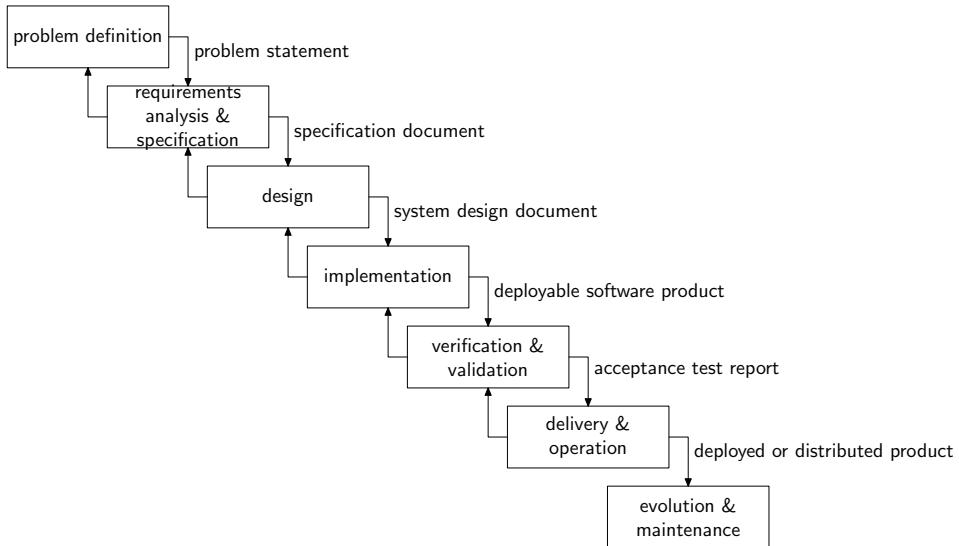


Figure 4.1: the waterfall SDP with feedback

ends up ‘looping’ endlessly without making real progress, or the documentation for the project becomes increasingly out of date.

Many software projects do not begin from an empty code base, but occur as the *evolution* of a part of a larger system, either to fix defects or to add new features. Varying estimates of maintenance effort range from 65% to 75% of a software project’s resources. Figure 4.2 shows that all of the previously described activities for a software engineering project (problem definition, requirements specification and so on) may occur during the software evolution phase.

Software evolution  
(114)

Projects of this type have been categorised as *brownfield* developments, analogous to a project occurring in the construction industry on land that has already been occupied by previous buildings (the opposite type of project is green field) [Hopkins and Jenkins, 2008]. Brownfield construction projects are constrained by the presence of legacy artefacts from the previous construction, for example infrastructure (water, gas or sewage pipes, electricity or communication cabling) or old foundations. These artefacts may be expensive or impossible to remove or relocate, or may disrupt services to other buildings in the locality, so the new building on the brownfield site must be designed to accommodate them.

Brownfield (115)

The analogy transfers neatly into the domain of software engineering. The presence of pre-existing legacy software infrastructures constrains the development and future evolution of components. Even worse, other projects may be occurring simultaneously that cause other parts of the legacy infrastructure to evolve as well!

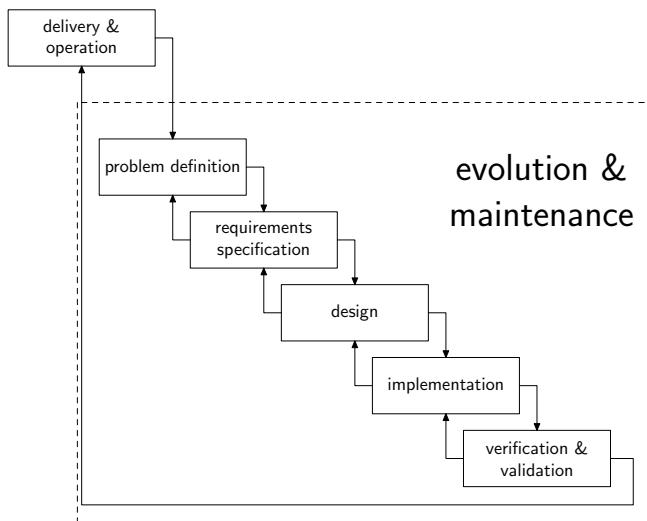


Figure 4.2: a close up view of the software evolution activity

## 4.2 V-Model

We need to investigate some alternatives that better address the risks affecting software projects, given the limitations we have identified in the waterfall model.

One limitation of the waterfall model is that it defers management of high risk requirements uncertainty (i.e. what does the client *really* want) until late in the software life-cycle during acceptance testing. The *v-model* of software development is an adaptation of the waterfall model that emphasises *verification* and *validation* activities during the software life-cycle. Verification and validation (V&V) are given particular definitions in the ISO 9000 quality management system<sup>1</sup>:

*“verification is the process of ensuring that technically the development is progressing correctly. All deliverables should be checked both for internal consistency and conformity with previous deliverables.”*

*“validation is the process of ensuring that the product being developed is what the user wants.”*

So, verification activities are used to determine if we are building the *product right*, i.e. does the complete product satisfy the formal requirements specification. Validation activities ask the broader question of whether we are building the *right product* for the customer, regardless of what the formal requirements specification document says.

Figure 4.3 illustrate the V-model of software development with (V&V) activities made explicit during the software life-cycle. The model is a waterfall,

---

<sup>1</sup><http://www.iso.org>

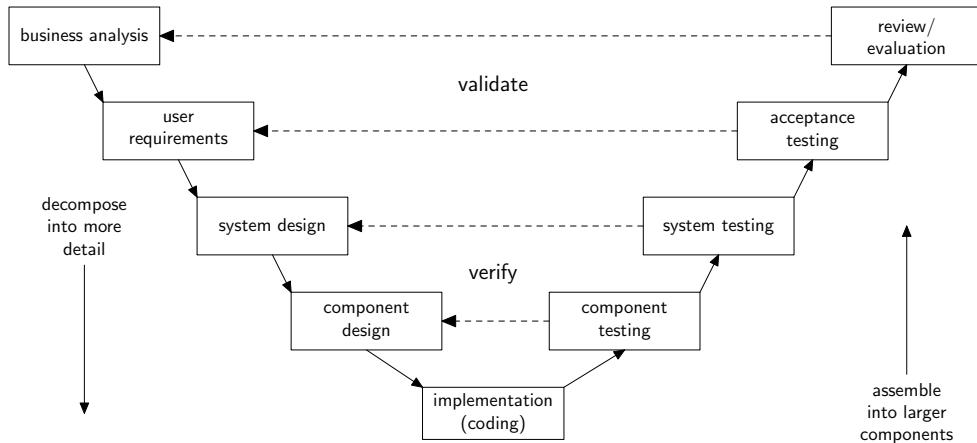


Figure 4.3: the V software development model

but shaped as a 'V', with development activities on the left hand side and V&V activities on the right. Development begins in the upper left hand side of the model with the analysis, requirements specification, design and implementation activities already discussed. The development proceeds by decomposing the development problem into finer levels of granularity.

V-model (117)

Each of these activities is matched by a V&V activity on the right hand side of the V. Component and system designs are *verified* during unit and system testing to demonstrate that the system satisfies its specification. As the system is assembled into larger units, validation activities are undertaken. Note that the V-model approach requires test plans to be developed *alongside* each stage of development. V&V activities are then managed using the previously developed plans.

### 4.3 Spiral Model

Although the V-model makes the relationship between development and V&V processes explicit, the risks associated with uncertain requirements may still be unaddressed until relatively late in the software life-cycle.

The spiral model is a *semi-iterative* software development process framework proposed by Boehm [1988]. The spiral model is similar to the waterfall, in that the relationship between major phase (requirements specification, design and so on) are assumed to be linear. However, the individual phases are treated as successive iterations of the software process, with the same activities occurring in each iteration.

In particular, the spiral model emphasises on-going:

- planning to review progress and set objectives
- risk analysis and management

Features of the spiral model (118)

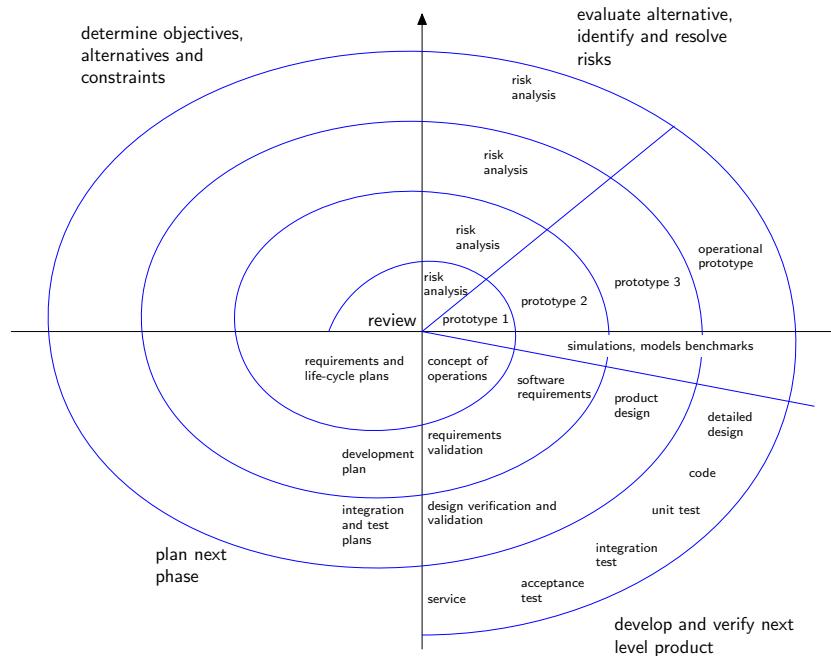


Figure 4.4: the spiral model showing activities

- prototyping to resolve uncertainty
- V&V driven development

Figure 4.4 illustrates the spiral process model (re-drawn from Sommerville [2007]). The software process is organised into a spiral to illustrate the re-occurrence of activities during each iteration. Work begins at the centre of the spiral and moves outwards in successive iterations. The spiral is divided into quadrants, with similar activities taking place in the quadrant during each iteration.

Objectives are decided for at the start of each iteration, along with the identification of major alternative options for progress and any constraints (budget, technology, etc.) affecting the project. Next a risk analysis is performed for the various options identified. A prototype is developed to address uncertainties identified during risk analysis.

A period of development and verification is then initiated for the main software development. Depending on the iteration of the spiral, activities here can range from the specification of requirements to the execution of an acceptance test with the client. Notice that regardless of the iteration, the development effort is concluded by one or more V&V activities. Requirements specification, for example, in the second iteration is matched by requirements validation.

Finally, a period of planning takes place to prepare for the next iteration. The outcome of the planning phase is a plan document for the next iteration.

The spiral model  
(119)

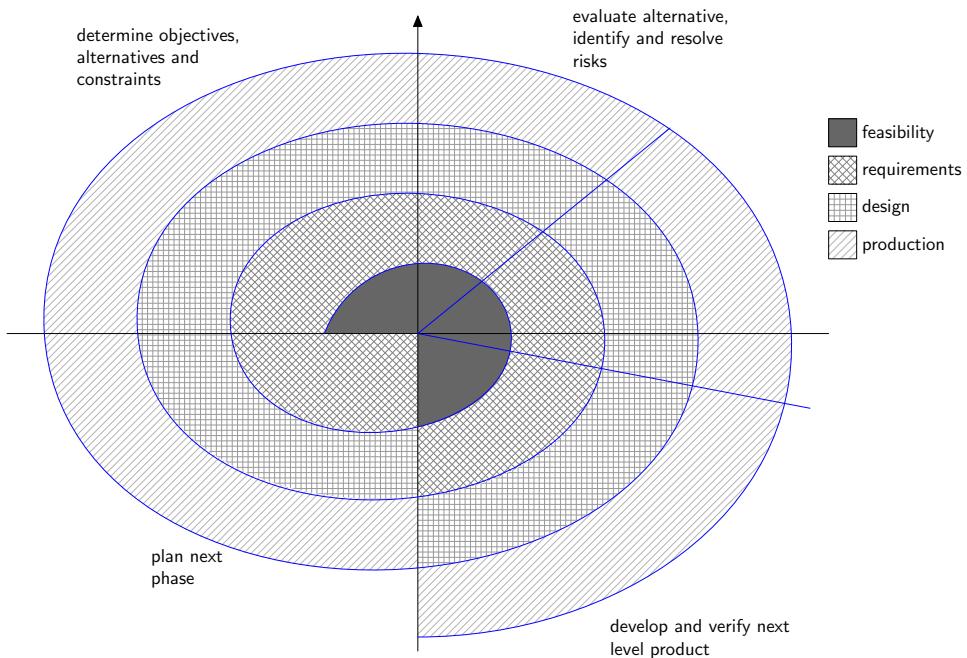


Figure 4.5: the phases of a software development organised into a spiral

Figure 4.5 illustrates a simplified view of the spiral model showing how each iteration is actually an activity in a linear software development model. From this perspective, the first iteration of the spiral model is a feasibility study, resulting in an early prototype, a business plan and a Concept of Operations.

During the requirements iteration, requirements gathering activities occur and the prototype is developed to resolve identified requirements risks. The result is a requirements specification document and a development plan. The design iteration leads to the development of a software design document, an implementation schedule and test plan. During the final iteration, the software system is actually implemented, tested and delivered.

We haven't talked much about specific software development practices for the spiral process model. Instead, the model can be thought of as a framework describing *process management* rather than a specific *development model*. In particular, the spiral model allocates explicit phases of development for risk analysis and quality assurance.

Many different practices can be incorporated into the spiral process in each iteration, including rapid prototyping, formal methods and the requirements elicitation and analysis techniques we have already seen in Chapter 11. The particular choice of practices or methods to be used is informed by the risk analysis phase of the spiral.

Iterations of the spiral model (120)

The spiral model as a framework (121)

Environmental pressures for change (122)

Requirements pressures (123)

## 4.4 Iterative Models

Some of the limitations of linear models of software development have already been discussed in Section 4.1. In particular, the resolution of requirements risks often occurs late in the software life-cycle because validation of requirements occurs immediately prior to deployment. At this point much of the budget for a project may have already been committed, entailing the risk that the wrong system (for the client's needs) will have been developed.

The previous discussion assumed that the unaddressed risk concerned incorrectly specified requirements, i.e. the development team were unable to discover or accurately record what the client wanted the system to do. However, even if the requirements are well established at the start of the project, they may be subject to change during a lengthy project life-cycle due to broader *environmental* and subsequent *requirements* pressures.

Environmental pressures refer to the context or market for which the software is being developed. Some examples of environmental pressures are:

- rapid technological change. Computing technology evolves so rapidly that the software may be obsolete even before it is delivered due to the introduction of competing products to the market place by competitors. This phenomenon has been described as 'perfect obsolescence' where well developed software becomes obsolete due to earlier market penetration of a competitor.<sup>2</sup>
- demand for return on investment. The quicker a software product is delivered, the sooner investors can begin to recover the cost of investment, and hopefully a profit.
- limited software 'shelf life'. There may be a deliberate policy by the developing organisation or client to only use the software for a relatively short period of time, perhaps as an interim measure until a more comprehensive solution is delivered by another development project.
- throw-away value software. Sometimes, software is deliberately developed speculatively to be discarded once the life-cycle is complete. Examples include the prototyping processes described in Chapter 16 and the development of software in the physical sciences domain [Morris, 2008, Sanders and Kelly, 2008, Howden, 1982, Miller, 2006].

Changes in the context of software development cause subsequent pressure for changes to the requirements specification such that:

1. rapid technological change causes feature competition;
2. customers demand the inclusion of new features in the system; and

---

<sup>2</sup>a colleague of mine claims that betamax video tape is an example of perfect obsolescence

- the requirements specification lacks stability.

Linear software process model life cycles often last too long to respond to the rapid changes in software and computing technology. We have already seen in the previous section how an iterative approach to software development can be used to manage risk within individual activities. This can be extended further by making the entire approach to software development iterative.

Iterative models are characterised by:

- incremental *releases* delivered to the customer or market at the end of each relatively short development cycles;
- early delivery of software to the client for review and feedback;
- deadline planning emphasizing what can be delivered within the current cycle of development, rather than trying to estimate how long it will take to deliver the complete functionality;
- development cycles directed by changes in the software context expressed as changes to the software's requirements specification; and
- concurrent software development activities within each cycle.

Characteristics of iterative models (124)

## 4.5 The Rational Unified Process

The *Rational Unified Process (RUP)* is a software development process model established by the Rational Corporation Rational. The process is *iterative*, beginning with the establishment of a *business case* and with an emphasis on relating the software development process to business concerns. RUP explicitly incorporates object oriented concepts and utilises the UML notation throughout the life cycle.

The RUP is characterised by the 'three P's, phases, processes and practices. The four phases of a RUP iteration are shown in (Figure 4.6). Each RUP phase is discrete, work is completed in one before progressing through a milestone to the next. If a project milestone for finishing a phase is not reached, more work is done in that phase before continuing. Consequently, The different phases are internally iterative, and will cause the project requirements and design to evolve over time. For example, the set of use cases will need to be refined and extended as more is learnt about the problem domain. More use cases may also transpire during the construction and transition phases.

RUP phases (125)

Conversely, RUP processes (Table 4.1), or *work flows*, describes activities conducted within an iteration that will occur in one or more phases. Core engineering processes are concerned with the development of the software product itself. Supporting processes are concerned with the surrounding project management, tracking progress towards goals and facilitating the project team in their work.

RUP processes (126)

	<b>business modelling</b>	understanding how the development project will affect the organisation. In particular, business workflow models are developed as a common interface between software and business engineers
engineering	<b>requirements</b>	eliciting, gathering, refining of the system specification describing <i>what</i> the system will do
	<b>analysis &amp; design</b>	development of a design model describing <i>how</i> the proposed system will satisfy the specification
	<b>implementation</b>	construction and unit testing of software components, organised into sub-systems
	<b>testing</b>	verification of the completed system against the specification document and the identification and removal of systemic defects
	<b>deployment</b>	preparing and packaging the software release, or installing the software and migrating the organisation and users to the new version
supporting	<b>configuration &amp; management</b>	on-going management of the integration of new and modified artefacts into the product
	<b>project management</b>	tracking progress towards objectives and managing risks
	<b>tool provision</b>	provision, configuration and maintenance of the Software Development Environment (SDE) and Computer Aided Software Engineering (CASE) tool set

Table 4.1: RUP work flows

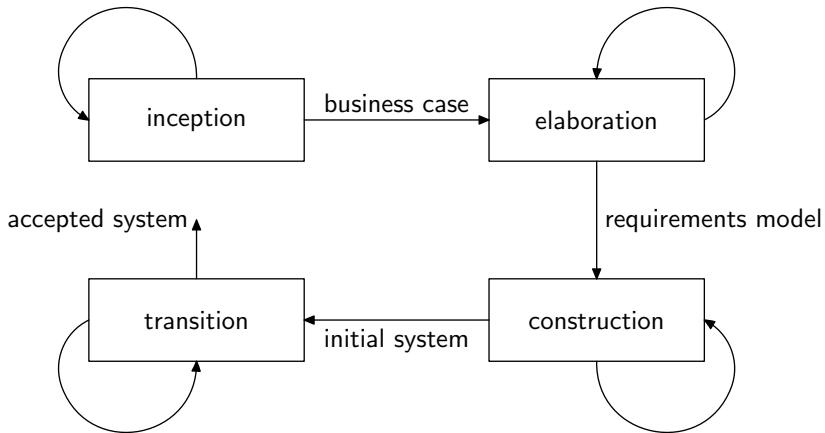


Figure 4.6: RUP phases

Different processes will require different amounts of effort within each phase of an iteration, as illustrated in Figure 4.7. For example, implementation will begin during the elaboration phase, as early prototypes are developed, will predominate in the construction phase as the main product is built and will continue during the transition phase as testing uncovers defects.

The RUP also advocates practices (Table 4.2) to be adopted by software developers. These are not specific to any phase, but should be conducted throughout a project's life-time.

Notice the emphasis on tool or automation support for the software process throughout RUP, with an emphasis on tool supported modelling, development and project management. Tools include compilers, model and source code editors, test suites, configuration management services and issue tracking systems.

Figure 4.8 illustrates the inputs and outputs to the *inception phase* of the RUP. The main input to the phase is a preliminary problem description, to-

RUP engineering  
phase effort (127)  
  
RUP practices (128)

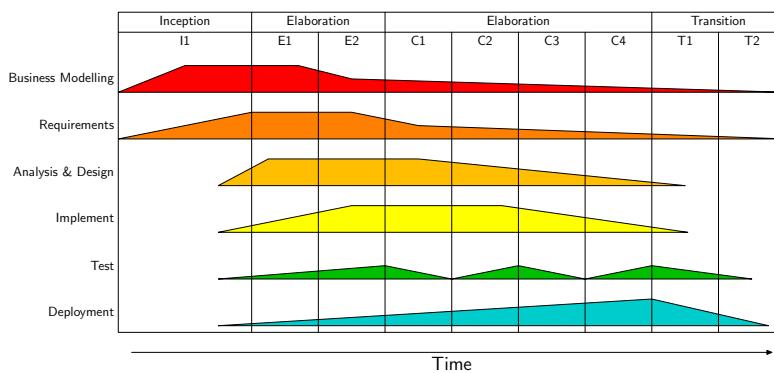


Figure 4.7: process effort in different RUP phases, adapted from Jacobson et al. [1999]

---

<b>develop software iteratively</b>	small changes to large software systems often reduce the amount of uncertainty
<b>manage requirements</b>	the system requirements will evolve during the project, so it is important to track these changes and their consequences for the development plan
<b>use component based architectures</b>	encapsulating software in components encourages software re-use and design for re-usability, reducing development costs for new projects
<b>visually model software</b>	visual models ease comprehension of large scale software projects. Employing a structured notation, such as the UML, helps to ensure consistency between different models
<b>verify software quality</b>	high quality software is more likely to be defect free
<b>control changes to software</b>	as software evolves with new features, defects may be introduced, removed and re-introduced. Implementing a process for changes to a source code base helps to limit these problems

---

Table 4.2: RUP practices



Figure 4.8: inception phase of the RUP

gether with an initial budget and intended scope. The purpose of the inception phase is to establish a business case for the proposed project, estimating the costs and duration against the potential benefit. These may lead to alterations in the project budget and duration as expressed in the business case. An acceptable business case for a project will obviously need to demonstrate a reasonable benefit over cost within a realistic time-scale! The estimates for the business case are parameterised by the available budget for the project and anticipated scope.

Once the business case has been prepared, it is necessary to elaborate a comprehensive understanding of the problem domain and establish the requirements for the system as a complete specification. Figure 4.9 illustrates the outputs from the elaboration phase:

- a *domain model* illustrating the key concepts in the problem domain and the relationships between them;
- a *set of use cases*, describing the available system interactions a user can perform;
- a description of the *non-functional requirements* of the system. Non-functional requirements refer to the emergent, measurable properties of the end system, such as performance, throughput and reliability;
- a *development plan*, including a schedule or work, project milestones and an estimated delivery date; and
- a *risk management plan* addressing the risks to the project, their consequential impact on the project's success and any strategies to be adopted

Inception phase (129)

Elaboration phase (130)

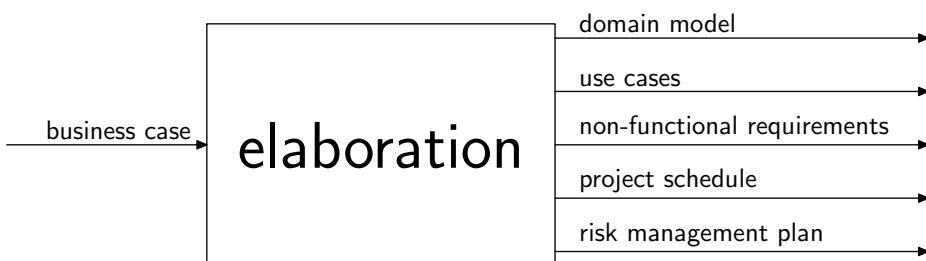


Figure 4.9: elaboration phase of the RUP



Figure 4.10: construction phase of the RUP

to mitigate them.

#### Construction phase of RUP (131)

The development process is driven by use cases which describe the functional requirements, or features, for the system, which in turn dictates the system architecture.

Figure 4.10 illustrates the inputs and outputs of the construction phase. The construction phase is dominated by detailed system and component design, implementation and unit testing. The purpose of the phase is to prepare a release of the software for delivery to the client. The inputs come directly from the elaboration phase of the process. The outputs are:

- a design document containing the collection of models developed from the use cases and domain model, following the method described in Chapter 15;
- an architectural model of the system illustrating the components, packages and classes, and the relationships between them. The architectural model should indicate the relationship with external libraries and components, but not detail their design;
- a test plan and test suite for the system. A test plan describes the collection of test cases to be applied to the system and a *testing strategy* describing the approach to testing and integrating components. The test suite is a harness for executing the automated aspects of the test plan.
- the implemented deliverable that will be the release for this iteration of the process; and
- any accompanying user documentation for the delivered system, e.g. installation procedures, quick start guides or reference manuals.

#### Transition phase of RUP (132)

Figure 4.11 illustrates the final, transition phase of the RUP cycle. The phase is concerned with deploying the software system within its organisational context, or distribution to customers. Installation, user training and *beta* testing according to the test plan takes place during the transition phase. *Legacy systems* or previous versions of the new software may need to be removed from



Figure 4.11: transition phase of the RUP

the organisation in parallel whilst causing minimal disruption. The inputs are the same as the outputs from the construction phase.

We have now considered the separate phases of a single RUP cycle in detail. Now let's look at how progress between phases and cycles is monitored in RUP. Figure 4.12 shows a number of cycles of the RUP process, separated by releases of the software. The decision to release a new version of the software is taken during the transition phase, when the features agreed for the cycle have been implemented, along with associated use case or system non-functional requirements.

The figure also shows the collective phases of the first RUP cycle. The phases are divided into a number of *milestones*. Milestones are pre-defined objectives set in the project plan that must be met before the project can proceed. The figure shows a number of 'pre-defined' life-cycle milestones separating the phases of the cycle, which are passed when each of the outputs identified in Figures 4.8, 4.9, 4.10 and 4.11 are satisfactorily delivered.

Milestones and  
Releases in RUP (133)

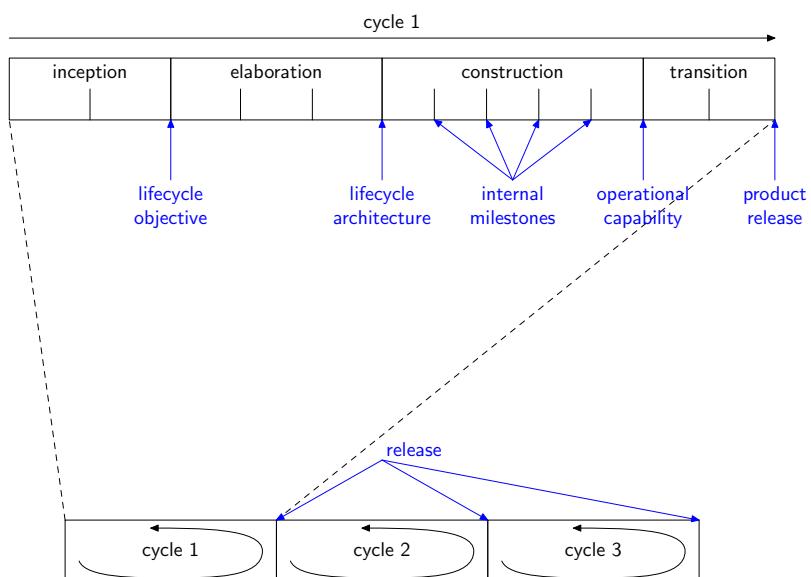


Figure 4.12: milestones and releases within the Rational Unified Process model

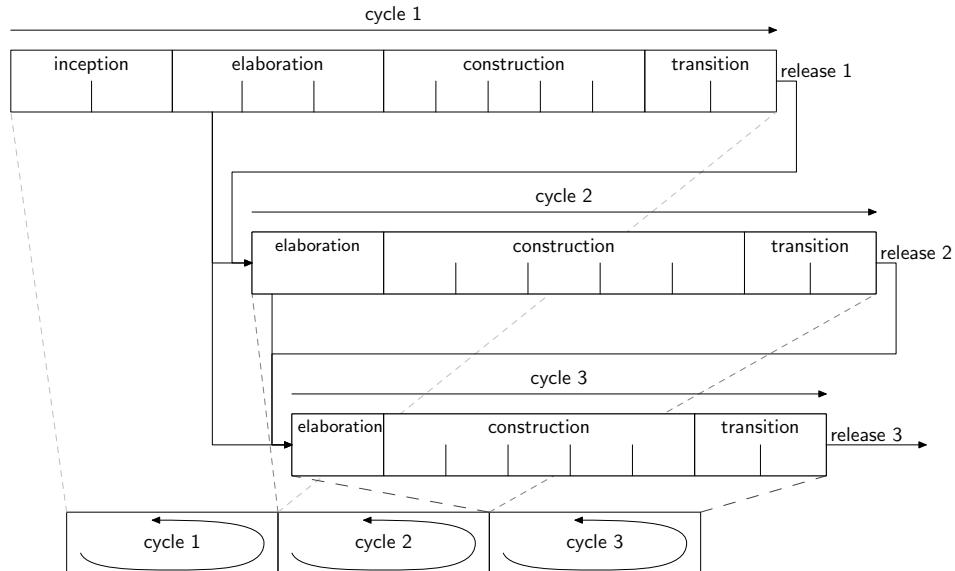


Figure 4.13: subsequent cycles of the Rational Unified Process model

### Subsequent cycles in RUP (134)

Figure 4.13 illustrates the phases of subsequent cycles of the RUP model. We have already seen in Chapter 11 that large software projects can be divided into smaller increments by prioritising the desired features of the system (e.g. using the MoSCoW rules). The figure shows a succession of cycles of the RUP model delivering incremental releases.

As can be seen in the figure, the inception phase occurs at the beginning of the first cycle only, when the business case and preliminary requirements for the system are established. In addition, the elaboration phase in the first cycle is substantially larger than in later cycles because the bulk of requirements and overall architecture of the system should be established at this point.

Subsequent cycles in the development begin with the release from the previous cycle and the collection of un-implemented use cases identified during the elaboration phase. A further sub-set of use cases is selected for implementation or revision, according to the priorities established in the project plan. The use cases are then developed in more detail and the design and implementation of the previous release are revised to incorporate them. The system is then tested and released as for the first cycle. Notice that further use cases may be identified during elaboration as details are added to currently un-implemented use cases and further requirements gathering takes place.

The RUP model assumes that the overall concept of the system can be established at this stage of the project, since inception and early elaboration activities are not intended to be re-visited. This has been called an “architectural centric” approach to software development, in which development is directed by the use cases identified for the system. Unfortunately, this means that the

software should still have a relatively stable set of functional requirements at the start of the software process.

## 4.6 Agile Methods

Agile methods refers to a class of particularly ‘light weight’, iterative, rapid software development models. Agile methods focus on the delivery of the software product as quickly as possible so that evaluation, feedback and subsequent revisions can be undertaken.

Agile methods bear some similarity to rapid prototyping as a software development model. However, agile methods also characterised by principles and practices that require a disciplined approach to software development in order to manage the risks associated with rapid prototyping. Agile methods are characterised by:

Agile methods (135)

- incremental delivery of *working* software to the customer, with releases as often as 1-3 weeks;
- customer involvement throughout a project, for continual feedback and resolution of uncertainties. This often means that a representative of the customer is a member of the project team and is co-located with the developers;
- small project teams to reduce communication overhead. Larger development efforts must be divided into smaller teams;
- reduced software overheads, with less emphasis on formal modelling or documentation;
- an emphasis on working code over design. Proponents of agile methods argue that the design of a system is best developed gradually as the problem becomes better understood. In addition, the feasibility of a design is best demonstrated in working code;
- the automation of tasks where possible, but in particular for unit testing, defect tracking, configuration management, documentation and software artifact build and deployment;
- light-weight documentation in order to reduce the risk that the documentation becomes out of date. In addition, source should be written to be *self documenting* as far as possible to make further documentation unnecessary. That is, the source code should be sufficiently clear that further explanatory notes are not needed; and
- value of people over processes. Agile advocates argue that individuals and the communications between them are more important than rigid adherence to particular processes and tools.

---

<b>scrum</b>
<b>crystal clear</b>
<b>lean</b>
<b>feature driven development</b>
<b>adaptive software development</b>
<b>extreme programming (XP)</b>

---

Table 4.3: some agile software development methods

Risks of agile methods      However, agile methods are themselves prone to risks due to the assumptions implicit in the principles adopted:  
(136)

- lack of customer engagement. This may be because the customer or doesn't prioritise the project. Alternatively, it may not be possible to identify a suitable representative of the customer, if for example, the development is a speculative effort, without a pre-identified market or need;
- stakeholder conflict. There may be many different stakeholders for a project with conflicting requirements for the software. The customer representative may not convey this complexity to the software project team;
- contract definition is usually simpler when there is a well-defined software product to be delivered to the customer. However, agile methods deliver software incrementally with no pre-defined end-goal. This risk can be mitigated by using incremental contracts, however, this itself can add overhead to the project (since each iteration must be contracted) and the development team cannot be confident of a stable income stream;
- loss of organisational memory (how we do things around here) due to a lack of documentation and compartmentalisation of agile teams. As a consequence, mistakes may be repeated in several parts of an organisation;
- poor code quality because software is developed rapidly without application of professional discipline. Several agile methods propose *constant refactoring* and *test first coding* to mitigate this risk; and
- developer cooperation/team cohesion. Agile methods are characterised by small groups of people working very closely together. The effectiveness of the group may be limited by personality conflicts or tensions within the group.

Some agile methods  
(137)

Table 4.3 lists some agile software development methods. The next section describes the specific practices of one agile method, XP, in detail. Many of the practices are also adopted by other agile methods, which You can find out further information on from the recommended reading.

## 4.7 Extreme Programming

Extreme programming [Beck and Andres, 2005] is a popular agile development process used in many organisations. Table 4.4 summarises the agile rules and practices incorporated into the XP process, categorised into four types of activity: planning, designing, coding and testing.

Rules and practices of XP (138)

### 4.7.1 Planning and Specification

Figure 4.14 illustrates the *concurrent* flow of activities during an XP iteration, i.e. activities occur in parallel. The current iteration is the central focus of an XP development. The iteration is managed through release planning, which is informed by:

- a collection of *user stories* describing desired, which represent the requirements for the development;
- the *project velocity* which gives an estimation of current progress; and
- estimates of development time derived from *spike prototyping*.

The result of release planning is a *release plan* which will be used to direct efforts during a single iteration, typically between one and three weeks. The plan identifies the user stories to be implemented in the current iteration and an anticipated delivery date.

The XP process model (139)

Planning *within* an iteration occurs on a weekly basis. Iteration planning meetings take place amongst the whole development team - some agile advocates have argued that iteration planning should be conducted standing up to make things happen more quickly. During iteration planning, progress against

---

<sup>3</sup><http://www.extremeprogramming.org>

<b>planning</b>	for each release on a weekly basis user stories project velocity
<b>designing</b>	simplicity prototyping
<b>coding</b>	pair programming standards frequent integrations and builds constant refactoring
<b>testing</b>	automated testing test first coding

Table 4.4: extreme programming practices and rules

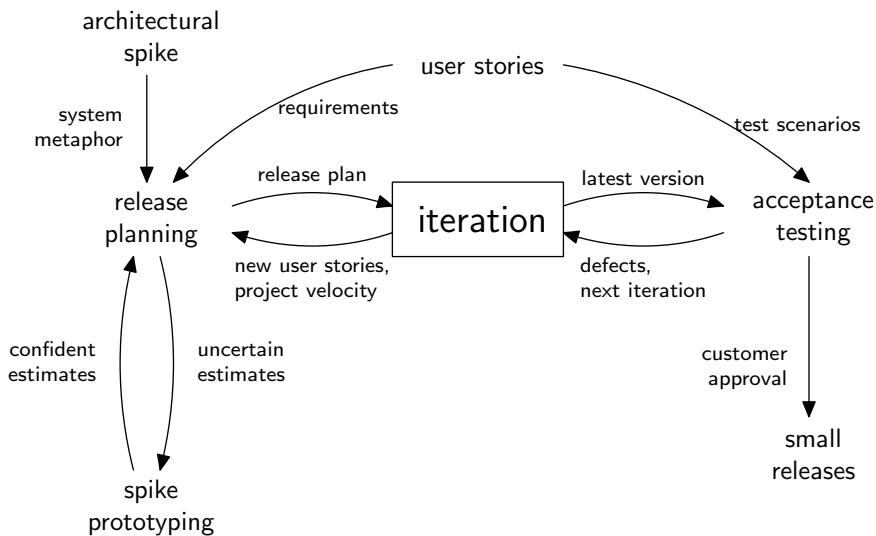


Figure 4.14: overview of the XP process (re-drawn from Wells<sup>3</sup>)

the release plan is measured and the list of identified *defects* (bugs) reviewed. Technical risks affecting the project are identified for investigation during spike prototyping. If necessary, the release plan is revised to accommodate delays or faster progress.

User stories are similar to the use case scenarios described in 11, except they tend to be briefer, and contain less detail. They describe the use of a particular feature of a system from the perspective of a user. Figure 4.15 illustrates a user story for the branch library system.

User stories (140)

User stories are usually hand written and brief, consisting of only two or three sentences and possibly a sketch of associated user interface features or process flows. If stories need more than this amount of information, they should

### Keepers

As the branch administrator, I need to be able to manage who is and who can keep books within the department. Book keepers can only be members of academic staff.

Estimate: 2 days

Figure 4.15: an example XP user story

probably be divided into two or three smaller descriptions.

The descriptions are written on small pieces of card<sup>4</sup> which are displayed on a wall in the project team's location. User story cards can be arranged on the wall to visually represent different aspects of the project, for example:

- the system functionality or architecture by grouping related stories;
- the project's progress by separating completed and uncompleted stories;
- the project's exposure to risk by grouping stories by priority; or
- the objective of the current iteration by grouping the 'in progress' cards.

User stories are typically developed in collaboration with the project customer, either during an initial elaboration iteration, or as a result of discovering new stories during development and acceptance testing. Of key importance, the system feature should be described in a way that the user and customer can understand. In particular, no technical information concerning implementation details is included on the card.

User stories are used to develop acceptance tests for a system and as a basis or more detailed specification. The more detailed requirements for a system are expressed as *story tasks*, derived from user stories. Tasks describe the functionality that will need to be implemented in order to realize the user story (and thus satisfy the related acceptance tests). The tasks for the user story given in Figure 4.15 are:

1. Search for users and check status ( 0 days – already done in search users)
2. Add an academic staff member as a keeper (1 day)
3. Remove a keeper (1 day)

The relationship between stories and tasks is many-many rather one-many. Typically a story will be realised by a number of tasks. However, some tasks will re-occur in different stories. These duplications need to be resolved during story elaboration.

Once the set of tasks to be implemented is established they must be allocated amongst the project team. This can be done by following several different strategies:

- the team or team manager can select the most appropriately developer;
- developers can volunteer for tasks; or
- tasks can be assigned at random.

---

<sup>4</sup>For some reason, many references assert they have to be 5x3in!

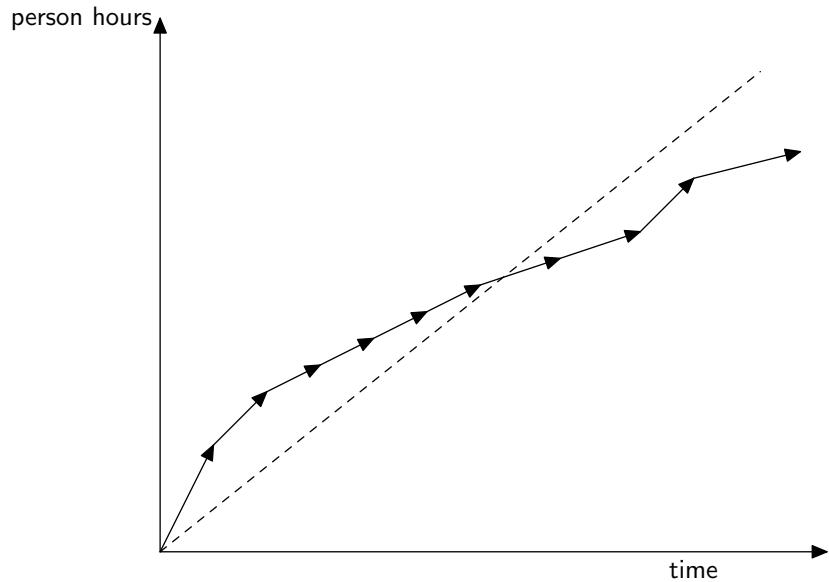


Figure 4.16: an example project velocity

Each approach has advantages and disadvantages. For example, asking developers to volunteer for tasks may mean that high risk tasks are not considered until late in the iteration. Alternatively, assigning tasks based on the competency of developers may mean that in-experienced members of staff do not learn new skills.

Once tasks are assigned, developers then provide an estimate of how long they think the task will take to implement, typically measured in 1-3 days. If a task is estimated as taking longer than this it should be reviewed and possibly divided into a larger number of smaller tasks. Alternatively, tasks that are too trivial should be grouped together. The aggregate of task time can then be used to compute the total time for implementing the user stories for iteration.

The task estimates can be used to calculate the *velocity* of a project by summing the project progress on completed user stories and comparing this against the person-hours of effort expended so far. The velocity gives an indication of estimation accuracy for user stories, as well as the likelihood of a project meeting the estimated release date.

Project velocity (141)

Figure 4.16 illustrates a project velocity for a project that has begun to fall behind schedule. Advocates of agile methods argue that progress on software projects is governed by four variables contained in the equation below (note that the relationship is only approximate):

$$\text{available time} \times \text{cost} \approx \text{scope} \times \text{quality}$$

Managing delays (142)

The cost of a project can be computed from the number of developers working in the team over the iteration (likely to be the dominant budget factor).

The time is calculated as described above from the story tasks. The scope of a project is defined by the user stories selected for implementation in an iteration. Finally, the quality of software can be defined by metrics such as defect rate, or *mean time to failure*.

The implication of the equation is that there are four options for a development team should their project experience delays:

- postpone the release date to allow more time for development;
- increase the budget to bring in more developers to the team;
- compromise on quality assurance by reducing testing and defect correction efforts; or
- reduce the scope by delaying some user stories to the next iteration.

Postponing a release date may mean delaying the point when a customer sees an update to the system and provides feedback, increasing the risk that the system development may diverge from their needs. Increasing the budget for a project can be politically difficult if an organisation has already invested substantially. In addition, adding more developers to a project may not actually have any effect at all on the schedule, or may even make it worse. Brooks famously referred to this as the 'Mythical Man Month' [Brooks, 1995].

On the other side of the equation, compromising on quality risks failure for the entire project because in the rush to complete all agreed features, defects may be introduced that make the system unworkable. *Reducing project scope* by delaying some of the stories to be implemented to the next iteration is often the most professional approach, since the customer is more likely to receive a (at least partially) functioning system on the scheduled release date.

#### 4.7.2 Testing and Coding

The first step when implementing a user story or task is for the developer to derive *acceptance test cases* for the story, in agreement with the customer. This means that test cases are developed *before* the features of the story are implemented. An implication of this approach is that the system should initially *fail* the new test cases, because the relevant features have not been implemented.

This *test first* approach to development extends to the implementation of individual software components. Before implementing a component, the developer specifies the component interface and then develops a number of test cases to exercise it. This forces the developer to design components to be testable, i.e. exposing the necessary functionality to demonstrate that a component passes or fails a test case.

Figure 4.17 illustrates the arrangement of test harness, interface and target component in a UML class diagram. First, the interface for the component

Test first development  
and automation (143)

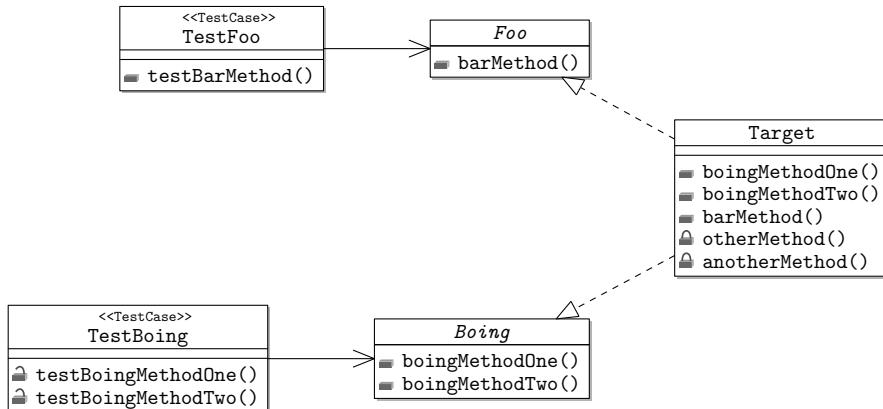


Figure 4.17: test first development

is specified as a collection of publicly accessible methods., then a test case is implemented against the interface. Finally, the component itself is implemented. If the component is to implement an additional interface, then the process is repeated. Note that the test cases are *only* implemented against the methods declared in the interfaces, not the component itself.

Test cases are best implemented within an automated *test harness framework*. There is a family of \*Unit testing frameworks for different programming languages, for example JUnit<sup>5</sup> for Java, CppUnit<sup>6</sup> for C++, PyUnit<sup>7</sup> for python and even fUnit<sup>8</sup> for Fortran 90.

Test first development shifts the problem of writing correct code to one of writing complete test cases. Assuming the test cases are complete, once all test cases are passed, the new feature can be integrated into the main system. However, writing complete test cases is an extremely challenging activity that requires considerable discipline. Writing complete test cases does not necessarily mean writing *exhaustive* test cases. There is an excellent anecdote on this issue, entitled “if you have 10,000,000 test cases, you probably missed one”<sup>9</sup>. We will return to the problem of writing good test cases later on.

During integration, the automated test cases are added to a *test suite* that can be automatically executed each time the complete system is re-built. This requires:

- the management of source code for both the system and the test cases in a version control repository, such as Subversion<sup>10</sup>; and

<sup>5</sup><http://www.junit.org>

<sup>6</sup><http://cppunit.sourceforge.net>

<sup>7</sup><http://pyunit.sourceforge.net>

<sup>8</sup><http://nasarb.rubyforge.org/funit>

<sup>9</sup><http://horningtales.blogspot.com/2006/09/exhaustive-testing.html>

<sup>10</sup><http://subversion.tigris.org>

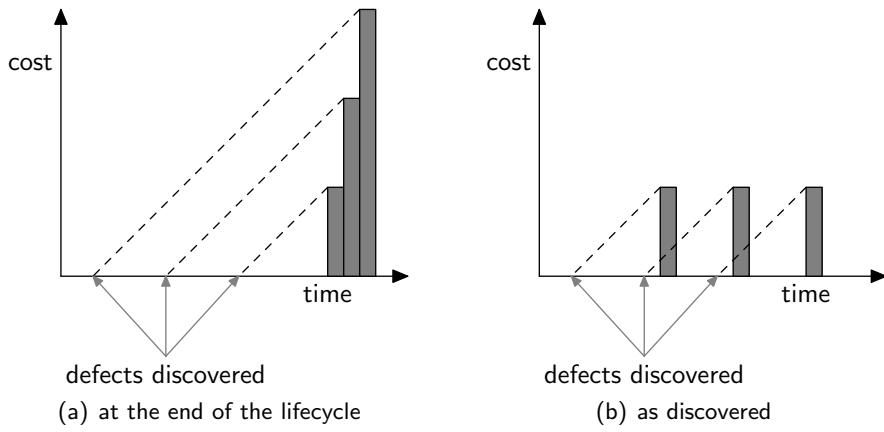


Figure 4.18: cost of fixing defects (redrawn from Beck and Andres [2005])

- the development of an automated build, deployment and test execution script for the complete system. Again, a number of frameworks are available for developing build scripts, including Apache Ant<sup>11</sup> or the \*nix `make` utility.

In agile projects, component build, integration and testing occurs frequently (at least once a day) so that progress towards completion of the current iteration can be measured and used for planning. Beck argues that it should be possible to automatically build a software system within 10 minutes [Beck and Andres, 2005].

Frequent automated testing means that the introduction of defects into the system is detected earlier, reducing the cost of correction. Figure 4.18(a) illustrates the introduction of several defects into a system in which testing occurs late in the software cycle. The cost of correcting the defect grows over time as the defects become embedded in the system (the development team may even implement features that *depend* on the defect). Figure 4.18(b) illustrates the same collection of defects and the reduced cost of detecting and fixing defects as they are introduced into the system.

The use of automated test harnesses also reduces the likelihood that defects are *re-introduced* into the system. Defects are either demonstrated by an existing test case, or when an error is reported by users, the presence of the responsible defect should first be demonstrated by implementing a new test case. Automated suites of test cases can be executed very cheaply as *regression tests* to demonstrate that previously fixed defects have not been re-introduced.

A consequence of frequent integration, testing and feedback is that the software source code will be subject to continuous inspection and alteration. Fixing defects is often an ideally opportunity to *refactor* (clean-up) a component

Fixing defects (144)

<sup>11</sup><http://ant.apache.org>

Constant refactoring (145)	implementation. Refactoring activities include:
	<ul style="list-style-type: none"> <li>• improving test case completeness;</li> <li>• improving the design by reducing coupling between components and increasing internal cohesion;</li> <li>• improving the readability of code, by ensuring consistency of identifiers for example; and</li> <li>• providing source code documentation.</li> </ul>
Pair programming (146)	<p><i>Pair programming</i> is a technique for combining source code inspections with the development activity itself. Pairs of programmers sit at a single terminal and work on one problem: one (the pilot) controls the mouse and keyboard; the other (the navigator) observes the work, identifies mistakes and suggests corrections.</p>
	<p>Collaborative work on development improves code quality, because the code is reviewed as it is written - defects are detected before they are even fully implemented. The practice does however require the developers to be open to criticism of their work by colleagues, but also be able to deliver criticism in a constructive way. Pair programming also requires discipline, because it can be tempting for one member of the pair to dominate the work. Alternatively, the navigator may lose concentration quicker than the pilot, so it can be useful to rotate roles frequently so that both members of the pair are engaged in the work.</p>
Simplicity in design (147)	<p>There are several alternative strategies for allocating team members to pairs, from random allocation to pairing experienced developers with newer members of the team. The optimal arrangement will to a certain extent depend on the personalities and dynamics within the team - some people just don't work well together.</p>

#### 4.7.3 Prototyping, Design and Architecture

Unusually, we have left a discussion of the role of software design and architecture in a software method until the end. Many agile methods have an ambivalent relationship with the notion of software design, instead arguing that a system's design will emerge as the customer's requirements become clearer. The key design principles for XP are *simplicity*, i.e. designs that are as easy as possible to comprehend for both the customer and new members of the project team; and *small up-front design*, i.e. avoid committing to large design solutions early in a software development.

In general, XP places less emphasis on designs that attempt to anticipate future change, since this can introduce unnecessary complexity. However, designs can be adopted to accommodate future change, by for example, adhering

to principles of low coupling between and high cohesion within components. In addition, continual refactoring allows design to be reviewed and re-implemented if necessary.

The use of a *spike prototype* can be used to resolve high risk issues affecting a project. The risks can be caused by uncertainty regarding the system architecture or estimates for implementing a user story.

Spike prototyping is similar to the prototyping process described in Section 16.4.1. The prototype is developed rapidly, with the conclusions of the activity fed back into the development process for the main system. The spike prototype can be developed by a single member of the project team working alone without applying the usual XP practices such as re-factoring and pair programming. However, the prototype should be discarded and the functionality re-implemented as part of the main development effort, once the risk represented by the design issue has been resolved.

In fact, as can be seen from Figure 4.14 the start of an XP development is informed by an initial spike prototyping phase in which the *system metaphor* is established. System metaphors are similar to architectural patterns. The metaphor should be used to identify the key components of the system architecture in an accessible manner.

Spike prototyping  
(148)

System metaphor  
(149)

## Summary

We have reviewed the problems of linear software process models, in particular the challenge of uncertain and unstable requirements. We have reviewed a number of iterative software models from the Rational Unified Process to the Extreme Agile method. A common thread through all these different process models, is that professional software development is concerned with reducing risk and requires a rigorously disciplined approach to planning and development.



# Chapter 5

# Project Management

## Recommended Reading

**PLACE HOLDER FOR sommerville10software**

Recommended reading  
(150)

**PLACE HOLDER FOR ludi06student**

Chapters 3 and 4 introduced the activities that can occur within a software system life cycle, and the different ways these activities can be organised. As we have seen, such projects are often large, complex endeavours and can last many years and involving many different people. The end products of this process may be quite uncertain when the project begins and the scope of the system to be built (as understood by the project team) may change as requirements are elaborated and development proceeds. Project *management* is concerned with reducing the risks associated with these uncertainties. This chapter explains different ways of organising, managing and planning team efforts within these different models.

## 5.1 Working in Groups

Many software engineering projects are beyond the capabilities of a single software engineer to deliver within a reasonable time-scale. In addition, many software projects persist for longer than the length of a typical engineer's service at a particular organisation, or even an entire career. Consequently, different people (and groups of people) will contribute effort to a software development project at different points in time.

Consequently, software project teams must be organised, with an appropriate structure to address the particular problem the project poses. In addition, this structure must be communicated effectively to new members of the team as they join the project; and the structure must be continually reviewed and adapted as the nature of the project changes and team members acquire new skills and experiences.

Working in groups  
(151)

Chief programmer team model (152)

### 5.1.1 Models of Team Organisation

There are numerous models for organizing a software development team. For example, in the *chief programmer model* authority resides in single team leader (chief programmer). The model was originally proposed by Brooks [1995], to replicate the structure of medical surgery team led by a senior surgeon.

Information and control is organised top-down and follows well defined paths. Each member of the team has well-defined roles for which they have explicit training and expertise.

A disadvantage of the approach is that it can cause teams to be inflexible when new situations or skill-requirements emerge, since the team organisation is dedicated to addressing a particular problem structure.

At the other extreme, the *laissez-faire* model provides much more individual freedom to the members of the team to contribute to the project in their own way. Roles within the team are not rigidly defined. Instead, each team member investigates the project problem, and works out for themselves how they will contribute to the project goals, based on their own preferences and skills. The team provides a forum for discussion, communication, coordination (and hopefully) agreement amongst team members, rather than as a means of directing efforts. Rules and conventions within the team emerge over time.

Laissez-faire (multi-user dungeon) model (153)

A colleague used to describe laissez-faire as being like a Multi-User Dungeon computer game (now called massively multi-player), because all the players have different skills, abilities and personal goals. The players will cooperate in order to achieve common goals (e.g. reaching the next level), but do so by bringing their personal skills to the team voluntarily.

A laissez-faire approach is commonly adopted in open source projects, either to get the project started, or as a mechanism for continuing support when a vendor has decided to divest a product. Raymond [2001] has described the effect of voluntary contributions to an open source project.

Role based (sports team) model (154)

In the *role based* model, the team is built around a set of roles. Roles can be either long-lived or created to handle specific tasks. Consequently, the assignment of roles to team members can also be flexible, depending on the circumstances of a project. Some roles, such as project manager may persist for the duration of the project, and only ever be assigned to a single person. Other roles may rotate within the group, such as project secretary, so that each team member contributes to some of the project's meetings. Alternatively, roles may be created for specific tasks, such as a requirements capture team.

Despite the flexibility, role based teams tend to have clearly defined mechanisms for making decisions, either democratically or through a management system. The authority for decisions may be distributed amongst different members of the teams for different aspects of the project. The general manager might allocate roles, for example, but the requirements manager might decide how best to document specifications.

### 5.1.2 Roles in Teams

Regardless of the exact structure of a team, a number of software development team *roles* tend to re-occur. Some common roles in a software team are discussed below.

- The *project manager* has overall responsibility for ensuring that the software product(s) from a project are delivered to schedule and within budget. They maintain the project schedule and track progress against these targets. They may also have broader organisational roles, such as developing the abilities of other members of the team.
- The *customer liaison* acts as the point of contact between the development team and project customers and users. The customer liaison communicates messages between the project team and the customer, such as progress updates (to the customer) and changes to requirements (to the project team).

Roles in a software development team (155)

A customer liaison may also arrange meetings between members of the project team and members of the customer's organisation for activities such as requirements gathering. In some software development processes, such as XP, the *customer* appoints the customer liaison from within their own organisation, rather than from the project team.

- The *librarian* maintains, organises and archives project documentation, including the project plan, requirements specification, design document and test plan. The librarian often works with the quality assessor to manage and archive different versions of the project documentation as they are updated and new versions produced.
- The *secretary* is responsible for producing and maintaining records of formal project meetings. During a meeting, a secretary will not take a full part in the discussions; rather they will document the key points discussed and the decisions made. After the meeting, the secretary will circulate a written record (the minutes) of the meeting to other team members using the communication tools agreed by the team. The minutes should also note the date, time location of (and agenda for) the next meeting, if this is not standardised.

One convenient way of producing minutes is to type them directly into a wiki editor as the meeting progresses. The progress of the meeting can thus be tracked in real time if the rest of the team can see the minutes as they are produced. This also allows corrections to the minutes to be made during the meeting, rather than after the fact.

The secretary may need to work with the project librarian to maintain documentation. Alternatively, these roles can be combined. It can also

be useful to rotate (or at least alternate) the role of project secretary, so that everybody can contribute fully to some of the project meetings.

- The *toolsmith* is responsible for maintaining and configuring the software tools used by the team. Software tools used include the project version control repository, development editor, integrated development environment, project management software, test harness framework, software build and deployment framework, project wiki and other documentation tools.

They are also responsible for researching new tools that might be used by the team, and providing training on the tool interfaces.

- The *quality assuror* is responsible for organising and maintaining the quality assurance process for the software development project. Quality assurance is concerned with both the software system itself and the associated documentation. Consequently, the quality assuror should maintain a release process for each new version of a document, which crucially includes a period of review and revision. The quality assuror also maintains a process for tracking and remedying reported defects in the software system and associated documentation.
- The *test manager* develops the test suite for the software system, based on the agreed test plan. A test suite consists of a large number of tests derived from the requirements specification, design document, implementation and from defect reports.;
- The *chief architect* is typically an experienced software engineer, charged with making high level design decisions for a system. A chief architect may well put contentious design decisions to the whole team for discussion. However, they will retain the final responsibility for choosing between different software designs.

#### Team organisation – roles (156)

The exact roles and assignments will depend on the characteristics of a project team. In order to decide on how to organise your team, you should consider each individual's:

- attributes;
- skills;
- personality;
- circumstances; and
- professional development aims.

Roles can be defined by the tasks they are required to undertake. Roles and team members can be assigned flexibly, i.e remember that:

- a role can be assigned to different team members at different points in time;
- more than one role can be assigned to a team member; and
- more than one team member can be assigned to the same role.

In addition, there may also be group roles for subsets of the team, particularly for larger software projects. For example, there may be a requirements capture team.

### 5.1.3 Group Communication

You should expect to have *regular, planned* meetings with clear *objectives*. At minimum your team should aim to meet once a week, at the same time and in the same place. Establishing a regular meeting time and venue means that the meeting process will become established more quickly. Resources that you will need for the meeting will also be available.

In addition, you should decide how and when your team will communicate with each other. Your organisational plan document should have explicit statements about:

- what must be communicated;
- when and how often;
- by whom;
- to whom; and
- by what means.

Team organisation – communication (157)

You can check the completeness of your document by asking questions about the contingencies that the team has made for when things go wrong. For example, you should checker whether your plan can answer the question:

“I can't make the next meeting. Who should I tell? How much advance notice? Email or note?”

If you can't answer the question, then you should think about extending your group organisation document to accomodate such situations.

You should develop your group organisation document as your discussions progress. The plan should also evolve over the life-time of the project, as your understanding of your team mates improves; and the *organisational culture* of your team emerges. Organisational culture can be described as ‘the way we do things around here’, and is often implicit in much of the work of an organisation. Explicitly documenting practices can help new members of the team to integrate more quickly with the work effort.

The structure of the plan should be as follows:

Structure of the group organisation document (158)

1. Roles  
Who does what?
2. Authority  
Who decides? How are decisions taken?
3. Communication  
Where and when will you meet? How will you communicate otherwise?
4. Information management  
Where is information kept? How and when will it distributed? Who can use it?
5. Organisational risks  
What are the threats to team success arising from the way you organise the group?

A `LATeX` template for a group organisation document is available on Moodle. Alternatively, you may consider maintaining your team organisation plan on a project wiki. Wikis are useful formats for documents that are expected to change over time, as they normally provide facilities for reviewing and reverting document changes. There is a tutorial on using the project management system Trac, which includes a wiki, in Section 5.4.

#### Tools for information management (159)

Other tools for information management and communication include:

- social media sites;
- email;
- instant messaging; and
- blogs.

Your plan should document what communication technologies you will use, but also *how* you will use them. This means you should describe any communication conventions that are not standardised in the technology. For example, a project may decree a format for email subject lines, such as [`<project acronym>:<work package>`]<Topic> to ease filing, archiving and conversation tracking.

## 5.2 Project Planning and Scheduling

Many software development process models are described as *plan* or more precisely *schedule driven*, meaning that project activities are conducted in accordance with a pre-prepared and agreed plan. Unfortunately, planning is often seen as a distraction from the main purpose of a software development project:

actually doing some software design and implementation that results in a working system. Why waste effort on drawing up a whole plan that is obsolete within the first month?

However, the value in undertaking planning is not primarily in the production of plans themselves (although these are useful for communication purposes). Rather, project planning can be an effective means of establishing a shared understanding of a project scope, goals and schedule amongst team members.

US President Dwight D. Eisenhower described this succinctly:

The value of planning  
(160)

“ I have always found that plans are useless but planning is indispensable.”

– Dwight D. Eisenhower

The amount of documentation produced in association with this process will vary from project to project. In particular, it is often useful to produce project documentation in large or long lived projects, where it is important to communicate project plans between different groups at different points in time. Crucially, any documentation that is produced must have value as a tool of communication. Unread documentation is worthless, regardless of the care taken to produce it.

### 5.2.1 Documenting Project Plans

A plan driven project is managed according to a *schedule*, which is an estimate of when the different activities within the project will occur and be completed. A schedule consists of the following elements.

Terminology in  
planning (161)

- *Tasks* describe a unit of activity to be undertaken by one or more members of the project team. The granularity of tasks will vary from project to project. A well defined task should have a clear outcome, such as one or more products, so that it is possible to measure when the task has been completed.
- *Work Packages* are collections of related tasks. Typically, a sub-set of team members will share the work in a same package of tasks.
- *Dependencies* are the tasks in a project schedule that must be completed before another task (the dependent) can be begun. Many tasks necessarily have dependencies (integrating requirements into a specification must be preceded by a period of requirements gathering, for example. However, where possible, dependencies should be minimised wherever possible. Minimising dependencies between tasks enhances flexibility should delays be introduced into a project, because project team members can be reassigned to other tasks.

- *Milestones* represent the end points for key periods of activity in a project, such as requirements specification or the transition of an organisation to a new version of a software product. Milestones can typically be identified as ‘pinch points’ in a project schedule, where a number of tasks are all dependent on the completion of previous tasks. Typically a milestone is associated with a number of deliverables.
- *Deliverables* are project artifacts that are distributed outside of the project team organisation, typically when a milestone is reached. Deliverables are the products by which a customer measures progress on a software project. Project deliverables may include requirements specifications, software releases, acceptance test reports, defect reports, prototypes or user manuals.
- *Deadlines* are the dates agreed with a customer for the delivery of key products. Software engineering processes are often described as either *feature driven* or *deadline driven*. In both cases, a plan is developed to deliver a software project within a set schedule. If delays are experienced in a feature driven process, then the schedule is allowed to *slip*; but in a deadline driven process, features are dropped from the current system release to ensure the deadline is met.

Deadline and feature driven processes (162)

Work packages and milestones (163)  
Describing a task (164)

Figure 5.1 summarises the tasks and milestones in two work packages for the branch library management system. The tasks are from two work packages, 2. *Requirements Gathering* and 3. *Requirements Documentation*.

The descriptions of the tasks need to be elaborated further. Each task description should have:

- a *unique identifier*, which also denotes which work package the task belongs to;
- a short *label* longer *description*, explaining what activities must be undertaken in the task;
- a list of concrete task *outcomes*, which can be used to determine when the task has been completed;
- a list of *deliverables*, describing which of the products of the task will be released outside of the project, e.g. to the customer;
- task *dependencies*, indicating what tasks must be completed before this task can begin;
- an estimate of the time needed to complete the task (the *duration*);
- the task *lead* and *sub-team* drawn from the larger project team; and finally
- any *risks* that may affect the successful completion of the task. Risk management is described in Chapter 6.

<b>Project Milestones</b>	
M1	Initial Requirements Specification
M2	Intermediate Requirements Specification
M3	Final Requirements Specification

<b>WP2: Requirements Gathering</b>	
T2.1	Prepare for Interview with Central Librarian
T2.2	Prepare for Interview with Head of School
T2.3	Conduct Interview with Central Librarian
T2.4	Conduct Interview with Head of School
T2.5	Prepare for User Panel
T2.6	Conduct User Panel
T2.7	Build Prototype for Requirements Validation
T2.8	Conduct Prototype Demonstration
T2.9	Build Prototype for Central Library API

<b>WP3: Requirements Documentation</b>	
T3.1	Develop Initial Use Cases and Domain Model
T3.2	Integrate Requirements from Central Librarian
T3.3	Integrate Requirements from Head of School
T3.4	Conduct Quality Assurance Review of Requirements Specification
T3.5	Integrate Requirements from User Panel
T3.6	Integrate Requirements from Prototyping Demonstration Exercise
T3.7	Integrate Non-functional Requirements from API Prototype Exercise
T3.8	Conduct Quality Assurance Review of Requirements Specification

Figure 5.1: work packages and tasks describing requirements engineering activities in the branch library management system project

An example task  
(165)

Figure 5.2 documents a selection of project tasks for the branch library management system in more detail. Notice that several of the fields in the task description have not been completed at this stage. This is because we have not yet established this information.

Example estimation  
and allocation (166)

Figure 5.3 illustrates an example task duration estimation and resource allocation for the branch library management system. This information should also be presented on each individual task description. Notice that each task is associated with a sub-team of members drawn from the overall project team. Where there is more than one member of the team, one team member is allocated the role of task *lead*, indicated in bold in the figure.

Time units in resource  
estimation and  
scheduling (167)

Notice that the task duration is reported in hours and days in the figure. It is usually possible to *estimate* the duration of a task in hours. For example, the interview with the Head of School and central librarian are estimated to take an hour (the length of the appointment that has been made with the Head of School to conduct the interview).

Making task duration  
estimates (168)

However, *scheduling* using more precise time units (hours for example) is impractical and would result in micro-managing of a team member's time. In the example above, it doesn't matter exactly when the interview takes place in a given day in terms of the overall project schedule, so long as the interview does take place. Conducting the interview at 5pm instead of 9am will not result in a measurable project slippage of eight hours.

To manage this situation, it is usually most convenient to estimate in hours and then convert this estimate into working days by dividing by eight and rounding up. For example, Task T2.9 has a duration of 39 hours, or  $39/8 = 4.875 = 5$  working days.

Factors to consider when making task duration estimates include:

- the number of team members to be assigned to the task;
- the rationale for assigning a team member to a task;
- the ability and experience of the team member(s) assigned the task.

<b>T3.1:</b>	Develop initial use case and domain model
<b>Description:</b>	Document an initial use case diagram and domain model for the branch library management system, based on requirements and domain elements identified from the project outline and scope. Any uncertainties regarding the functions or scope of the system should also be documented.
<b>Product:</b>	3.1 (Requirements Specification) Release (Internal) 0.1
<b>Deliverable:</b>	None
<b>Risks:</b>	
<b>T2.1:</b>	Prepare interview plan for central librarian
<b>Description:</b>	Develop an interview script of questions addressing current uncertainties in the requirements specification. Questions should be focused on uncertainties regarding the current use cases and domain model, with priority given to uncertainties which reflect high risk for the branch library management system. Roles during the interview must also be assigned, specifically interviewer(s), note taker(s).
<b>Product:</b>	Interview script
<b>Deliverable:</b>	None
<b>Risks:</b>	
<b>T2.2:</b>	Conduct interview with central librarian
<b>Description:</b>	Question central librarian using previously prepared script and document answers. Interview may deviate from script if significant previously unknown requirements are elicited and need to be elaborated.
<b>Product:</b>	Interview report
<b>Deliverable:</b>	Central librarian may not know answers to priority questions, or who to ask. No backup identified for interviewer
<b>Risks:</b>	
<b>T3.2:</b>	Integrate central librarian interview report into requirements specification.
<b>Description:</b>	The requirements specification document must be altered to reflect new understanding of system scope and requirements following interview. This will involve extension and changes to the domain model class diagram, use case diagram(s) and use case description(s) for the branch library system.
<b>Product:</b>	3 (Requirements Specification) Release (Internal) 0.2
<b>Deliverable:</b>	None
<b>Risks:</b>	
<b>T3.4:</b>	Quality assure draft initial requirements specification
<b>Description:</b>	The requirements specification document must be reviewed by the quality assurance manager to ensure the quality and consistency of documentation, prior to release to customer.
<b>Product:</b>	3 (Requirements Specification) Release (Interim) 1.0
<b>Deliverable:</b>	3 (Requirements Specification) Release (Interim) 1.0
<b>Risks:</b>	None

Figure 5.2: example tasks from the branch library management system project

WP2: Requirements Gathering				
task	title	hours	days	team members
T2.1	Prepare for Interview with Central Librarian	2	1	Cooper
T2.2	Prepare for Interview with Head of School	4	1	Hoffstadter
T2.3	Conduct Interview with Central Librarian	1	1	Cooper, Hoffstadter,
T2.4	Conduct Interview with Head of School	1	1	Hoffstadter, Cooper
T2.5	Prepare for User Panel	10	2	Cooper
T2.6	Conduct User Panel	1	1	Cooper, Hoffstadter
T2.7	Build Prototype for Requirements Validation	35	5	Kuthrapali
T2.8	Conduct Prototype Demonstration	1	1	Kuthrapali, Hoffstadter
T2.9	Build Prototype for Central Library API	39	5	Kuthrapali, Cooper

Figure 5.3: example task estimation and allocation for the branch library system

Problems in task duration estimation (169)

Making accurate estimations of project task duration can be very challenging. The accuracy of an estimate will depend on:

- the *intrinsic* complexity of the task;
- the *political priority* of the task and any associated requirements;
- the availability of clear outcomes for the task, by which progress can be measured;
- the availability of previous similar tasks on which to base estimates; and
- the *experience* of the team member(s) making the estimate of previous similar tasks.

Where possible it is usually better to partition complex tasks into their constituent tasks and clearly identify the resulting dependencies between the component tasks. This is because it is easier to make accurate estimates of durations for smaller, simpler tasks. Compare, for example, the difficulty of producing a reliable estimate for building the infrastructure for a major sporting event, such as the Olympics, compared with estimating how long it will take to prepare the foundations for a specific building.

Unfortunately, it isn't always possible to take this approach, because of the inter-dependencies between the components of a partitioned task. For example, the task of writing a paper for a coursework assignment *could* be broken down into:

Essentially complex tasks – an example (170)

1. reviewing the literature;
2. formulating a central narrative or argument;
3. setting up the structure and outline (main section titles);
4. adding the meta-data (author, title, date and so on);
5. writing each of the body sections;
6. writing an introduction and a conclusion; and
7. writing the abstract.

These tasks could then be allotted to each member of a team of authors. However, this would result in a very patchy and poorly written paper because:

- it would likely lack a consistent style as each of the authors wrote in their own way, using their own conventions, referencing formats and grammatical styles;
- many of the sections would show a poor understanding of the relevant literature, since not all team members were involved in background reading; and
- there would be no opportunity to improve the paper as each team member formed a better understanding of the background literature.

This is of course an extreme example, as many of the problems can be mitigated by adding extra tasks (quality assurance reviews, for example) and intermediate deliverables (early drafts, for example). However, these additional tasks impose their own overhead on the project, because of the complex inter-dependencies between the component tasks.

The influence of task complexity on the accuracy of a schedule has been discussed extensively by Brooks [1995]. Brooks worked on the IBM 360 operating system project in the early 1970's. The book records his reflections on this and other experiences as an engineer.

A central argument of the book is that the notion of a *man month* is a dangerous myth, which doesn't reflect the complexities of many software development based tasks.<sup>1</sup> The myth is dangerous (Brooks argues, because it encourages the idea that project schedule slippage can be remedied by adding additional team members. Brooks argues that instead, adding additional team members will cause an already late project to be even later.

Figure 5.4 illustrates this argument (redrawn from Brooks [1995]). The figure shows the effect of adding further team members to different types of software project for the project schedule (measured in months).

The Mythical Man Month (171)

The effect of changing team size in complex projects (172)

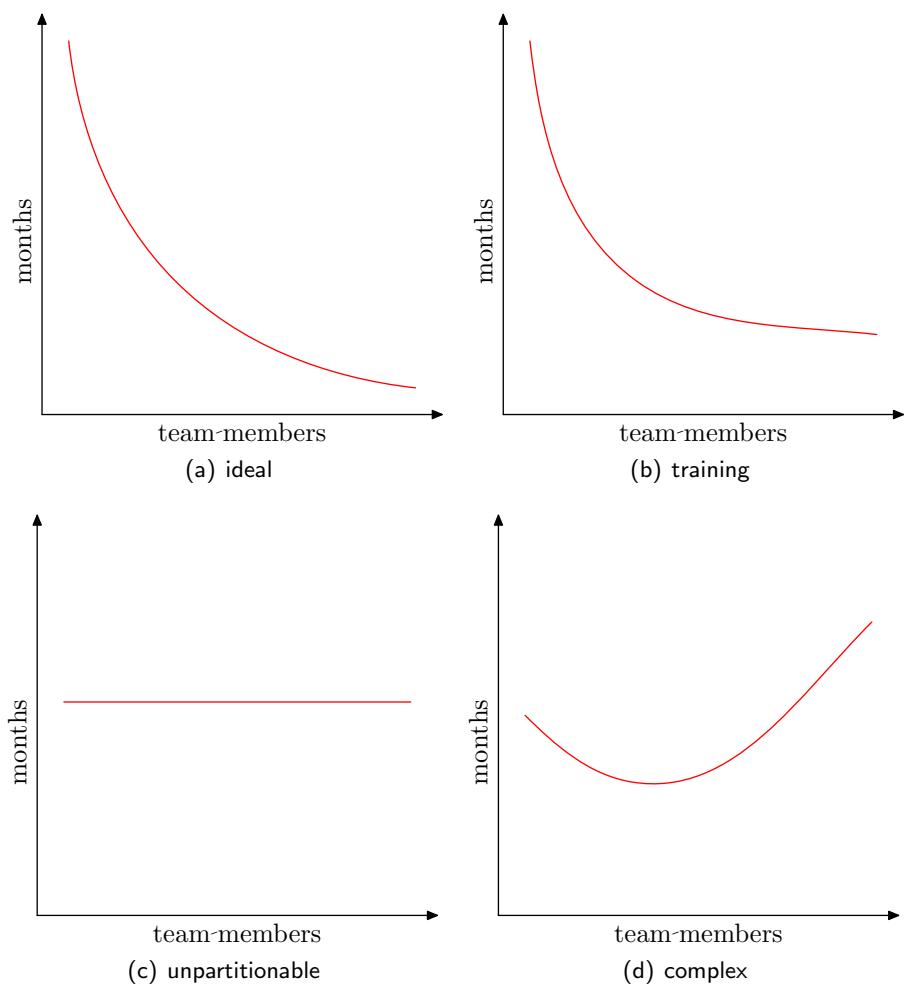


Figure 5.4: the effect of adding additional team members to a late project  
(redrawn from Brooks [1995])

Figure 5.4(a) illustrates the ideal case, in which all tasks are infinitely partitionable. Adding further programmers means that any task can be further sub-divided and undertaken in parallel with existing efforts.

Figure 5.4(b) illustrates the more pragmatic case, where the pool of additional programmers must undergo training, in the development environment, programming language, project conventions or details of the problem domain. This imposes a constant cost (in terms of person months) on the project that cannot be remedied by adding further programmers.

Figure 5.4(c) illustrates the effect of adding new team members to a project that cannot be partitioned into further parallel sub tasks. The duration of the project cannot be lessened, no matter how many new team members are added, because the dependencies between the sub-tasks means they must be completed sequentially.

An alternative approach, when it is not possible to partition a task, is to record how *confident* the estimator was in a particular task duration estimate. If an estimate is associated with a low confidence, this uncertainty can be recorded as a risk in the project's risk management plan. One way of mitigating this risk is ensuring that low confidence task duration estimates have plenty of *slack time* in the project schedule (see Section 5.2.4 below).

We can see from the preceding discussion that many projects may experience over-run not because team members are taking too long, but because the original estimates were unrealistic in the first place. Unfortunately, this is known to be a problem, even when uncertainties associated with estimation are taken into account! *Hofstadter's Law* puts this very nicely:

*"It always takes longer than you expect, even when you take into account Hofstadter's Law."*

Confidence for task duration estimates (173)

Hofstadter's Law (174)

In the end, we have to accept that project schedules are estimates, and that these estimates will need to be revised as a project progresses. Consequently, project planning and scheduling continues throughout the project lifecycle, as progress is made on implementation and new the scope of the project becomes better understood.

### 5.2.3 Visualising Project Schedules

There are two established ways of visualising a project plan:

Gantt and PERT charts (175)

- Gantt charts, which display the schedule of tasks with respect to a calendar; and
- PERT charts which are used to display the dependencies between tasks within a project.

---

<sup>1</sup>(The book was first published in the 1970s after all, and a mythical person-month doesn't have quite the same evocative visual imagery.)

Example Gantt chart  
(176)

Benefits of Gantt  
charts (177)

Example PERT chart  
(178)

Dummy tasks on  
project PERT charts  
(179)

Figure 5.5 illustrates a Gantt chart for the early stages of the branch library management system. The timeline for the project is displayed horizontally. The time units depend on the scope and scale of the project. In the figure, the units used are working days.

Project tasks are listed down the left hand side of the chart. The time each task is active is displayed as a bar on the chart. Milestones are represented as diamonds, and do not have a duration. Dependencies between tasks are illustrated as arrows drawn between task bars.

Gantt charts are useful for visualising:

- what activities will take place when;
- over or underloaded staff at particular points in time;
- repeated or intermittent tasks; and
- task slack time.

Although it is possible to visualise dependencies, these can be harder to manage on a Gantt chart, and can clutter the presentation of timings.

Figure 5.6 illustrates a Program Evaluation and Review Technique (PERT) chart for the same tasks as Figure 5.5. Milestones are represented as nodes. Tasks are represented as arcs between milestones. Each arc is labelled with the duration of the task. Incoming activities to a milestone are the dependencies of any out-going activities. For example, Task 3.4 is dependent on tasks T3.2 and T3.3. Finally, each task duration is shown in parentheses next to the task arc label.

Notice in the figure that not all milestones have explicit labels on a PERT chart. Any confluence of tasks is considered a milestone for the project. This flexibility can be exploited to ensure that a project schedule is not held up by unnecessary dependencies to labelled milestones. Figure 5.7 illustrates how 'dummy' tasks can be introduced into a schedule.

Figure 5.7(a) illustrates the initial structure for the project schedule. The figure shows a milestone, A, with three dependencies (T1, T2 and T3) and two dependents (T4 and T5).

However, it is discovered that T4 is only dependent on T1 and T2, and that T5 is dependent on T1, T2 and T3. This is unfortunate because in the current schedule, T5 cannot begin until T3 ends even though it isn't a dependency for T4.

To remedy this problem, we can introduce a new milestone and a dummy task (T6 with duration 0) as shown in Figure 5.7(b) to separate out the relationship between T4 and T5. The schedule now shows the 'real' dependencies for the project, with T4 only dependent on T1 and T2.

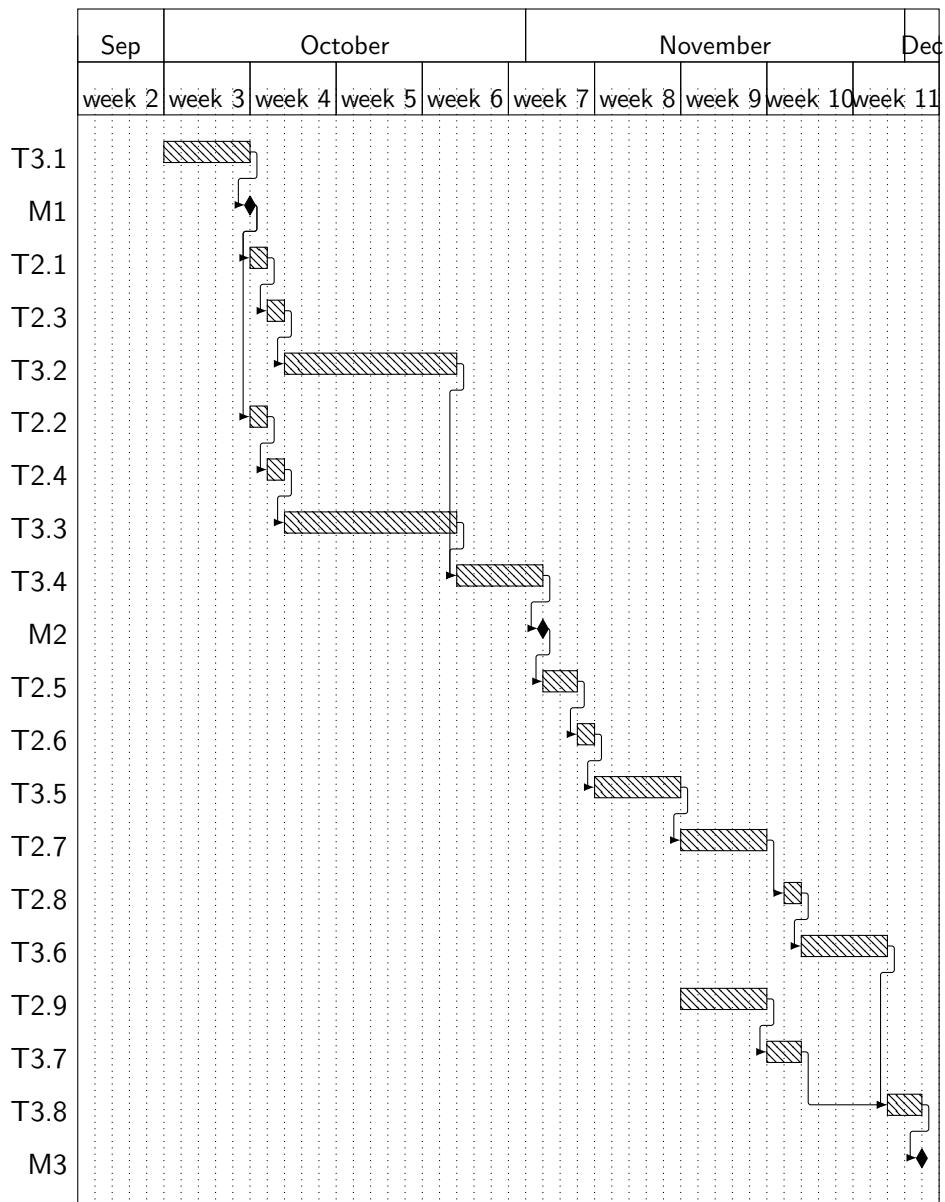


Figure 5.5: a Gantt chart for the Branch Library Management System (requirements elaboration phase)

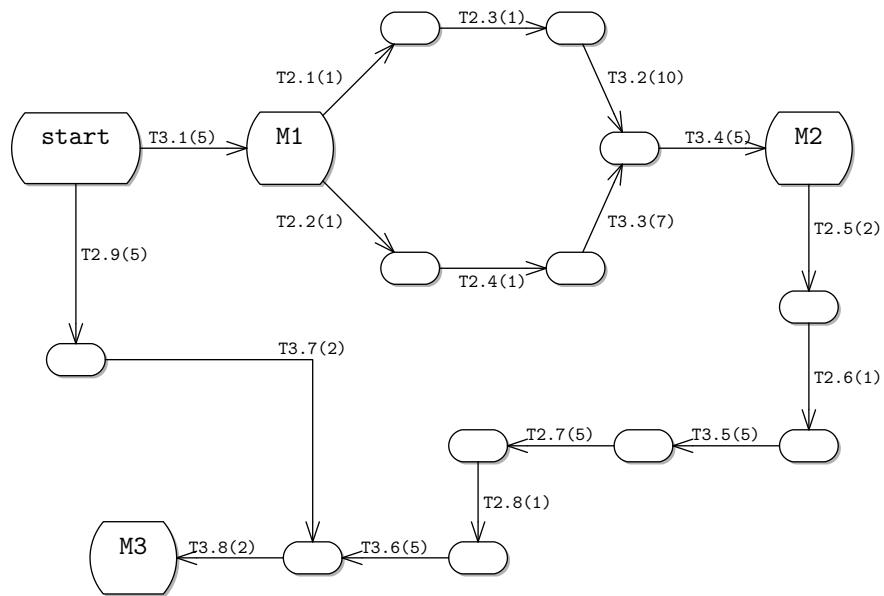


Figure 5.6: example PERT chart for the branch library management system project

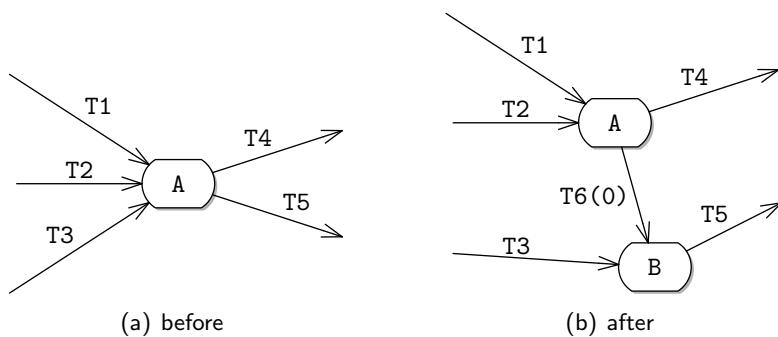


Figure 5.7: introduction of dummy tasks into a project schedule

### 5.2.4 Calculating Critical Paths in a Schedule

PERT charts complement the advantages of Gantt charts, and are particularly useful for calculating and visualising the the critical path(s) through a project. There are three steps involved in calculating a critical path:

1. A forward pass calculating the *earliest time* ( $e$ ) that each milestone could be reached. The total length of each path through the task network to the milestone from the start is added up. The longest path is then selected as the earliest time that all dependent tasks for a project will be reached.
2. A backward pass calculating the latest time ( $l$ ) that a milestone can be reached without delaying the overall project. Starting at the final milestone, the duration of each out-going task from a milestone is subtracted from the destination milestone's  $l$  value. If there are two or more possible values for  $l$  (because there are two or more out-going paths), then the least value is chosen.
3. The slack time is for each milestone is calculated as  $l - e$ . Any milestones that have a slack time of 0 are on the critical path for a project, because any delay in reaching the milestone will cause a delay in the overall project.

Figures 5.8 illustrates the calculation of the critical path for the branch library management system project's requirements gathering and documentation tasks. The values of  $e$  and  $l$  are recorded for each milestone as <1>/<e>.

First, the earliest time that each milestone can be reached is calculated, as show in Figure 5.8(a). During the forward pass, the earliest time that a milestone can be reached is the shortest path to that milestone from the start of the chart. So, the earliest time to reach the start milestone is 0 and to reach milestone M1 is  $0 + 5 = 5$ . The earliest time to reach the milestone just before M2 needs to be selected from the two possible paths to the milestone, either:

$$0 + 5 + 1 + 1 + 10 = 22 \text{ for the upper path; or}$$

$$0 + 5 + 1 + 1 + 7 = 17 \text{ for the lower path.}$$

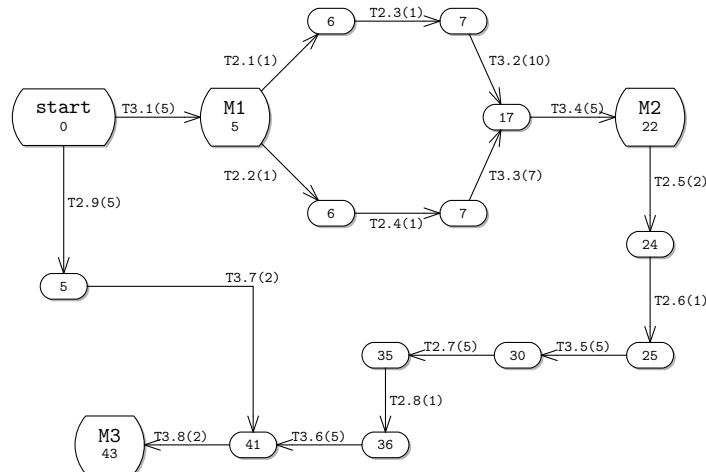
This forward pass continues for each milestone until the end (M3) is reached with a minimum time of 43.

Next, a backward pass is used to calculate the latest time that a milestone can be reached without delaying the project (Figure 5.8(b)). The process begins with the last milestone. For M3, the latest time is  $43 - 0 = 43$ , because it has no out-going branches. For the milestone just before M3, the latest time it can be reached without delaying the finish is  $43 - 2 = 41$  and for the milestone just to the right it is  $41 - 5 = 36$ .

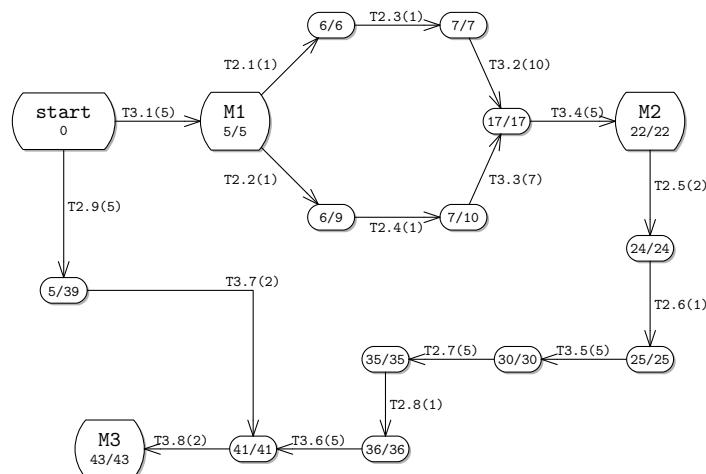
Milestone M1 has two out-going branches from which the backward pas must be computed:

Calculating Critical Paths in a Schedule (180)

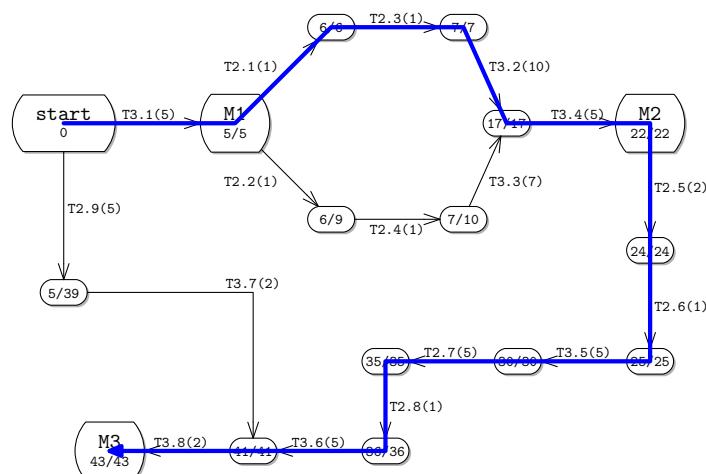
Example critical path calculation (181)



(a) forward pass



(b) backward passs



(c) critical path

Figure 5.8: using a PERT chart to calculate the critical path through a project's activities

$6 - 1 = 5$  for the upper path; or

$9 - 1 = 8$  for the lower path.

So, the upper path's  $l$  value is chosen, as it is the smaller of the two.

Finally, Figure 5.8(c) shows the critical path for this part of the project. Each milestone on the critical path has a slack time of 0 (i.e. they have equal  $e$  and  $l$  values).

Notice that the critical path calculation is only valid at the start of the project (i.e. when project time is 0), when no tasks have been completed and no delays experienced. This means that the critical path for a project needs to be continually revised as the project progresses, delays experienced and estimates are revised.

There are two further metrics that can be calculated for each task on a PERT chart. For a given task represented as an arc from milestone  $i$  to  $j$ , where  $d_{i,j}$  denotes the duration of the task arc between milestones:

- *Total float ( $tf$ )* is the amount of time a given activity can be delayed beyond its earliest start time, without causing the whole project schedule to slip.

$$tf_{i,j} = l_j - e_i - d_{i,j}$$

Total float and free float for an activity  
(182)

Total float is calculated as the *latest* time the destination milestone can be reached (without reducing delay), minus the earliest time that the source milestone can be reached, minus the expected duration of the task.

- *Free float ( $ff$ )* is the amount of time that a task can be delayed without affecting the earliest start time of any other task.

$$ff_{i,j} = e_j - e_i - d_{i,j}$$

Free float is calculated as the *earliest* time the destination milestone can be reached (without reducing delay), minus the earliest time that the source milestone can be reached, minus the expected duration of the task.

Total float calculates how much slippage a task can accommodate without causing the overall project to slip. Free float calculates how much slippage a task can accommodate without absorbing some of the slack time of other tasks.

## Summary

This chapter has introduced the principles and practices of project organisation and scheduling in software development. The key point is that project management is primarily concerned with ensuring good communication between

different team members. This communication has a variety of purposes, from ensuring that managers are aware of risks and delays, to integrating new team members into the project culture. Project scheduling is part of this communication. The act of project scheduling establishes an understanding of when different tasks are expected to be completed.

### 5.3 Exercises

1. Consider the project plan, given as a dependency table in Figure 5.9

task	dependencies	duration
T1	-	3
T2	-	7
T3	-	2
T4	T1	4
T5	T3	12
T6	T3	4
T7	T2, T4, T5	8
T8	T6	10

Figure 5.9: An example project plan

- (a) Construct a PERT chart for the project.  
(b) Calculate the slack time for the project milestones and the critical path(s).
2. Take another look at Figure 5.7. How would you re-organise the schedule if it turned out that T5 was only dependent on tasks T1 and T3?

## 5.4 Workshop: Using Trac for Project Planning

This workshop goes through the basic steps of configuring and using Trac<sup>2</sup>. Trac is a configurable web based issue tracking and management tool which integrates with subversion. Trac organises issue tracking around *tickets* that describe features to be implemented, bugs to be fixed or tasks to be completed. The Trac system also integrates with the Subversion version control system, so we will also set up a version control repository in anticipation of the lectures on change management. You can find out more about Trac from the online documentation wiki available on the project website.

**Important** the instructions for this workshop will work, provided you follow them exactly. Be careful to type in the commands exactly as they are given.

The workshop will assume that you are a student with username 1234567c and are a member of a Team labelled blue. This means that you will belong to a unix group wwwblue on the Trac server, hoved. You will need to substitute your username and team label, respectively, in the instructions below.

### 5.4.1 Logging into the Trac Server

Trac is hosted on the server hoved.dcs.gla.ac.uk. You can find your group's Trac site by visiting the URL:

<http://hoved.dcs.gla.ac.uk/blue/trac>

Obviously, you haven't set Trac up just yet, so it won't work right away! The first thing you need to do is log in to hoved using your active directory credentials using a command prompt.

If you are using a Linux distribution (e.g. in the Level 3 laboratory), open a terminal application and type:

```
]# ssh -l 1234567c hoved.dcs.gla.ac.uk  
1234567c@hoved's password:  
Last login: Wed Oct 20 16:57:51 2010 from misima  
Dept. of Computing Science hoved  
]#
```

If you are using a Windows machine (e.g. in the Level M laboratory) then you should start the putty application from the start menu button **Start Menu**→**putty**→**putty**. Then, enter the host name to connect to in the dialogue as hoved.dcs.gla.ac.uk and click **Ok**. A dialogue box will open asking whether you want to accept the remote machine's credentials, so click **Ok**. A command prompt will start and ask for your username and password. If you enter these correctly, you should be logged into the Trac server.

The ]# symbol means a command prompt in the notes. Output from the terminal is indicated on lines without a command prompt.

---

<sup>2</sup><http://trac.edgewall.org>

### 5.4.2 Setting up the Site

Change to the directory that has been created to store your team's Trac (you can use the tab key to auto-complete command and file path names):

```
]# cd /extra/2012/blue/
```

Now create a subversion repository to store your deliverables in.

```
]# svnadmin create repos
```

This creates a repository at path location /extra/2012/blue/repos. You can examine the repository directory listing by typing:

```
]# ls repos  
conf db format hooks locks README.txt
```

Your subversion repository will also need to be readable by other for the Apache web server to access it (this is the default). To make the repository writeable, you need to modify the group ownership and permissions of the repos directory.

```
]# chgrp -R wwwblue repos  
]# chmod -R g+rw repos
```

We'll explore using Subversion for change management in Section 8.5.

Now create a Trac repository in the directory by typing the command:

```
]# mkdir /extra/2012/blue/trac  
]# cd /extra/2012/blue/trac  
]# trac-admin ./ initenv
```

and then providing answers to the prompted questions:

- Project Name → Team Blue
- Database connection string → sqlite:db/trac.db
- Repository **type** → svn
- Path to repository → /extra/2012/blue/repos

The admin tool will then create the basic layout of the Trac in the current directory.

The next step is to configure access to the site for the web server on hoved. To do this, you need to make the database directory in your Trac accessible to the Apache daemon:

```
]# chgrp -R wwwblue db  
]# chmod -R g+rw db
```

Also, you need to create user accounts for each of the members of the team so that they can login. The prompt will ask you to enter a username and password.

```
[# cd /extra/2012/blue/
[# htpasswd -c team.htpasswd 1234567c
New password:
Re-type new password:
Adding password for user 1234567c
```

You only need the `-c` option the first time you create an account. You may need to set the password file to be readable by your group, if you each create your passwords from your own accounts. To do this, type:

```
[# chmod g+rw team.htpasswd
```

in your team's trac directory.

Try visiting the URL for your Trac site again in a browser. If all went well, you should see the front page of the Trac website, similar to Figure 5.10.

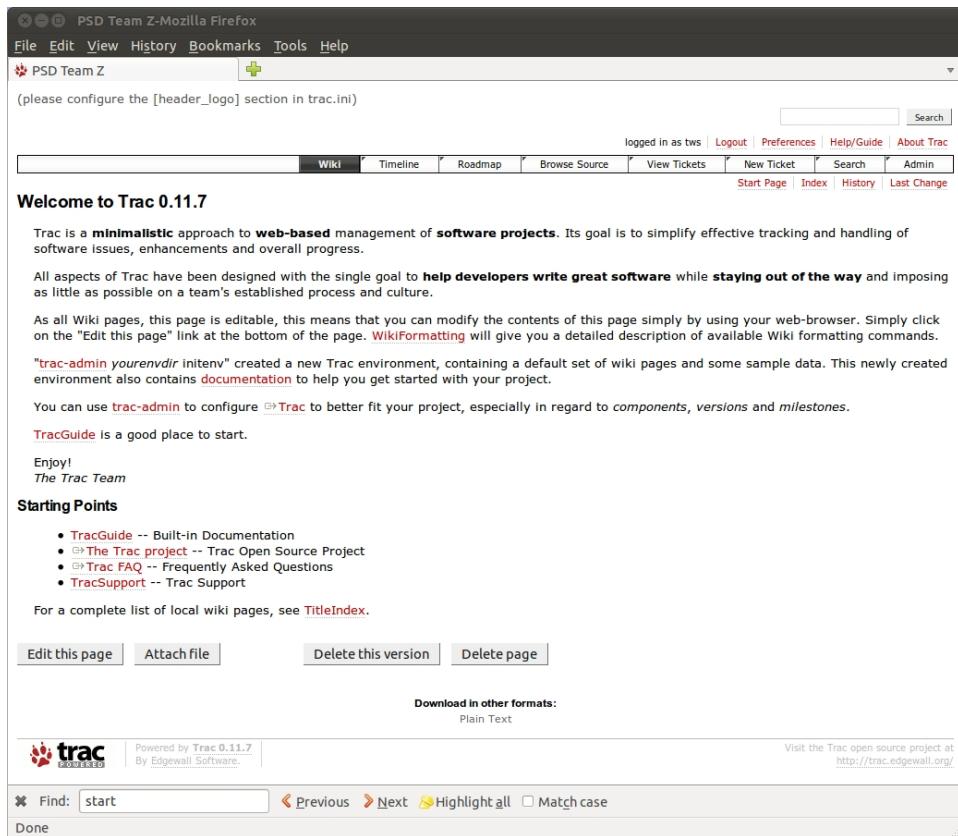


Figure 5.10: the front page of Trac

Finally, we need to set up Trac so that it can be administered from the web interface, rather than the command line client. Returning to the command prompt, start the Trac administration command line client `trac-admin` in the Trac directory:

```
[# cd /extra/2012/blue/trac
```

```
]# trac-admin ./  
Trac ]#
```

Whenever you are using the trac-admin client you can get help with the available commands by typing `help`, producing a list commands and the syntax of their arguments.

To view the current permissions for a user, type:

```
Trac #] permission list  
User   Action  
-----  
tws    BROWSER_VIEW  
tws    CHANGESSET_VIEW  
tws    CONFIG_VIEW  
tws    EMAIL_VIEW  
tws    FILE_VIEW  
[...snip...]  
tws    WIKI_VIEW  
  
Available actions:  
BROWSER_VIEW, CHANGESSET_VIEW,[snip...]
```

To set administrative privileges for a user, type:

```
Trac #] permission add 1234567c TRAC_ADMIN
```

This gives the specified user full administrative privileges in the Trac system - so use them carefully!

You can log out of the Trac admin console by typing:

```
Trac #] exit
```

#### 5.4.3 Creating and Viewing Tickets

You can explore the various menu options on the web interface at this stage, including:

- the **Wiki** for maintaining documentation;
- the **Timeline** which reports project events in Trac;
- the **Roadmap** which summarises tickets by milestone;
- a link to **Browse Source**, which presents a view of your subversion repository;
- a collection of customisable queries for **View Tickets**;
- a form for creating a **New Ticket**;
- a general **Search** feature for all of the content in Trac; and

- an **Admin** screen for configuring milestones, components, priorities and other features of Trac.

Try creating a ticket in Trac (Click on **New Ticket**). Tickets are descriptions of small packages of work to be completed by the team. Tickets can be:

- associated with project milestones;
- associated with particular versions and/or components of the project;
- given a type (defect, task etc.);
- given file attachments;
- assigned to team members for completion;
- prioritised, updated with further information (e.g. the cause of a defect); and
- marked as completed.

The ticket can be altered at any time by clicking on **View Tickets**→**Active Tickets** and selecting the ticket to edit. Each time the ticket is edited, the change is recorded in the ticket's history.

#### 5.4.4 Customising Trac to Fit your Project

Once you are comfortable with the features of tickets, you need to customise the features of the ticket creation form to fit your project.

First, start setting up project milestones to match those of your project . To do this, click on the **Roadmap** button on the top right of the window. This lists the current milestones set for the project, labelled milestone1, milestone2, milestone3 and milestone4: not very interesting!

Click on the first default milestone, **milestone1**, and then choose the **Delete milestone** button. A dialogue box will ask if you wish to re-assign tickets associated with milestone1 to another milestone. This is convenient if you have already added tickets, but since track is currently empty, you can just go ahead and click **Delete milestone** to confirm. Repeat this process for the other milestones listed in the Roadmap.

Next, create a new milestone by clicking on the **Add new milestone** button in the **Roadmap** window. A web form should appear, similar to that shown in Figure 5.11. Fill in the web-form for one of your deliverables, with a due date of 29<sup>th</sup> Nov. You can also add a description to the milestone if you wish. Click **Add milestone** when you are ready to confirm. You can view the newly added milestone by clicking on the **Roadmap** button as before.

Returning to the Trac website, you can open one of the tickets you created earlier (e.g. by browsing through **View Tickets** → **Active Tickets**) and change

New Milestone – PSD Team Z +

## New Milestone

Name of the milestone:  
D4 PSD Group Exercise 1 Final Hand in

Schedule

Due:  
12/01/11 Format: MM/DD/YY

Completed:  
09/27/11 18:27:37 Format: MM/DD/YY hh:mm:ss

Description (you may use [WikiFormatting](#) here):  
B I A The final hand in combines all three previous hand ins.

[Add milestone](#) [Cancel](#)

Figure 5.11: adding a milestone to the Trac

the ticket's milestone to the deliverable milestone you just created. If you now browse to the **Roadmap** menu option, you can see that the hand in milestone now has a progress tracker and the due date we just set. Try creating a few more tickets for the milestone and setting some of them as complete. You can use the **Roadmap** presentation as a progress tracker for your project.

Try configuring other aspects of Trac from the **Admin** window (selected from the Trac window menu). For example, you could change the prioritisation scheme for tickets to fit with the MoSCoW approach illustrated in Table 11.4 of the notes. To do this, choose **Admin→Priorities**. Click the check box next to each priority level listed and then choose **Remove selected items**. Next add each of the four priority types from the Moscow Rules, so that you should see a screen similar to Figure 5.12.

You should also alter the list of **components** in the project to fit with the different groups of functions you have identified in your requirements specification as the project proceeds. Once you are satisfied with the configuration, add tickets to your Trac for the tasks you identified in your project plan.

Name	Default	Order
MustHave	<input type="radio"/>	1
CouldHave	<input type="radio"/>	2
ShouldHave	<input type="radio"/>	3
WouldBeNiceToHave	<input type="radio"/>	4

Add Priority  
Name: \_\_\_\_\_  
Add

Figure 5.12: configuring Trac with the MoSCoW rules.

#### 5.4.5 Extending Trac

This section describes some optional extra activities that you may try with your Trac deployment. Both extensions require you to edit the file `/extra/2012/blue/trac/conf/trac.ini` using a command line editor such as `vi` or `nano`. The file contains configuration details for Trac. Every configuration option is a name/value pair of the form `name=value`. Configuration options are grouped into sections with headers labelled `[like this]`.

##### Enabling Plugins for Gantt and Dependency Charts

Several plugins have been installed on the Trac server to support the generation of Gantt and dependency charts.

- JsGantt<sup>3</sup> for Gantt charts
- MasterTickets<sup>4</sup> for dependency charts

To enable these plugins, edit the configuration file, so that it contains the following lines:

---

<sup>3</sup><http://trac-hacks.org/wiki/TracJsGanttPlugin>

<sup>4</sup><http://trac-hacks.org/wiki/MasterTicketsPlugin>

```

[components]
tracjsgantt.* = enabled
mastertickets.* = enabled

[ticket-custom]
finish = text
finish.label = Expected finish (YYYY-MM-DD)?
finish.order = 5
estimate.label = Estimated hours?
estimate = integer
blocking = text
blocking.label = Blocking
blockedby = text
blockedby.label = Blocked By

[trac-jsgantt]
date_format = %Y-%m-%d
fields.finish = finish
fields.estimate = estimate

[mastertickets]
dot_path=/usr/bin/dot

```

Now access one of the tickets you have created. The ticket form should have extra fields for specifying the:

- finish date (note the awkward format);
- the estimated duration (in hours, not counting weekends and assuming an eight hour day);
- a comma separated list of ticket ids blocked by this ticket; and
- a comma separated list of tickets this ticket is blocked by.

Try creating (or altering) new ticket with some of these features filled in. You can then view the dependency graph between tasks by clicking on the **Depgraph** button in the top right hand corner of the window.

You can also view a Gantt chart version of your tasks by creating a wiki page and embedding a Gantt chart widget. To do this click on the **Wiki** button in the menu and then click **Edit this page**. Scroll to the bottom of the editable content and type

```
[wiki:Gantt]
```

This creates an internal link to the wiki page 'Gantt' inside Trac. Click **Submit changes**, and you will be taken back to the newly edited front page.

Of course, the Gantt page doesn't exist yet, because we haven't created it. To do this, click on the **Gantt** link you have just created at the bottom of the start page and then choose **Create this page**. You can now add content to the page called Gantt. To insert a Gantt Chart of your tasks, insert the following text anywhere on the page:

```
[[TracJSGanttChart ()]]
```

and click **Submit changes**. If all goes well, you should have a Gantt chart displayed on the page, similar to Figure 5.13.

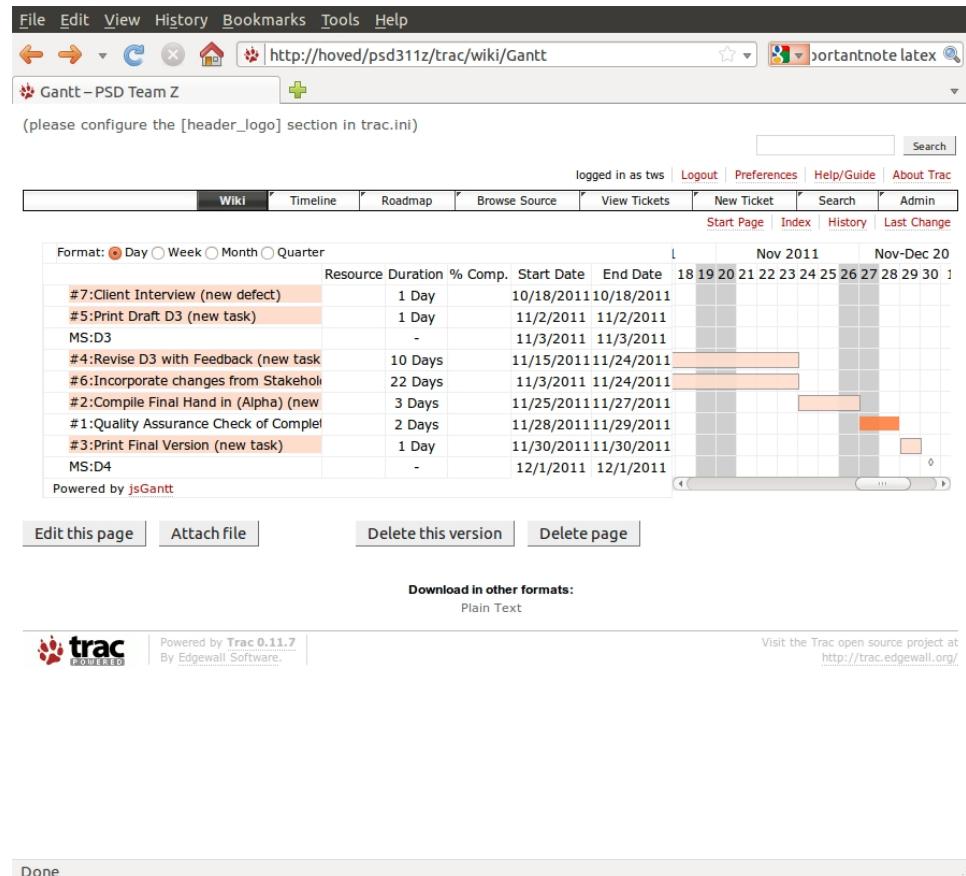


Figure 5.13: embedding a Gantt chart in Trac

## Apearance

There are a number of customisation options available in the file `conf/trac.ini`, including for example, an option for setting the project logo, logging options and configuring email notifications. It's a good idea to make a backup copy of this file before you make any changes.

You can set the logo that appears on the front page of your Trac site, by changing the `[header logo] src` field to the university's crest:

```
http://www.gla.ac.uk/0t4/generic/images/logo_print.gif
```

# Chapter 6

# Risk Management

## Recommended Reading

PLACE HOLDER FOR sommerville10software

Recommended reading  
(185)

PLACE HOLDER FOR boehm91software

PLACE HOLDER FOR cerpa09why

The concept of *risks*, potential future events that can adversely affect the outcome of a software development project was introduced in Chapter 3. Risk is therefore characterised by the probability of its occurrence, and by the impact of the risk on a project outcome should it manifest itself.

This is often expressed mathematically for adverse event  $e$ :

Defining project risk  
(186)

$$risk(e) \approx probability(e) \times impact(e)$$

Pay attention to the use of approximation, rather than equals, particularly when dealing with risks categorised as having catastrophic impact for a project. This chapter explores the concept of project risk further, and describes a process for planning for and managing project risks throughout a software development life-cycle.

## 6.1 The Risk Management Process

Like other project management activities such as planning and change control, the risk management process is orthogonal to the main software engineering activities, such as requirements gathering, design or testing. This means that the risk management process runs concurrently with engineering processes. Figure 6.1 illustrates this relationship.

The outputs from other software engineering activities are used to inform the risk management process. Constantly changing requirements may indicate that

Engineering processes  
and risk management  
(187)

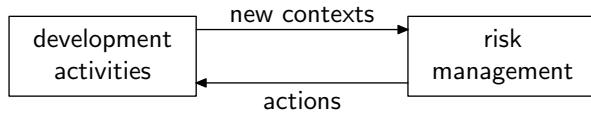


Figure 6.1: relationship between development and risk management processes

a risk of poorly understood problem domain is becoming manifest, for example. Similarly, the results of risk management activities influence how core software engineering processes are undertaken. There may be a decision, for example, to reduce the scope of a system design if a project is at risk of slipping behind schedule.

This arrangement may be made explicit in a process model. In the Rational Unified Process model, for example, risk management is part of the project management supporting process (see Section 4.5).

Figure 6.2, adapted from [Freimut et al., 2001], illustrates a generic risk management process. Risk management should begin from the outset of a software engineering project, when the initial scope is defined. An initial set of risk should be identified within this scope. These are then analysed, prioritised and documented. A risk management tool may be used for this part of the process. The most serious risks are then addressed through the application of controls (specific actions taken in the project). The effectiveness of the existing set of controls, and the project context are constantly monitored. If new risks emerge, more information about existing risks is obtained, or identified risks manifest themselves, further identification, analysis and control work is undertaken, respectively. The following sections detail each part of the process.

Risk management process overview  
(188)

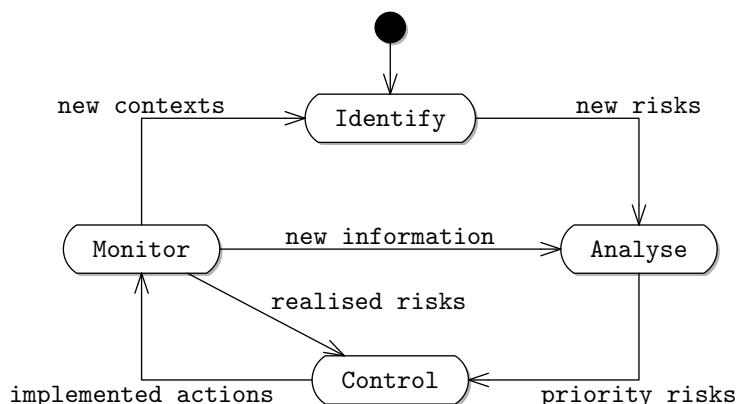


Figure 6.2: a risk management process, adapted from Freimut et al.

## 6.2 Identifying Risks

Ideally, risk management should be undertaken by a member of the team with experience of previous similar projects, but this isn't always possible. We look in Chapter 1 at how the nature of software means that many software engineering projects are genuinely unique. In some situations, *software metrics* gathered from previous projects can be used to inform the risk management process. We will look at software metrics in more detail in Chapter 9.

Boehm [1991] and Cerpa and Verner [2009] identified a number of high frequency causes of project failure as sources of potential risk for other projects:

- personnel shortfalls
- unrealistic schedules and budgets
- incorrect requirements
- wrong user interface
- gold plating
- requirements volatility
- shortfalls in external components
- shortfalls in externally performed tasks
- real-time performance shortfalls
- straining CS capability

Boehm's top 10 software risk categories (189)

Another approach is to identify risk by the different aspects of the project affected. Table 6.1 lists some categories of risks that need to be considered by a software project team.

Pressman [1997] proposed using a series of questions concerning the nature of the project as a checklist to elicit risks, for example:

- Number of users of the product?
- Costs associated with late delivery or a defective product?
- Number of other products with which this software must be interoperable?
- Have you worked with the customer in the past?
- Is the customer willing to spend time in reviews?
- Is a mechanism used for controlling changes to customer requirements?

Project risk categories (190)

Pressman [1997]'s risk elicitation questions (191)

<b>political</b>	can be either in external to the project (in the big) or internal (in the small)	re-organisation of the business, conflict between software development teams
<b>technical</b>	new technologies represent an unknown for a project team	re-implementation of a desktop package to a web-based online application
<b>skills</b>	there is often a tension within an organisation between training staff and ensuring that a project gets completed	developers wish to acquire experience in using AJAX for interactive web applications
<b>requirements</b>	mis-understanding of or lack of clear requirements for the system is a common cause of software project failure	the customer wants an online, but not necessarily web-based application

Table 6.1: types of project risk

- Is a procedure followed for tracking and reviewing performance of sub-contractors?
- Are software tools used to support the software analysis, design and testing processes?
- Is the technology built new to your organisation?
- Are the best people available?
- Do the people have the right combination of skills?

An extended version of this list is available on the web.<sup>1</sup>

In general, the key to effective risk management is to focus risks that are *specific* to the project at hand. Abstract or generic risks are more difficult to quantify, or should be handled by wider organisational risk management processes. For example, the risk that:

“The project will fail because one of the developers will be lazy.”

is difficult to manage because:

---

<sup>1</sup><http://gin.umd.umich.edu/CIS/course.des/cis375/projects/risktable/risks.htm>

- there isn't a specific cause of the risk, so the probability of occurrence is hard to estimate;
- there is no concrete impact, as the developer's specific role is not specified; and
- the risk affects all the projects in an organisation, so is better managed by an organisation-wide policy.

Instead, risks that are specific to the particular project and the characteristics of the project should be identified in a project risk management process. Risks are also easiest to identify when there is some uncertainty regarding an aspect of the project. For example, recalling the branch library management system project introduced in Section 3.5, consider the risk that:

“The central librarian does not know enough about the central library book management system to address integration questions.”

Example project risk  
(193)

This is a project specific risk that can be identified when it occurs within the project (due to uncertainty concerning requirements) and addressed by actions taken within the project, such as identifying further stakeholders (such as IT support within the library) to interview.

One thing to be aware of is that the risk identification process may also have associated risks! In particular, Glass [1997a] and Cerpa and Verner [2009] have both noted that there are often multiple, interrelated causes of project failure. Consequently it is important to identify relationships between risks and risk management strategies.

### 6.3 Analysing Risks

Recall that project risk is characterised by its probability and impact. This section discusses how risk can be categorised according to these two dimensions.

The probability of a risk occurring can be measured in a number of ways, such as:

- quantitatively, by stating either the
  - probability of occurrence in a given time frame, or
  - the average frequency of occurrence; and
- qualitatively, by grouping numerical probabilities into categories such as low, medium or high.

Measuring risk probability  
(194)

There is no certainty that any project risk will definitely occur. Any risk that are certain to arise are project *constraints*.

There are several problems with estimating probabilities:

Problems in  
estimating probability  
(195)

- Quantitative estimates can be difficult to obtain, because there may not be past experience of the risk, or statistical evidence to draw upon. Even if this is available, it may be unclear whether the risks measured for one project are applicable to another one.
- Qualitative probabilities may be subjective because, although they are often easier to produce, the boundaries between categories or probability may not be well defined, meaning that different estimators will allocate the same risk to different categories.
- There is a substantial amount of psychological research demonstrating that humans are very bad at differentiating between low probability risks. Humans, for example struggle to make a meaningful distinction between a probability of occurrence of 0.0001 and 0.00001 for an event. The two probabilities are below the threshold at which they are just perceived as 'low'.

One approach to handling for this uncertainty is to attempt to estimate the *confidence* of the estimator when they define the probability of the risk. If an estimator has high confidence, then the probability estimate is assumed to be reasonably accurate. However, if the estimator has low confidence, then the real risk probability can be thought of as broad distribution around the estimate. However, this approach assumes that the estimator can judge their own expertise and the complexity of the problem correctly. Unfortunately, psychological research has shown that humans are susceptible to *optimism bias*, indicating an over-confidence in their abilities. For example, on average, humans will estimate themselves to be above average drivers.

The overall impact of a risk on a project can also be measured in different ways:

- quantitatively, in terms of additional resources consumed, such as developer time, money or schedule over-runs; or
- qualitatively, using terms such as minimal, serious, severe, catastrophic.

Again, estimating the impact of a risk can be difficult, because of the particular way in which the risk is manifested. For example, the impact of the absence of a developer will depend on the role they play in a project and the extent to which they bring unique skills or abilities to the team.

In general, three project aspects can be affected by the occurrence of risk:

- product quality and functionality, due to decisions to reduce product feature set or limit quality assurance activities;
- cost, in terms of additional personnel time, implementation of newly identified features, penalties applied by the customer, or acquisition and integration of alternative software

- schedule, as delays arise in a project where some high priority requirements have over-run.

These aspects of a project affected by risk have been summarised by the phrase “cheap, on time, high quality: pick two”, because when a risk is manifested in a project, one of the three will need to be allowed to slip.

There are also other, wider aspects of a software development environment that may be affected by project risk. For example:

- the morale or productivity of personnel in an organisation may suffer due to the perception of failure. This may have effects on the performance of activities elsewhere in the organisation;
- dependent projects may suffer because required deliverables such as software components from the affected project may be inadequate or delayed; and
- decisions regarding risk control may cause other risks to become manifest. For example, introducing extra personnel may actually make a late project later (see below).

Figure 6.3 illustrates a two dimensional space for the overall categorisation of risks in terms of their probability and impact. Risks that have low probability and negligible impact on a project are unlikely to receive much attention within a project. Conversely, risks that have high impact and high probability are serious threats to the success of a project should be addressed as a matter of priority. If these risks cannot be addressed, it may be prudent to consider the long term viability of the proposed project as a whole.

Wider effects of risk (198)

Prioritising risks (199)

## 6.4 Controlling Risk

Referring to Figure 6.3 there are, in principle, three strategies for controlling risks:

- *Risk avoidance* reduces the probability that a risk will occur at all.
- Conversely, *risk mitigation* reduces the impact of a risk should it occur.
- *Contingency planning* develops strategies for alternative actions should a risk manifest itself.

Strategies for controlling risk (200)

A *control* is a specific action to be taken within one of these three control categories. For example, consider the risk proposed on page 135 that the central librarian to be interviewed may not have sufficient knowledge about the implementation details of the central library management system to inform the requirements for the branch system. Strategies for controlling this risk include:

Example risk controls (201)

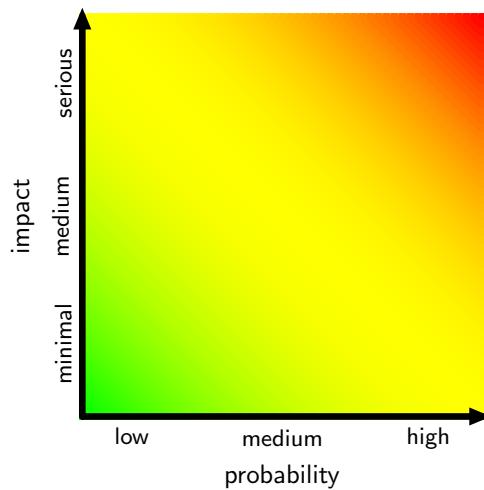


Figure 6.3: categorising risks by probability and impact

- Undertake a survey of library staff to decide who is the most suitable interview candidate (avoidance).
- More than one interview can be planned to ensure a sufficient spread of expertise is approached (mitigation).
- Questions can be included in the interview to determine who else to approach if information is insufficient (contingency planning).

Notice that risk avoidance and mitigation are *preemptive* risk management strategies, attempting to reduce either the probability of a risk manifesting itself, or the consequent impact when it does. Conversely, contingency planning is a *reactive* strategy - identifying controls that can be implemented once a risk has already manifested itself.

## 6.5 Documentation and Risk Management

Documentation and risk (202)

There are two forms of documentation associated with a risk management process:

- a *risk management plan* describing the procedures adopted for each of the risk management activities (identification, analysis, monitoring and so on); and
- a *risk register* describing the actual risks affecting the project.

The risk management plan should document how risks will be managed, including when each management activities will occur, and who in the team is

<b>Id</b>	R1
<b>Category</b>	requirements
<b>Description</b>	The central librarian does not know enough about the central library book management system to address integration questions.
<b>Probability</b>	Medium
<b>Impact</b>	The central librarian does not know enough about the central library book management system to address integration questions.
<b>Aspects</b>	Schedule
<b>Severity</b>	Severe
<b>Status</b>	Monitored
<b>Controls</b>	C1, C2

Figure 6.4: example risk in a risk register

responsible for implementing them. The plan should also describe the schema adopted by the team for documenting risks and controls.

The risk register records the risks and controls that have been identified by the risk management team. Figure 6.4 illustrates an example risk register.

The project team has adopted a qualitative scale for recording the probability and severity, low, medium... and high, minimal, severe, catastrophic... and so on. Each risk is also categorised by source (political, requirements and so on) and affected aspect (schedule, cost or quality). The status of the risk is also recorded, stating whether the risk is being monitored, or whether it has been realised yet.

All risks identified by the project team should be recorded in the register even if they have not yet been fully analysed. The risk register should thus be considered an on-going record of the state of risks affecting the software project. Risk registers may be maintained using general purpose Office productivity software, such as a spreadsheet or desktop database application. Alternatively, specialised risk management software which integrates with software metric monitoring can be used. The format of the risk register should be decided by the team, but should include tables for both risks and controls.

Figure 6.5 illustrates the controls register, which is complementary to the risk register. Each control is categorised as either risk reduction, risk mitigation or contingency plan.

Notice that there is a many-to-many relationship between risks and controls. A risk may be managed by implementing one or more controls. Conversely, a control can also help manage more than risk. We can model this relationship using a UML class diagram as shown in Figure 6.6. If you are unfamiliar with class diagrams, see Chapter 13 for an explanation of the notation.

This class diagram could be further extended using an *association class* (see

Documenting risks (203)

Control Register (204)

Risks and controls class diagram (205)

<b>Id</b>	<b>Category</b>	<b>Description</b>	<b>Status</b>
C1	Cont.Plan.	Reverse engineer central library interface from available software and documentation.	ready
C2	Reduction	Conduct pre-interview survey to identify most suitable interviewees for information on the central library system.	enacted

Figure 6.5: example control register

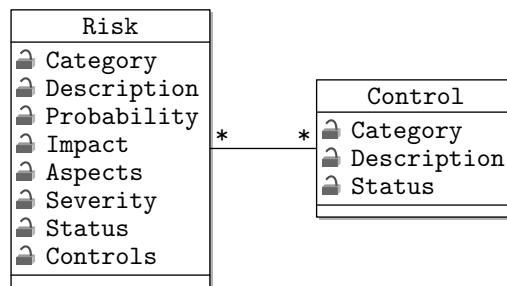


Figure 6.6: risks and controls class diagram

Section 13.3.3), to record the effect of applying a control to a risk. This effect should describe the change in the likelihood of occurrence or impact of a risk.

## Summary

Risk management is a process that runs concurrently to the whole software development life cycle. This is necessary because new risks will be identified during the entire project lifecycle from requirements gathering through design and deployment. Risk management is necessary to develop strategies to prevent project failure.

# **Chapter 7**

# **Quality Assurance**

## **Recommended Reading**

Recommended reading  
(206)

## **Summary**

## **Exercises**

1. Question 1...

# Chapter 8

# Change Management

## Recommended Reading

**PLACE HOLDER FOR sommerville10software**

Recommended reading  
(208)

Any sizeable software development effort must be distributed amongst a number of different developers, software teams or even whole organisations, if the product is to be delivered within a realistic time scale. Project developers may be located in many different countries and work in different time zones, meaning that development efforts are distributed in both space and time. This distribution makes managing changes to a common software base challenging. This chapter describes the processes and tools used to ensure that change occurs in a disciplined, controlled and predictable manner.

### 8.1 Software and Change

Figure 8.1 illustrates a classic change management problem when development efforts are uncoordinated. Two developers are working on the same software project. The project is behind schedule, and a number of important features still need to be completed before an upcoming demonstration. Both developers create local copies of the project software within their own development environments from a master copy kept on a central server. The developers copy changes back to the central server whenever they implement a new feature.

The developers are working on different classes that have a common dependency on a number of library methods in a third class. The first developer makes a modification to their class that means that a new method needs to be added to the utility class; and one of the methods in the utility class is no longer required. The developer is warned by the development environment that the old method is no longer used, so removes it from the class to avoid proliferation of redundant legacy code. The first developer copies both changed files back to the master copy kept on the centralised server.

Risks of uncontrolled  
software changes  
(209)

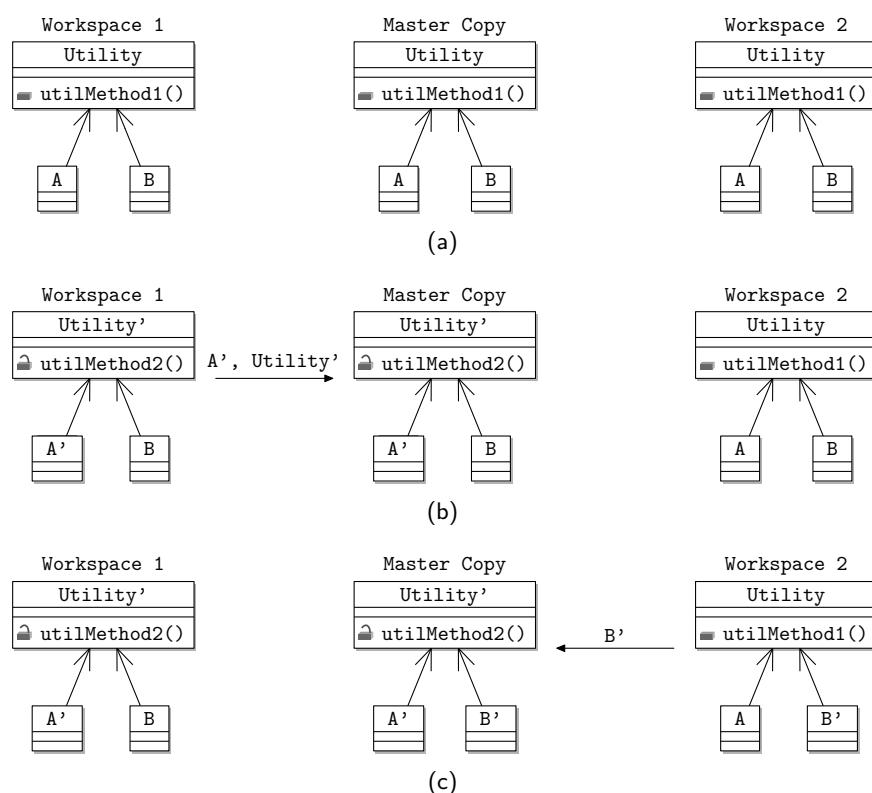


Figure 8.1: conflicts caused by uncontrolled software changes

Simultaneously, and without the knowledge of the first developer, the second developer adds functionality to the class they are working on that depends on the utility class method that has just been removed by the first developer. Her code compiles because she is still using a local copy of the utility class. However, when the modified class is copied back to the centralised master copy, the code does not compile because the utility class in the master copy is missing the required method. Worse, if the second developer copies her version of the utility class to the server the project it will still not be possible to compile the code because the utility class is missing the new method provided by the first developer.

This situation arises because:

- two or more developers are working on parallel development efforts;
- the project team permits uncontrolled change to the source code master copy;
- there was no agreed system of preserving previous (working) versions of the software;
- and the team did not have a procedure in place for notifying others that a change was to be made to a class application programming interface (API).

What caused this problem? (210)

This problem doesn't just concern the program source code for the software project. Things that can change include:

- problem definition and requirements specification;
- the software design and implementation;
- test plan and harness;
- user documentation;
- system dependencies; and
- associated development and management tools.

Change isn't just about source code: example configuration items (211)

All of these artifacts are examples of *configuration items*. A configuration item is any artifact that is created, altered or edited by members of the software team. Artifacts that are generated from other artifacts (such as test results or compiled binaries), or whose change cannot be controlled by the software team (such as external dependencies) are not configuration items.

Change to one configuration item can often cause of *ripple effect* of required changes to other items. Figure 8.2 illustrates an example of this effect. The arrows on the diagram indicate where a change in one configuration item has caused a change in another. Starting at the bottom of the diagram, adding a feature to a requirements specification as a use case (see Chapter 11) means that:

Change and the ripple effect (212)

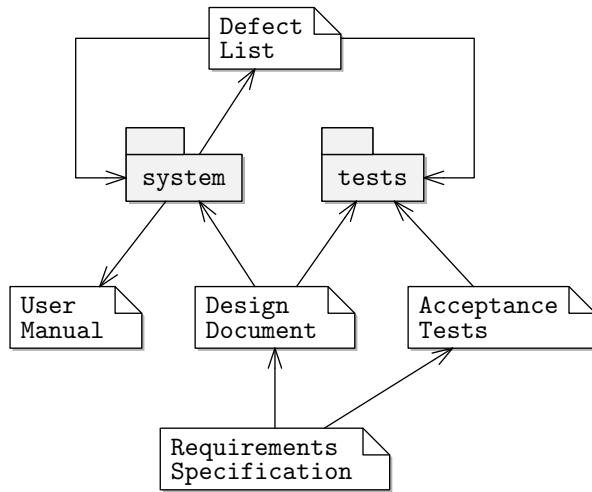


Figure 8.2: the ripple effect caused by a single change to the requirements specification.

- the design must be updated to realise the feature;
- acceptance test cases should be prepared to demonstrate the feature;
- the feature should eventually be implemented in the software source code;
- unit test cases should be developed to demonstrate the design has been correctly implemented; and
- the user documentation should be updated to describe the new feature.

In addition, it becomes apparent that the software change has caused a defect to be introduced into the software. The fault is reported in the bug tracking system used by the team. Further test cases are developed to document the presence of the fault and changes are made to the source code to fix the problem.

Consequently, it is important that changes to configuration items are carefully managed within a process that is understood within the software development team.

## 8.2 Change Management Process

Figure 8.3 illustrates a general process for managing change in a software system, divided into a number of concurrent activities. These activities may be undertaken by one or more software developers or teams, and may be automated to a varying extent, depending on the nature of the software project.

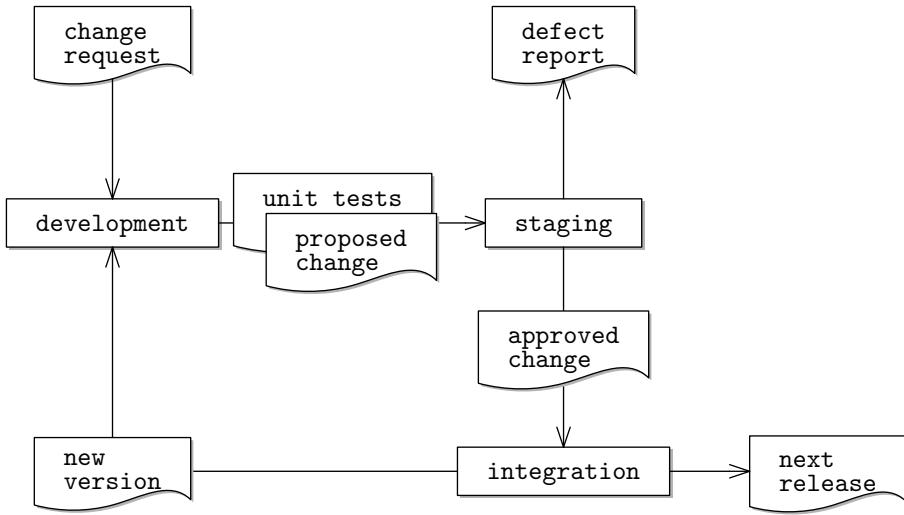


Figure 8.3: a generic change management process

**Development** During development, *change requests*, sometimes called *tickets* are selected for implementation based on the project plan, their priority and/or cost. A change request may describe the addition of a new feature, or the correction of a defect identified by users (see Chapter 26) for a discussion of different causes of software evolution. Change requests are usually stored in a change tracking system, such as Trac (see Chapter 5) or Bugzilla.<sup>1</sup>

The prioritisation of change requests may change depending on the state of the project schedule. If a project is deadline driven, for example, a project may enter a *feature freeze* period shortly before the next release. In the run-up to the release no new features are added and the software development team concentrates on fixing defects.

Developers will typically work on the most up to date stable version of the software system, so it is important to be able to track, manage and distribute the latest versions of the different configuration items to the developers in a team. Consequently, configuration items are accompanied by several different types of meta-data:

- A unique *identifier*, within the scope of the project. The file path of an item in a revision control repository is often the defacto identifier for a configuration item, particularly for source code files. However, this can cause problems if the file is moved to a different location in the hierarchy.
- A *version* or *revision* label, indicating the relationship between the configuration item and its predecessor and successors. Each change made to a configuration item results in a new configuration item with the same identifier but different revision.

Configuration item meta-data (214)

<sup>1</sup>//<http://www.bugzilla.org/>

There are many different models for tracking configuration item versions implemented in different revision management systems. A configuration item can have its own independently maintained version label (as in RCS and CVS), or there may be a project wide version counter (as in Subversion).

- A revision *history* that provides commentary on the changes made by different authors of the software. Each entry in a revision history should record the date of the change, the author and a description of what changes were made and why they were undertaken. Change descriptions may reference defect reports or feature requests stored in an issue tracking system.
- The Principle author or owner of the configuration item. This can be helpful when determining who to inform about a proposed change to a software item.

Older software engineering textbooks refer to a software system *baseline* as a collection of configuration items that have been formally reviewed and documented, for the purpose of further development efforts. This concept has largely been superseded as software change management has become increasingly automated.

Every configuration item in a software project will evolve through numerous versions. In addition, it can sometimes be useful to create two up-to-date versions of the same configuration item so that work can continue simultaneously.

Uses of variants (215) There are several reasons for doing this:

- to experiment with a new feature that (to be implemented) will cause substantial disruption to a large part of the system and its dependencies;
- to undertake a substantial reorganisation of the configuration items managed within the software project;
- to support a specialised variant that will deployed on a platform or within an organisation with particular needs; or
- to create a release that is to be preserved for reference within the repository (this is sometimes called tagging).

Branches and merges  
(216)

Figure 8.4 illustrates a small scale example of this situation.

The figure illustrates changes to a class that implements the *factory pattern* (see Section 18.3.2). A feature request is made for the *Product* to be configurable. The *Factory* class is initially changed by adding an attribute to indicate what type of product should be produced. This is a quick solution to the problem, however it is unsatisfactory because it isn't possible to use a single instance of the factory class to produce different products.

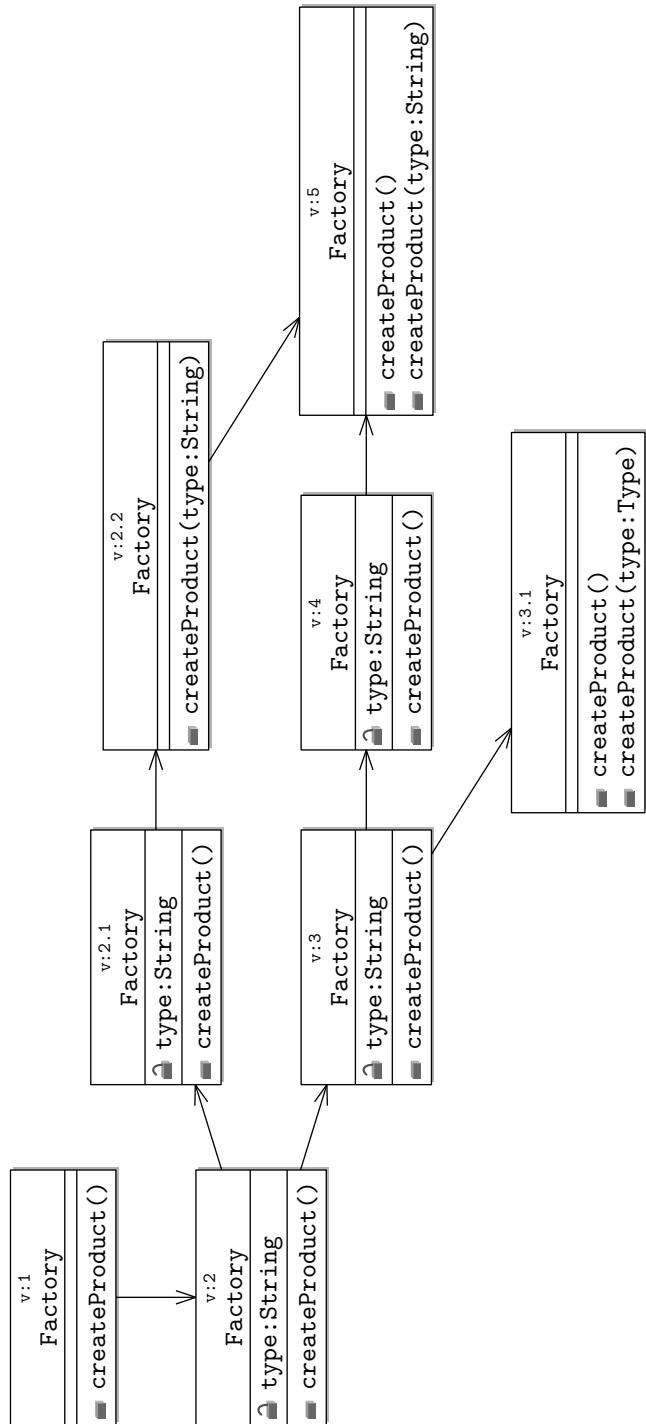


Figure 8.4: branching and merging during change management

Consequently, a developer decides to experiment with adding a parameter to the `createProduct()` method. To do this, she creates a *branch* of the class, by creating a copy labelled `v:2.1`. The version tree labelled `2,3` and so on is referred to as the *trunk* or *canonical* variant of the software.

Changes to the two versions of the class can now proceed in parallel. The developer substantially alters the `createProduct()` method by adding a **type** parameter, resulting in a new version labelled `v:2.2`.

Simultaneously, additional work is done on the main *trunk* version of the class, resulting in the version labelled `v:4`. Finally, the developer working on the branch merges the changes she has made back into the main development effort, resulting in the version labelled `v:5`. This class contains the new method with the type parameter and the old version that is altered by the developer to deliver a default product type when called.

Notice also that a separate developer has attempted to tackle the same problem by creating another branch labelled `v:3.1`. This developer has decided to use an enumeration to flag the different product types. Work on this version is incomplete and has not yet been merged back into the main development effort.

## Staging (217)

**Staging** Changes to software need to be checked before they are released to a wider user base, in order to gain assurance that any proposed change does not introduce additional defects. A *staging platform*, an environment configured to replicate or simulate the conditions the software will be used in, is used to test the software before it is released to users. The staging platform is configured with the current hardware and software dependencies of the current baseline of the software system.

Sometimes it may be necessary to have more than one staging environment. The system may be comprised of several distributed components, each of which are intended for use on a different software platform, or the same component may need to be executable on different platforms. For example, a monitoring system for a server farm may have a number of agent components operating on a collection of embedded sensors (temperature, humidity, power and so on); a service component running on a server monitoring the software agents; and a number of client 'dashboards' for end users on common desktop operating systems. Each of the environments in which the software is intended to be run need to be available for testing during staging.

The proposed change is delivered to the staging platform, along with unit tests developed alongside the change and any accompanying documentation. These items can be subjected to a variety of quality assurance procedures, including:

- application of unit tests to the proposed change;
- static analysis of the proposed change;

## Quality assurance procedures (218)

- evaluation of the provided unit test case coverage;
- assessment of supplied technical documentation, such as requirements specifications and design documents;
- application of system integration tests; and
- application of system acceptance tests.

If the proposed change passes the quality assurance procedures it can be passed on to production for integration into the product base for the system and distribution in future releases. If defects are discovered, the software change may be rejected until they are fixed, or the defects may be reported in the issue tracking system.

Of course, it isn't always feasible to replicate all aspects of the 'real' environment in which the production version of the software will be used. This may be because:

- the system will be distributed on too many computing nodes to test realistic scenarios;
- the system will be used on too many different platform configurations to test them all;
- the system will have too many simultaneous users;
- there are too many diverse user types for all to be tested;
- only one platform for the software is available and is used for production; or
- software dependencies (such as libraries or external data sources) cannot be accessed from within the organisation for security reasons.

Consequently, it is sometimes necessary to develop approximations to some of the features of the production environment. Several possibilities are:

- emulate the functionality of a missing hardware or software component;
- simulate the behaviour of different user types using automated user interface test frameworks; and
- run a distributed application on a simulation of the real network.

You can find out more about the challenges of large scale testing of systems in Chapter 22.

Limits of staging platforms (219)

Approximating real world conditions (220)

Release schedules and types (221)

**Production** *Software releases* are compiled during *production*, according to a *release schedule*, typically a release is built according to an automated process and assigned a release version label, which may be a different scheme to that used for configuration item versions. A release version may be a prepared by *tagging* the collection of configuration items to be distributed.

As discussed in Section 5.2.1, software processes are typically either feature or deadline driven, and the same is true for releases. A software release may mark:

- a major project *milestone*, representing, for example, implementation of all high priority use cases; or
- according to a project calendar issuing new releases every week, month, or quarter for example.

Releases can also be produced for several different purposes within a project life-cycle or iteration, depending on the software process followed:

- *Nightly builds* are sometimes called cutting edge (or bleeding edge) releases because they represent the most up to date version of the software product submitted by the development team that day. The software is typically not intended for general purpose use, but are instead compiled for initial testing details purposes.
- *Beta test releases* are typically distributed to a restricted, select group of users (see Chapter 21) for a first stage of acceptance testing.
- *Release candidates* are anticipated to contain all functionality to be incorporated in the final release, although the software may still contain some known but less critical defects.
- *Production release* are sometimes called go-live releases. These are distributed to the general user base of the system.

### 8.3 Communicating Change

Change management plans (222)

Changes needs to be communicated to others who might be affected by this change. This is done through the development of a *change management plan*, which should describe:

- how proposed changes are to be managed, including software tool support;
- what process is followed for making changes;
- who is responsible for agreeing, authorising or accepting changes;
- who needs to know about changes to different parts of the system; and

- how proposed changes are communicated;

These issues need to be resolved for each aspect of the project subject to change management, i.e all types of configuration item.

The range of people who need to be informed about a change will depend on the *scope* of the feature subject to the proposed change. For object oriented systems, there are three broad *defacto* levels scope for a feature [Fowler, 2000]:

- *Private* internal implementation details of a class, such as the implementation details of a method, the identifier for a private attribute, or the signature for a private method. Changes at this level should only concern developers working on the same class.
- *Public specification of a class*, such as a public method signature, or the identifier and type of a public attribute. Changes to public features concern other developers in the software team who are working on classes dependent on the feature that will change.
- *Published features* of a software product comprise those parts of the public specification that are intended for use by developers external to the software development team.

Clearly, the fewer people who need to be informed (and potentially disrupted) by a proposed change, the better. Consequently, it is desirable to keep as much of a class's implementation details as possible hidden from developers who use the class. Similarly, it is generally preferable to maintain a stable published API for a system. If the published API changes too frequently, users may decide to opt for a different system (an unstable API is also indicative of an immature system that may contain numerous defects).

One way of managing changes to public and published features of a class is to implement the new version of the feature in parallel to the old version. The old version can then be documented as deprecated, meaning that it is a feature that has been left in place for compatibility reasons, but will removed from a future release of the wider system.

This approach gives users notice that their software will eventually stop working, but gives them time to adapt their own systems to the new way of doing things, rather than forcing them to make the change immediately.

Typically, the documentation for a deprecated feature should include:

- the scope of the deprecation, i.e. what features are affected;
- the release version of the software in which the new way of doing things first appeared;
- a schedule for when the deprecated feature will be removed;

Scoping the  
dependants of change  
[Fowler, 2000] (223)

Deprecated features  
(224)

- an explanation of why the feature has been deprecated; and
- a description of how to change dependent code, or a reference to the new recommended feature.

## Deprecation in Java (225)

The Java platform has built in support for deprecating features. Any class, or class member can be given the annotation `@Deprecated`. For example:

```
public abstract class AbstractFactory {
    /**
     * @Deprecated As of release 2.0, replaced by
     * makeProduct(String). To be removed in release 3.0.
     * @return a Product instance with a concrete factory
     * implementation specific configuration.
     */
    public abstract Product createProduct();

    /**
     *
     * @param type
     * configuration string for product creation.
     *
     * @return
     */
    public abstract Product createProduct(String type);
}
```

When a program that uses the `Factory` class is compiled, any uses of the deprecated `makeProduct()` method generate warnings to the user.

If there is a substantial amount of change to the published interfaces of a software product between two production releases, it may be necessary to provide a *migration plan* for users of the software. A migration plan should describe the steps that a developer should take when upgrading the version of a software dependency. The plan may need to include:

- a means of estimating how long the migration will take;
- a summary of known issues with the new release of the software, including situations in which migration should not be performed;
- data migration, including necessary backups and alterations to data formats;
- upgrades of any sub-dependencies of the changed software;
- necessary changes to configuration methods and options; and
- how to adapt existing source code to use the new published APIs.

Ideally, as much of the migration process as possible should be automated. There are many tools available for handling many of the common tasks of preparing migration scripts, called *package managers*.

Local	Revision Control System (rcs)	1982
Client-Server	Concurrent Versions System (cvs)	1990
	ClearCase	1992
	Subversion (svn)	2000
Distributed	BitKeeper	2000
	Mercurial (Hg)	2005
	Git	2005

Table 8.1: examples of automated revision management systems

## 8.4 Automated Version Management

Like many software engineering practices, many of the activities of configuration management are repetitive and monotonous, and so suitable for automation. There has been a plethora of software tools developed for managing document revisions. A small selection of the more popular tools is shown in Table 8.1.

There are three main categories of version control software shown in the table:

Examples of version management software (227)

- *Local* version control systems operate within a single development environment. Previous versions of a file are typically contained within the same or a sub-directory as the current version.

Local version control is particularly useful for managing configuration files for deployed systems such as web servers, because administrators can review previous versions of a configuration file in the same directory. However, it is difficult to coordinate collaborative projects using local revision control because

- *Client-server* or centralised version control hosts all configuration items in a central repository. The repository can be accessed via a variety of protocols, such as HTTP or SSH. Each client maintains a copy of the most recent version of each configuration item contained in the repository. If necessary, older versions of the configuration item can be accessed from the repository.

The client server model is useful for managing collaborations within small teams, as it provides a mechanism for collating and distributing changes to a software system as a project progresses. However, the model can be difficult to apply in more disparate software development efforts (such as open source projects) where it may be difficult to apply a formal process of change control or maintain a single agreed version of a system.

- *Distributed version control* developed in the open source community in response to the challenges encountered controlling change when a project lack a single focal point of control. Each development environment is

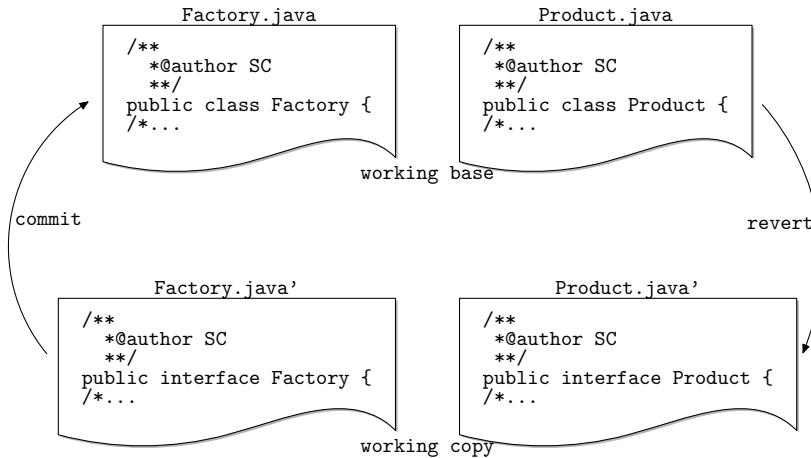


Figure 8.5: working bases and working copies

effectively a repository, similar to the local model described above. In addition, the distributed model supports a hierarchy or peer-to-peer network of local repositories to exchange changes as required.

A disadvantage of distributed version control is that it's very flexible can make it more difficult to manage within a project. A developer may struggle to conceptualise which version of the software they are working on when changes are distributed amongst different developers.

Regardless of the particular software used, the process followed by a developer for interacting with a version control system is largely the same. Each developer maintains a *working base* and *local working copy* of each configuration item. The working base is the most recent version of the configuration item submitted to the version control repository. The working copy records any changes to the working base made locally by the developer. Figure 8.5 illustrates this arrangement.

The figure illustrates the `Factory` and `Product` classes that the developer has been working on, as described above. The figure shows the state of the local data held by the version control software. The working base contains the current versions of the `Factory` and `Product` classes. The figure shows that the developer has made some further changes to these classes.

In particular, the developer has decided to turn them into interfaces, so that a number of different concrete versions of the classes can be implemented according to a common specification. These changes are reflected in the state of the working base. If the developer keeps these changes, they will need to *commit* them to the repository, which simultaneously updates the working base.

If on the other hand they decide to discard the changes, they will be able to *revert* the state of the local working copy back to that of the working base. This will mean any local changes to the software will be lost.

## Working bases and working copies (228)

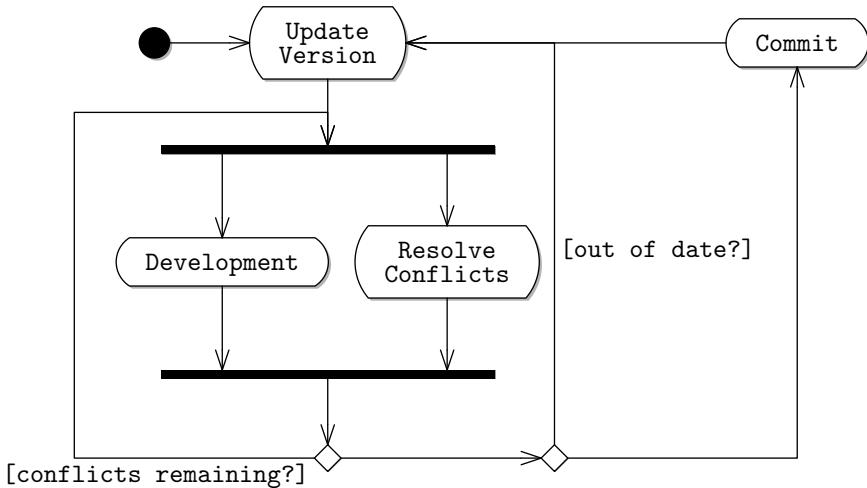


Figure 8.6: the update, resolve, commit cycle

Typically, developers using version control software will work on the same part of a system simultaneously. Consequently, there can be *conflicts* if two developers attempt to commit different changes to the same configuration item simultaneously.

Developers work in a cycle of activities with the repository, called the *update*, *resolve* and *commit* cycle. Figure 8.6 illustrates this process from the perspective of a single developer, using a UML *activity diagram* (see Chapter 12).

Initially, the developer obtains a copy of the software held in the version control repository using an initial update action. As the developer does not have a local version of the software, there shouldn't be any conflicts at this point. The developer works on the software for a period of time, making some changes.

When the developer has finished, she can now attempt to commit the changes she has made back to the repository. There are not currently any local conflicts, so the next step is to check whether her working base is still up to date. Unfortunately, she discovers that someone else has made changes while she was working, so she must update her working base. This causes a number of conflicts that must be resolved, although she can also work on the development at the same time.

Once the conflicts have been resolved, it is necessary to check the repository again to make sure that no further updates have taken place. This sub-cycle continues until the developer has the most up to date version of the software as her working base, immediately prior to committing changes. When this happens, she will again be in the position of having the most up to date version of the software as her working base without any conflicts.

A developer can minimise the amount of time spent on resolving conflicts by frequently updating their working base. This minimises the risk of their

The update, resolve, commit cycle (229)

## Automated version management (230)

own copy becoming very outdated by changes made by other developers (and therefore having to resolve a large number of conflicts). Thus, they are more likely to be making their own changes to the most recent version of the files kept in the repository.

Aside from configuration management, there are several other benefits of using a version control repository for managing electronic documents:

- for synchronisation between development environments;
- for creating backups;
- for tracking the evolution of a system; and
- for managing collaborative developments.

## Summary

Change management requires a combination of agreed practices *and* software tools. It is important to agree standards for making changes to software, alongside the implementation of tool support. Allowing standards to emerge informally can lead to uncontrolled change, even with a repository in place, or it can mean that tools are not used effectively.

## 8.5 Workshop: Version Control using Subversion

This workshop introduces the Subversion version control system. The Subversion<sup>2</sup> manual is an excellent source of further guidance on using the subversion system. The workshop assumes that you have already tried the Trac workshop in Section 5.4, or that you are familiar with the Linux command line console.

Subversion is well supported on all the major development platforms, including:

- client applications for Windows (the TortoiseSVN<sup>3</sup> Windows GUI is an excellent example of open source software)
- On \*nix systems, installation can be done using package managers, e.g.
  - aptitude on Debian and Ubuntu:

```
sudo apt-get install svn
```
  - or the Yellow dog update manager on Redhat and CentOS

```
sudo apt-get install svn!
```

Installing Subversion  
(231)

Subversion is already installed on the server you will be using, however. However, you may also wish to install the client and/or server parts of the software on other systems for your personal use. Subversion repositories can also be accessed via http using the apache2 web server with libapache2-svn module.

Start by logging into the project server, however (see Section 5.4).

The easiest way to get help with Subversion is to type:

```
%> svn help
usage: svn <subcommand> [options] [args]
```

Getting help with  
Subversion (232)

```
Available subcommands:
  add
  blame (praise, annotate, ann)
  cat
  changelist (cl)
  checkout (co)
  cleanup
  commit (ci)
```

Which provides an overview of the sub-commands supported. You can also get help with a specific command, for example:

```
%> svn help commit
commit (ci): Send changes from your working copy to the
repository.
usage: commit [PATH...]
```

Getting help with a
specific sub-command
(233)

<sup>2</sup><http://svnbook.red-bean.com/en/1.5/svn-book.html>

<sup>3</sup><http://tortoisesvn.tigris.org/>

```

A log message must be provided, but it can be empty. If it
is not
given by a --message or --file option, an editor will be
started.
If any targets are (or contain) locked items, those will be
unlocked after a successful commit.

Valid options:
-q [--quiet] : print nothing, or only summary
               information
-N [--non-recursive] : obsolete; try --depth=files or --
                      depth=immediates
--depth ARG : limit operation by depth ARG ('empty',
               'files',
               'immediates', or 'infinity')
--targets ARG : pass contents of file ARG as
               additional args
--no-unlock : don't unlock the targets
-m [--message] ARG : specify log message ARG

```

### 8.5.1 Creating a Repository

The first step is to create a Subversion repository. You may have already taken this step following the Trac workshop (see Section 5.4). If so your repository contents will be visible from the Trac web interface, although you won't be able to alter content from there.

```
%>cd /extra/2012/z/
%>mkdir repos
%>svnadmin create repos
```

You can also configure Subversion to be accessible over the network using the `svn` and `http` protocols (this is not necessary for this workshop).

Once we have the repository created we need to follow some conventions for storing configuration items in it. In particular we need to create three top level directories in the repository, `trunk`, `tag` and `branch`:

- The `trunk` directory is used to store the canonical variant of the software product under version control.
- The `branch` will contain other working variants.
- The `tag` directory will contain 'frozen' variants of the software product, typically variants which represent particular releases or deliverables.

The `mkdir` sub-command is used to create directories in a repository, so use it to create three high level directories described above:

```
%>svn mkdir file:///extra/2012/z/repos/trunk \
-m "Creating trunk."
```

Creating a repository  
(234)

Directory conventions  
in Subversion (235)

Creating trunk,  
branch and tag  
directories (236)

```

Committed revision 1.
%>svn mkdir file:///extra/2012/z/repos/branch \
-m "Creating branch."

Committed revision 2.

%>svn mkdir file:///extra/2012/z/repos/tag \
-m "Creating tag."

Committed revision 3.

```

Notice that we need to use a URL (using the file protocol) when interacting with a repository.

Some organisations keep separate trunk and branch directories for each project in a single repository. Others prefer to keep separate repositories for each project. How you organise your own repository should be decided within your project team.

Now let's add a project under the `trunk` directory.

```

%>svn mkdir file:///extra/2012/z/repos/trunk/blib \
-m "Creating Branch library project."

Committed revision 4.

```

Next, a new project can be imported into the repository, ready to be checked out. We'll use some of the  $\text{\LaTeX}$  deliverable templates available with the coursework that accompanies these lecture notes as an example. If you are unfamiliar with  $\text{\LaTeX}$  and do not wish to learn about it now, you can follow the workshop by making edits to the `README` file in the template directory.

You can download the templates from your moodle course page and transfer them to the `hoved` server using `FTP` or `putty`.

```

%>cd ~/templates
%>ls -l
d1.tex  d2.tex  d3.tex  d4.4.tex  img  deliverable.cls  README

```

Now import the templates into a `doc` sub-directory within the `branch` library directory using the following command:

```

%>svn mkdir file:///extra/2012/z/repos/trunk/blib/doc \
-m "Adding documents directory."

Committed revision 5.

%>svn import ./ file:///extra/2012/z/repos/trunk/blib/doc \
-m "Adding document templates."
Adding          d1.tex
Adding          d2.tex
Adding          d3.tex
Adding          d4.4.tex
Adding          img
Adding (bin)    img/CompSci_mono.eps

```

Adding a project directory (237)

Importing content (238)

```

Adding  (bin)  img/CompSci_mono.pdf
Adding          README
Adding          deliverable.cls

Committed revision 6.

```

**Viewing the repository  
file system (239)**

The contents of the repository and any sub-directory can be viewed using the `list` command:

```
%> svn list -R file:///extra/2012/z/repos/
branch/
trunk/
trunk/blib/
trunk/blib/doc/
trunk/blib/doc/README
trunk/blib/doc/d1.tex
trunk/blib/doc/d2.tex
trunk/blib/doc/d3.tex
trunk/blib/doc/d4.4.tex
trunk/blib/doc/deliverable.cls
trunk/blib/doc/img/
trunk/blib/doc/img/CompSci_mono.eps
trunk/blib/doc/img/CompSci_mono.pdf
```

**Making a working  
copy - checking out  
(240)**

The `-R` switch makes the listing recursive.

Next we can check out a copy of the latest version of the project by using the `checkout` command. You can check the project out to anywhere in your file system. As I do a lot of work with Eclipse, I tend to keep my local working copies of projects in a directory called `workspace` in my home directory:

```
%> cd ~(workspace
%> svn checkout file:///extra/2012/z/repos/trunk/blib ./blib
A   blib/doc
A   blib/doc/d4.4.tex
A   blib/doc/img
A   blib/doc/img/CompSci_mono.eps
A   blib/doc/img/CompSci_mono.pdf
A   blib/doc/README
A   blib/doc/deliverable.cls
A   blib/doc/d1.tex
A   blib/doc/d2.tex
A   blib/doc/d3.tex
Checked out revision 6.

%> cd blib
%> ls -A
.svn  doc
```

Notice that the local `blib` directory is created automatically by Subversion.

### 8.5.2 Working with Versions

Once some changes have been made, it is necessary to check them back into the revision control system. Make a change to `d2.tex` by entering the correct

list of authors in the `author` macro:

```
%%authors
\author{
    Sheldon Cooper \\
    Howard Wollowitz\\
    Raj Koothrapali
}
```

Changing some content in a document (241)

We now need to commit the changed document to the repository. We do this in three stages. Initially, we should review what changes we have made using the `stat` command.

```
%>cd ~/workspace/blib
%>svn stat
M      doc/d2.tex
```

Committing a change (242)

This shows that we have modified the D2 deliverable document. Next, we should check that we aren't going to get any conflicts by making the commit by updating our own working copy:

```
%>svn update
Updating '.':
At revision 6.
```

This shows that no other changes have been made while we have been working. Finally, we can commit our changes to the repository.

```
%>svn commit ./ \
-m "Fixed author names in Deliverable 2."
Sending      doc/d2.tex
Transmitting file data .
Committed revision 7.
```

We can review the history of changes made to the code using the `log` sub-command:

```
%>svn log
-----
r6 | tws | 2012-09-26 07:56:39 +0100 (Wed, 26 Sep 2012) | 1
line

Adding document templates.
-----
r5 | tws | 2012-09-26 07:51:59 +0100 (Wed, 26 Sep 2012) | 1
line

Adding documents directory.
-----
r2 | tws | 2012-09-26 07:46:50 +0100 (Wed, 26 Sep 2012) | 1
line

Creating Branch library directory.
```

Reviewing change history (243)

## Adding new files (244)

The command lists all the previous versions of the specified URL. Notice that only changes that affected the specified path are listed. You should also try using the `log` command on the `workspace/blib/doc` directory.

Once a working copy has been created, it may be necessary to add further additional files to the working copy, so that they will be included in subsequent revisions. The `add` command can be used to achieve this.

Suppose that a new deliverable is started based on the `d1.tex` template. The deliverable can be added into the working copy and committed to the repository as follows:

```
%> cd ~/workspace/blib/doc/
%> cp d1.tex d5.tex
%> svn add d5.tex
A      d5.tex
%> svn commit -m "Created new Deliverable D5"
Adding      d5.tex
Transmitting file data .
Committed revision 8.
```

## Reverting changes (245)

Sometimes we may decide to discard the changes we have made to our working copy and revert to the working base version. The `revert` command allows us to do this. For example, if the `deliverable.cls` template becomes corrupted by an attempted modification and it isn't possible to fix the problem, the command:

```
%> svn revert deliverable.cls
Reverted 'deliverable.cls'
```

Will cause all changes since the last update to be lost.

This also works if we add a file into the working copy by mistake, for example, temporary files generated by L<sup>A</sup>T<sub>E</sub>X:

```
%> latex -quiet d1.tex
%> ls -A
d1.aux  d1.log  d2.tex  d4.4.tex  deliverable.cls  README
d1.dvi  d1.tex  d3.tex  d5.tex      img
%> svn add *
A      d1.aux
A  (bin)  d1.dvi
A      d1.log
%> svn revert *.aux *.dvi *.log
Reverted 'd1.aux'
Reverted 'd1.dvi'
Reverted 'd1.log'
```

### 8.5.3 Using Properties

Subversion supports the use of *properties* to attach metadata to files and directories under version control. One particular use of this is to specify what

files should be ignored by the `stat` command when listing files in a directory. The following script shows how to specify that certain file extensions should be ignored.

```
%> svn propset svn:ignore "*.*aux  
> *.dvi  
> *.log" ./  
property 'svn:ignore' set on '.'  
  
%> svn commit \  
-m "Configured the doc directory to ignore temporary latex  
files."  
Sending .  
  
Committed revision 10.
```

Ignoring  
non-configuration  
items (246)

Notice the open string quote on the first line terminated on by a closing quote on the third. This is necessary as each ignore pattern must start on a new line.

The property mechanism can also be used to displayed metadata inside printed documents. The first step is to specify that Subversion should change the information inside the configuration item when it is updated:

```
%> svn propset svn:keywords 'Id' d1.tex  
property 'svn:keywords' set on 'd1.tex'
```

Adding version  
metadata to  
documents (247)

Then add the following lines to the document (already on `d1.tex`).

```
\usepackage{svn-multi}  
\svnid{$Id:$}
```

Finally, commit the changed file to the repository.

```
%> svn commit \  
-m "Configured d1.tex to display metadata"  
Sending d1.tex  
Transmitting file data .  
Committed revision 11.
```

The line in `d1.tex` will now look something like this:

```
\svnid{$Id: d1.tex 10 2012-09-26 11:35:30Z 1234567c $}
```

You can now use the following  $\text{\LaTeX}$  commands from the `svn-multi` package to pretty print this information in the  $\text{\LaTeX}$  document:

- `\svnrev`
- `\svnday`
- `\svnmonth`
- `\svnyear`
- `\svnauthor`

Full documentation on integrating Subversion and L<sup>A</sup>T<sub>E</sub>X can be found in the `svn-multi` package on CTAN.<sup>4</sup>

#### 8.5.4 Branches, Tags, Conflicts and Merges

Branches in Subversion are copies of part or all of the trunk directory stored (by convention) under the `branch` directory. Suppose, for example, that one member of the branch library development team is tasked with re-organising the layout of how tasks are presented in the project schedule. This could potentially be disruptive for the entire project plan. Consequently they can create a branch of the `doc` trunk sub-directory and make their change to that instead.

Creating branches  
(248)

The `copy` command is used to create a branch of the trunk:

```
%> svn copy \
  file:///2012/extraz/repos/trunk/blib/doc\
  file:///2012/extraz/repos/branch/blib-alt-sched\
  -m "Creating branch to experiment with alternative layout
  of schedule."
Committed revision 12.
```

Notice that only the `doc` directory has been copied to the branch. This minimises the potential source of conflicts if the branch is later merged back into the trunk.

Creating tagged versions follows exactly the same procedure, except that copies are made into a sub-directory of the `tag` top level directory. Try making a tagged release of the entire `trunk/blib` project called `v0.1`.

The next step is to merging the branch you have created back into the trunk. To do this, you will need to first take the following steps (using what you have been shown above):

1. Check the `branch/blib-alt-sched` directory out of the repository into a new working copy.

Hint: you can also use the `switch` subcommand to switch the existing working copy of the `blib/doc` directory to the branch. Look up the documentation of the `switch` command in the manual, or use `help` subcommand to work out how to do this.

2. Make some modification to one of the deliverables in the `blib-alt-sched` working copy. The example transcript assumes that the `d2.tex` file has been modified so that the way tasks are presented is changed.
3. Commit the working copy of the branch to the repository.
4. Return to the working copy of the trunk `blib/doc` directory.

Now, all that remains is to merge the changes made to the files back into the trunk and commit the working copy to the repository:

---

<sup>4</sup><http://www.ctan.org/tex-archive/macros/latex/contrib/svn-multi>

Merging variants  
(250)

```
%> svn merge file:///extra/2012/z/repos/branch/blib-alt-sched
--- Merging r12 through r13 into '.':
U   d2.tex
--- Recording mergeinfo for merge of r12 through r13 into '.':
U   .

%> cd ..
%> svn commit -m ""
Sending      doc
Sending      doc/d2.tex
Transmitting file data .
Committed revision 14.
```

The merge command can cope with much more complicated merger scenarios than the one presented here. You can find out more from the Subversion manual.

Sometimes when a working copy is updated or merged, it results in conflicts between the copy in the repository and the copy held locally. This can happen when two developers are working on the same set of configuration items at the same time.

For example, suppose that you and another developer are working on the file `doc/d3.tex` in the trunk of the project. You will need to play the part of the other developer, by checking out a separate working copy of the `blib` project to another local director (e.g. `blib2`), or get a team colleague to do so.

Both of you change the list of authors from the template. You lists the authors in alphabetical author by surname:

[Resolving conflicts \(251\)](#)

```
\author{%
  Sheldon Cooper \\
  Leonard Hoffstadter \\
  Raj Koothrapali \\
  Howard Wollowitz
}
```

and your colleague lists them by (perceived) contribution:

```
\author{%
  Howard Wollowitz \\
  Raj Koothrapali \\
  Sheldon Cooper \\
  Leonard Hoffstadter
}
```

Commit your changes first. Then get your colleague to commit their changes. As normal, they should first update their working copy as shown below:

[Conflicts during updates \(252\)](#)

```
%> svn update blib2
Updating 'blib2':
Conflict discovered in '/home/tws/workspace/blib2/doc/d3.tex'.
Select: (p) postpone, (df) diff-full, (e) edit,
       (mc) mine-conflict, (tc) theirs-conflict,
       (s) show all options:
```

## Conflict diff report from subversion (253)

The update results in a reported conflict. The dialogue offers a number of options for resolving this. Your colleague should first choose the `diff-full` option to show the difference between the three conflicted version of `d3.tex`. Fortunately, the `diff` file shows that the difference is quite small:

```
@@ -22,10 +22,21 @@\n\n\\title{Requirements Document}\n\n-\\author{Author 1 \\\n-      Author 2 \\\n-      Author 3 \\\n-      ...}\n+<<<<< .mine\n+\\author{%\n+ Howard Wollawitz \\\n+ Raj Kuthrapali \\\n+ Sheldon Cooper \\\n+ Leonard Hoffstadter\n+}\n+=====+\n+\\author{%\n+ Sheldon Cooper \\\n+ Leonard Hoffstadter \\\n+ Raj Kuthrapali \\\n+ Howard Wollawitz\n+}\n+>>>>> .r15\n\n\\date{9 January 2009}
```

The output shows the difference between:

- the working base (r14),
- the working copy held by the second developer and
- the version already committed by the first developer (r15).

## Options for resolving conflicts (254)

There are a range of possible options at this point:

- accept the full version of one of the files (working or committed);
- accept all conflicts from one version of the file (working or committed);
- use an external tool to edit conflicts;
- resolve the conflicts manually (leaving differences marked up in the file);  
or
- postpone resolution for later.

rectory containing  
nflicted file (255)

Have your colleague choose to manually edit the file, so types 'p'. This also leaves several extra files in the directory containing the different versions of the conflicted file:

```
%> ls -w 60
d1.tex  d3.tex.mine  d4.4.tex          img
d2.tex  d3.tex.r14   d5.tex            README
d3.tex  d3.tex.r15   deliverable.cls
```

This means they will need to clean up the difference markup themselves, either manually using a text editor, or through an external interactive merge tool.

In any case, once the conflict has been cleaned up (you don't actually have to alter the file for the purpose of this workshop) your colleague can tell subversion that the conflict has been satisfactorily resolved and commit the new version:

```
%> svn resolve --accept working d3.tex
Resolved conflicted state of 'd3.tex'

%> svn commit \
  -m "Resolved authorship standard with SC."
Sending      d3.tex
Transmitting file data .
Committed revision 16.
```

A directory containing  
a conflicted file (256)

Notice that this means that a version control tool cannot resolve conflicts for you, and doesn't solve the problem of conflicting versions entirely. Rather, it provides a means of tracking and managing conflicts, as well as preventing the loss of work through unmanaged over-writes of changes.

### 8.5.5 Going Further

There are numerous other useful commands include:

Other Useful  
sub-commands (257)

- export
- diff
- info
- delete
- lock
- relocate
- blame

You can investigate how to use them in the Subversion manual.



# **Chapter 9**

# **Software Metrics**

## **Recommended Reading**

Recommended reading  
(258)

## **Summary**

## **Exercises**

1. Question 1...

## Chapter 10

# Documentation and Technical Writing

### Recommended Reading

PLACE HOLDER FOR sommerville10software

Recommended reading (260)

### 10.1 General Considerations

Some questions to consider when producing written documentation include:

Considerations (261)

- Who are your audience, what can you expect them to know?
- What are the key concepts you want to convey?
- How widely disseminated will the work be?
- What will be the final format?

There are several types of software documentation.

Types of software documentation (262)

- *project* or *software process* documentation describes the processes undertaken by the project team and some of their outputs, including the project organisation and schedules, test plans and acceptance test reports;
- *product* documentation describes the underlying software application or component, including the requirements specification, design description, and source code comments;
- *maintenance* documentation describes the status of a system and includes release notes describing what has changed in the system since the previous release, and defect tracking reports; and

- *user* documentation, which provides information on how to install, configure, administer and use a system. User documentation may include a tutorial or guided tour of a systems features for getting started, as well as a complete reference manual for the system's feature set.

## 10.2 Structuring Documentation

Structuring a technical document or paper (263)

Any technical document will follow a broadly standard pattern, as follows:

1. Meta data
2. Abstract or executive summary
3. Introduction
4. Main body section 1
5. Main body section 2
6. ...
7. Summary and or Conclusions
8. References
9. Appendices

This basic patterns is generally applicable regardless of the size of the document, or whether the high level sections are called parts, chapters or sections. However, the structure of a document does need to reflect the contents of the document. It is not useful to slavishly follow this pattern if this results in awkwardly structured prose. The basic template for technical documentation is a good place to start when writing documentation, but it should be expected that the document structure will change as the target of the documentation process is better understood. Each of the parts of the

Meta-data (264)

**Meta data** is information attached to a document to support indexing, archiving and subsequent searching and retrieval. Depending on the document type, meta data can include the:

- title;
- names of the author(s);
- date of publication;
- owning or sponsoring organization(s);

- contracted organization(s);
- identifier;
- revision number of version;
- sensitivity or clearance level, public or confidential for example;
- document status, preliminary, draft or final for example;
- revision schedule;
- revision history;
- keywords; and/or
- related documentation.

Many organisations provide templates for their documents to encourage standardization. The L<sup>A</sup>T<sub>E</sub>X preparation system also provides standard macros for specifying document title, author and so on.

**The abstract** provides summary of the contents of the paper, describing the motivation and noting the key results or findings.

Abstract (265)

**The Introduction** is the first top level section sets the scene, and explains how the paper will progress. The introduction should be used to provide a foundation for the rest of the paper. This includes:

Introduction (266)

- a justification for the work described in the document. This should explain the motivation or purpose for writing the document. A typical way of introducing the documents motivation is a sentence beginning with “This paper argues that...” or “This report concerns...”;
- the key definitions and concepts necessary to understand the rest of the document;
- references to earlier or companion documents necessary to understand the concepts of this document; and
- a final sub section of the introduction explaining the structure of the rest of the document. This section of the document usually begins with a sentence such as “Section 2 describes the current state of the art in the chicken plucking industry. Section 3 then relates these developments to the growth of fox populations...”

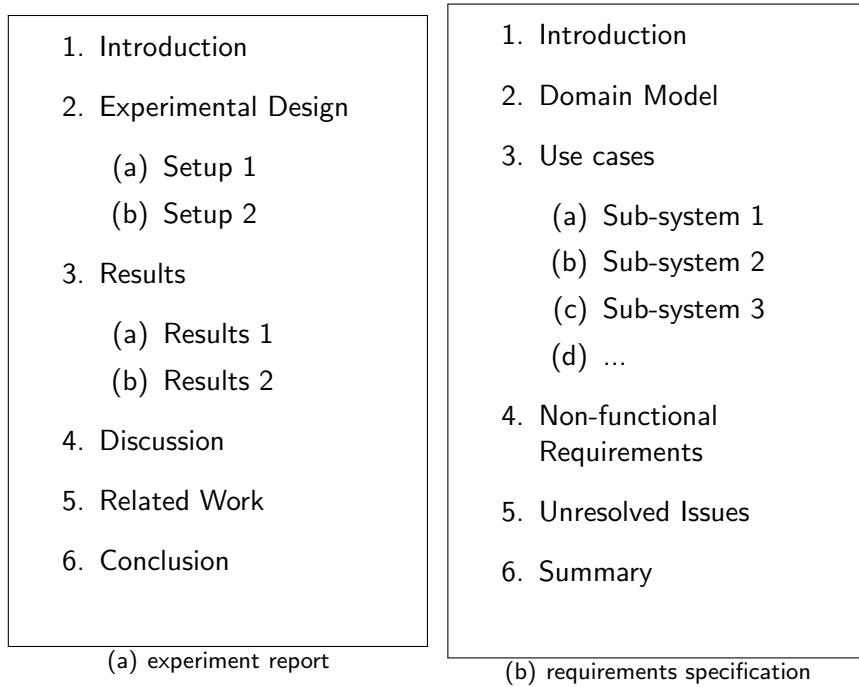


Figure 10.1: sample structures for two types of technical document

Main body (267)

Summary and or conclusions (268)

**The Main Body** consists of one or more high level sections, developing the detailed arguments and/or content of the paper. Sections should be used to represent significant breaks in the content of argument contained in the paper. A paper reporting empirical results might have sections on related work, experimental design and results. Each paragraph within a section should build on the prose contained in the previous one(s). Figure 10.1 illustrates example structures for two documents, a report of an experiment (Figure 10.1(a)) and a requirements specification (Figure 10.1(b)).

**The Conclusion** or summary, along with the abstract, may be the most read part of a paper. The section should:

- contain a summary of key points made in the paper;
- describe the main results of the paper;
- make a concluding argument; and/or
- state what the wider consequences or implications of the contents of the main sections are.

A conclusion may also describe the next steps to be undertaken in the work (sometimes this is in a separate section or sub-section called “Future Work”).

**References** Almost all work builds on that published by others. The references contain a list of sources used in the documentation as supporting evidence, or of related interest.

The appropriate use of the work of others in your own documents is very challenging. You *must* cite the work you have referred to in producing your own report. However, it is also in-appropriate to quote at length from a source without explanation or analysis, even if you provide a citation.

If you wish to refer to another author's work, you should provide a short summary in your own words which makes clear:

- what the relevant contents of the reference are; and
- *how it relates to your work.*

For example, to note a case study on open source software development, I might write:

Rosenberg has reported on the experiences of the Open Source Applications Foundation (OSAF) work on the Chandler project [Rosenberg, 2007].

Quotations taken from another source should only be used *if you wish to say something about the exact wording the author has used*. For example, I might write:

Sommerville defines Software Engineering as "the principles, methods, techniques and tools for the specification, development, management and evolution of software systems" [Sommerville, 2010]. Note the inclusion of management and evolution of software systems in the definition, reflecting a recognition that software systems may have very long 'shelf lives'.

The mark of good academic writing is how it *critically analyses and relates* relevant literature, not simply reports it.

There are numerous standard citation and reference styles, including the:

- Harvard style (author-year in parentheses);
- the American Psychological Association (APA) style; and
- the footnote/endnote styles.

The Harvard style is useful for an expert to quickly resolve which paper or document is being referred to. However, the endnote/footnote approaches tend to be less disruptive to the document's narrative.

Whichever standard you choose, the most important thing is to be consistent. It is good practice to use reference management systems to ensure consistency of referencing style. Examples are:

Refering to related work (269)

Using quotations (270)

Citation style (271)

- Endnote<sup>1</sup> for Microsoft's Word word processor; and
- BibTeX<sup>2</sup> for the LATEX type-setting system.

The natbib package for LATEX/BibTeX provides a flexible mechanism for incorporating references into documents as needed.

## Appendices (272)

**The Appendices** of a document contain the things that many of a document's readers don't *need* to know in order to understand the main sections. This may include:

- tabulation of raw measurement data;
- tool configuration details;
- a glossary;
- a list of acronyms;
- a bibliography of further reading . Note that this is different from a list of references;
- detailed mathematical proofs;
- test data; and/or
- a reference guide.

## 10.3 Quality Assurance and Documentation

### Why review documentation? (273)

Software documentation should be considered as a maintenance *process*, rather than the generation of a one-off document. This is because of:

- changes in the artifact to be documented (the requirements specification, software design, source code implementation, of the project team, for example);
- changes in the scope and level of detail in the document, as its purpose is better understood by its authors;
- the need to integrate style from multiple authors. As new content is added to a document by different authors, a periodic revision must take place to ensure a consistent style throughout the whole document; and
- the discovery and correction of defects.

---

<sup>1</sup><http://www.endnote.com>

<sup>2</sup><http://www.bibtex.org/>

Consequently, software documentation is a *continuous process* alongside other software development activities. The principle objective of the process is keeping the software documentation relevant to the software project, and therefore retaining its value. There are several strategies for maintaining the relevance of documentation.

Keeping  
documentation  
relevant (274)

- Don't produce more documentation than you can effectively maintain. Producing copious amounts of out-dated (and therefore un-informative) documentation is a waste of effort. Instead, prioritise what information needs to be formally documented.
- Integrate the cost of maintaining documentation into your project plan. There is a trade-off to be made between the extra cost of maintaining and communicating information without good documentation and the on-going cost of maintaining the information.
- Explicitly plan the review schedule for documentation to be within the life-cycle of change for the underlying artifact. For example:
  - user documentation should be updated as each new function is added to a system;
  - defect tracking documentation should be updated as bugs are assigned and corrected; and
  - software design documentation should be updated due to changes in the source code.
- Keep documentation close to source of change. This means that any discrepancies between the artifact and the associated documentation will be more evident and easier to detect and correct. In some cases, it is possible to completely unify the documentation and the target artifact. There are two approaches to this:
  - generate the documentation from the product, using documentation generators such as Doxygen<sup>3</sup> or JavaDoc<sup>4</sup> ; or
  - generate the product from the documentation, using *model driven development* techniques (See Section 25.7).

Notice that both approaches observe the principle 'don't repeat yourself'. The documentation and the underlying artifact are automatically changed together.

---

<sup>3</sup><http://www.doxygen.org/>

<sup>4</sup><http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html>

Revising  
documentation –  
stages (275)

You should expect to revise a document several times before it reaches an acceptable state for distribution outside of the project. The document should pass through several *drafts*:

- the *document plan* is an outline of what the structure and contents of the document will be. A document plan can be specified in varying levels of detail, from first order section headings, through to the contents of individual paragraphs. It may be possible to divide the writing of the document's different sections between several different authors based on the plan. Further more detailed section plans may be prepared by the individual authors;
- the *first draft* should contain all content so far identified that might be relevant to the document's topic. The first draft does not have to have a consistent style, or be particularly well written. The purpose is to get an understanding of the overall scope and direction of the document's prose. If the document has been written by several different authors it is good practice to appoint a single author to integrate the different parts. When the first draft is completed, it can be circulated for comment within the project team. At this stage, comments should focus on the structure and overall content of the document, rather than providing detailed comment on style, language, grammar or spelling issues;
- for second and subsequent drafts, changes can be made based on feedback from other members of the project team, the language and structure is revised and improved, irrelevant content is removed and new content that has been identified as missing is added. Each new draft of the document may be circulated for further comment. The changes made to the document are recorded in the revision log, along with the reason for change;
- a *release* of a document is published and/or circulated outside of the project team when the quality assurance team for the document are satisfied that it is acceptable. A release version of a document may not contain all of the meta-data recorded during its preparation. The revision schedule and log may be omitted for example.

In summary, don't be tempted to circulate a 'write-once' document outside of a project team. Take the time to get documentation right.

As discussed, documentation is subject to continual change. This can cause confusion between different versions of documents regarding what changes have been effected. Consequently, it is useful to maintain:

- a *revision schedule*, which records
  - when a document will be reviewed and why,

Revision schedules  
and logs (276)

- who will do the review and who will integrate changes; and
- a *revision log*, which records:
  - what changes were made to a document,
  - when they were made,
  - who made them and
  - the reason for the change.

A document should be read several times during a review to check for mistakes during a review. Some of the checks can automated, particularly for spelling and at least partially for grammar. Some tips for effective document review are:

- read each sentence in the document slowly and out loud. Saying the content of a document helps to make the content literal;
- consider what a *literal* interpretation of the sentence would mean;
- share the reviewing process amongst the team, so that each author reviews something they didn't write. This avoid mistakes being missed because the author skims content they think is correct; and
- experiment with collaborative writing, particularly for key sections of a document such as an abstract or the conclusions. Thinking through the expression of a phrase within a group allows the use of a much greater vocabulary.

A wiki can be an extremely powerful tool for collaborative software documentation. Typical features of a wiki include:

- change tracking, when what and by whom;
- document partition into pages;
- cross-referencing; and
- artifact integration (via plugins).

Most technical documentation is written in English. Consequently, there is a temptation to use metaphors, analogies, slang or colloquialisms which are native to English in order to improve understanding of the concepts being explained. Examples of such idioms include:

“The Internet is like a spider's web”

Is it really? So you can get tangled up in it? Does it have regular geometric patterns?

Reviewing documentation (277)

Using wikis for documentation (278)

Improving (simplifying) language (279)

"Eckythump!"

Crikey.

"The chap had pan fried".

He died.

"I am Hank Marvin!"

Quite hungry.

"The PC was just bog standard"

Do you mean the PC was equipped with a 1GHz x86 processor, 1GB of DDM2 RAM and a 30GB SATA hard disk?

However, many readers of technical documents do not read English as a first language and may not be familiar with the target of a particular idiom.

Hedging (280)

Similarly, it can be tempting to hedge, for example:

"It would seem to be case that the mouse ran into the hole"

can be simplified to:

"It is the case that the mouse ran into the hole."

which can also be simplified to:

"The mouse ran into the hole."

As can be seen, removing 'hedges' doesn't reduce the meaning of the sentence, but it does simplify the meaning and reduce the word count.

Asserting/assuming  
(281)

Finally, it can be tempting to propose how the reader should react to the contents of a document, for example:

"It is well known that..."

"It is clear from the graph that..."

"You will be able to see..."

"It is interesting that..."

"It is indisputable that..."

"It is obvious that..."

"It cannot be argued that..."

However, this may annoy a reader if the document doesn't cause the anticipated reaction. For example, stating that a concept is obvious may make a reader assume you the author thinks they are stupid if they find the concept challenging.

Consequently, simplicity is a good guideline for writing high quality technical documentation.

Some tips for simplifying language are:

- keep sentences to short statements. Break long sentences with many clauses into shorter statements that make the same argument. This makes the overall argument easier to follow because it can be read in smaller 'chunks';
- employ a consistent style, grammar and spelling. For example, it matters less whether the British or American spelling of 'through' ('thru') is used, than that it is used consistently. If an organization does have a policy of one spelling form over another, then use it;
- maintain a *bijection* relationship between terms and their definition. This means that every term you define has a only one meaning, which is not defined by any other term. It can be particularly tempting to use several terms interchangably for the same definition, particularly when:
  - several different authors have used different terms for the same definition; or (even worse)
  - different terms with different meanings are used interchangably.

For example, The terms "the Internet" and "the World Wide Web" are often used to refer to the same thing, despite their accepted definitions referring to different technologies. Documents containing the sentence "the terms X and Y will be used interchangably throughout this document" can be improved by ensuring the definition is consistent.

- define acronyms and use them consistently (and define them in a consistent way). For example "The World Wide Web (WWW) emerged in the early 1990s as a collection of technologies centered around the Hyper-Text Transfer Protocol (HTTP) and Hyper-Text Markup Language (HTML).";
- don't repeat points without a good reason. The document should be structured so that each statement, paragraph and section follows naturally from the information contained its immediate predecessor. Forward referencing ("See Section 4.5 for an explanation of this argument")is often an indication of poor structure in a document; and
- refer to and describe all figures, tables and other 'floating' artifacts included in the paper in the main text. In addition, all floating artifacts

Guidelines for  
simplifying language  
(282)

should appear as close as possible to where they are first referenced. Ideally, a figure should appear on the same page as where it is first referenced, or on the following page. The contents of each figure and table should be explained in the main text. If the content does not appear to need a description or reference, it probably isn't relevant to the topic.

Third person, passive voice (283) Technical documentation should only contain descriptions in the third person, passive voice. This means that:

- the narrator of a description is not present in the description, so the words 'I' and 'we' are not used; and
- the *subject* is the focus of a sentence rather than the *object*.

For example:

"I began the experiment by installing Windows XP on my laptop."

vs.

"The experiment was initiated by installation of Windows XP on the laptop."

One exception to this is that it is sometimes acceptable to use 'I' or 'we' in the introduction or conclusions to a document, particularly if the purpose of the document is to state the author's opinion on a technical matter (sometimes called a position paper). However, this approach does annoy some readers – generally speaking, only professors get away with it.

## Summary

Producing and maintaining high quality documentation takes time. Be prepared to automate as much of the repetitive work in document preparation as possible. Allow plenty of time for document review and revision.

## **Part II**

# **Requirements Engineering**



# Chapter 11

# Requirements Engineering

The problem of requirements risk was introduced in Chapter 3. In this Chapter, we will introduce the process of managing and gathering the requirements for a software system.

## Recommended Reading

PLACE HOLDER FOR sommerville10software

Recommended reading  
(284)

### 11.1 Requirements Risk

Recall the problem definition for the branch library system in Figure 3.6. Table 11.1 identifies some initial risks associated with the branch library system.

The political risks that have been identified may mean that the project's scope will need to be reduced, or the project be cancelled altogether. Notice also that the technical risks affecting the project are indicative of a *brownfield*

Branch library risks  
(285)

<b>political</b>	the university may cancel the entire branch system product, there may not be resources in departments to host branches
<b>technical</b>	the programming interface to the central library system is poorly understood and may not be compatible with the branch system platform
<b>skills</b>	the software development team do not have any experience of web application programming
<b>requirements</b>	different departments may want different models of a branch system, it is unclear how lending policy will be implemented

Table 11.1: risks to the branch library system

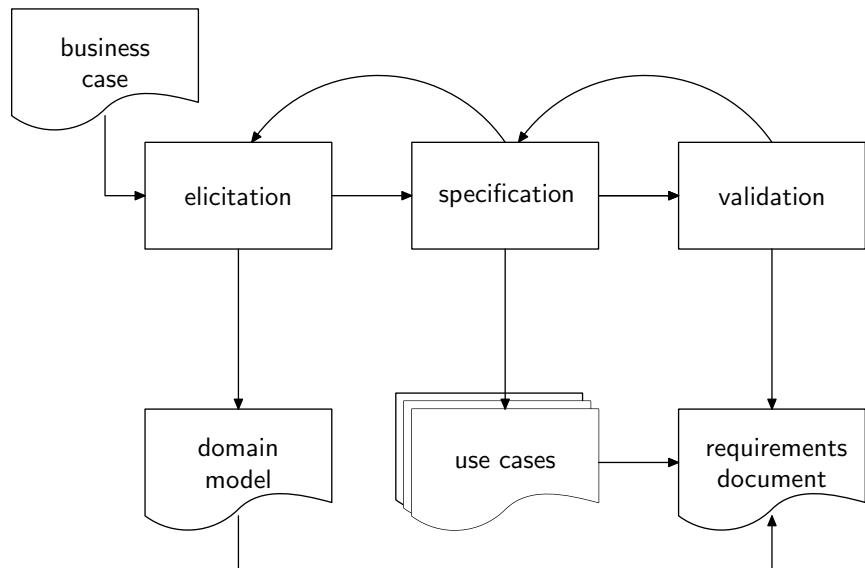


Figure 11.1: the requirements engineering process

(see Chapter 4) development - the project team will need to integrate the branch library system with some existing artifacts - library management system used by the library.

However, the requirements risks are the most pressing at the start of the project, since it isn't clear what form of branch library system the university, individual departments or the central library want. Let's now look at applying requirements engineering methods to address the requirements risks affecting the branch library system project.

## 11.2 The Requirements Process

Figure 11.1 illustrates the general process for eliciting and documenting software requirements. The input to the process is the business case, including the problem definition, an initial project scope and viability assessment.

More detailed requirements are then elicited from the customer and users using methods such as observations, interviews and prototypes. One output from requirements elicitation is a description of the environment in which the problem exists, called a *domain model*. The domain model explains the various artifacts found in the problem environment and the relationship between them. We will return to domain modelling in Chapter 14.

The other output from the elicitation phase is the collection of raw *user requirements* for the system, describing what the user needs to be able to do with the system. These requirements must be documented, organised and refined, producing a collection of use cases.

**F6.3.1 request books from central library**

A user shall be able to use the system to request that a book be transferred from the central library to the branch

Figure 11.2: user requirements for the request book use case

**SR6.3.1.1**

A member of the department (user) shall be able to log in to their account with the branch library.

**SR6.3.1.2**

A user shall be able to search for books in the central library from the branch library system

**SR6.3.1.2**

A user shall be able to select a book to be requested to be transferred to the branch library

Figure 11.3: example system requirements for the branch library

Often, the most natural way to organise requirements is by grouping related functions together. Figure 11.2 documents a functional requirement for the branch library system

User requirements, expressed as use cases, should be comprehensible to users. However, they are often too high level to be directly implemented by software engineers. *System requirements* are more detailed descriptions of what a user requirement, suitable for translation into a system design. Figure 11.3 documents several system requirements for the user requirement documented in Figure 11.2. Note that there is usually a one-many relationship between user and system requirements.

Finally, the domain model and use cases are combined into a requirements document that must be validated. Requirements validation is the process of obtaining agreement with the customer that the right system is being built.

When the system is verified with respect to the requirements document, it should be possible to determine from the system's design whether functional requirements have been satisfied (assuming the implementation is faithful to the design). However, non-functional requirements usually have to be measured from the real system as it is deployed.

Notice that the requirements engineering process depicted in Figure 11.1 is iterative. Outcomes from the validation phase lead to changes in the requirements documentation. The identification of omissions in the documentation may require further elicitation activities. In addition, the requirements themselves will evolve as the project progresses because:

- the problem domain changes;

User requirements  
(287)

From requirements to  
design (288)

Requirements  
evolution (289)

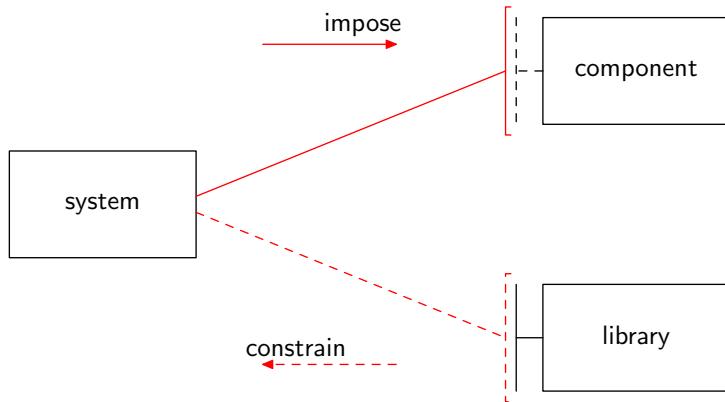


Figure 11.4: the relationship between requirements specification and design in a brownfield context

- the customer's own understanding of the problem increases; and
- progress on system design and implementation uncovers new risks and opportunities.

This evolution should be detected through continual review and evolution of the requirements documentation.

The discussion above implies a complicated relationship between requirements and design. In principle there is a clear distinction between a problem definition, which describes what is inadequate about the world; a requirements specification, which describes *what* a system should do to correct the problem; and the system design, which describes *how* it should do it.

This implies a top-down flow from requirements to design. At the highest level of abstraction, the system specification states what the system will do. The system design describes the high level arrangement of sub-systems and then imposes specifications on these system components. In turn these components are designed, imposing further specification on sub-components and so on.

However, the design of pre-existing brownfield software artifacts can also constrain what a new software system is able to do. Figure 11.4 illustrates the symbiotic relationship between the requirements and design of a software system.

At the system level, a system design imposes a specification on new components to be implemented for the system before they are designed. The dotted 'T' attached indicates the specification of the component's interface by the system design. However, the use of pre-existing *software libraries, frameworks or platforms* which provide some functionality for the system, imposes constraints on *what* the overall system is able to do, and thus also constrains its design. The dotted red line from the library to the system indicates the constraint on the system's design imposed by the library's specification.

## 11.3 Requirements Elicitation

The principle task of requirements elicitation is to elaborate with the customer and users what must be achieved in the project in order to resolve the original problem. This involves identifying the actors who will interact with the system and the goals they need to reach with it.

Requirements elicitation is also concerned with understanding the organisational context in which the system will be deployed, including both the formal activities and the *organisational culture*. Culture can be summarised as *the way we do things around here*, reflecting the collective outlook and values of the organisation.

Organisational culture can mask implicit assumptions about the system and what it will do. These need to be identified and documented as part of the elicitation process. In addition, there may be political reasons that the project has been started. Requirements elicitation can help uncover these and make them explicit.

In principal, the requirements elicitation process should not commit a project to particular design solutions. In practice, requirements inevitably constrain the final design because most projects begin with some pre-conceptions about what will be built to solve the problem. In addition, outlining design options may also help stakeholders think about the problem itself. A compromise is to start by exploring the problem space and consider many possible solutions.

There are numerous methods for eliciting requirements. Figure 11.5 illustrates some methods that will be discussed below. The methods are shown along an active-passive axis, indicating the extent to which the requirements engineer interacts with the problem domain.

Eliciting requirements  
(291)

### 11.3.1 Questionnaires

Questionnaires are the least interactive form of gathering requirements. Typically, the questionnaire is compiled and distributed to customers and users for completion. Results are then collected and aggregated. Questions can either offer a discrete number of finite responses, such as a Likert scale, or request a natural language unstructured response. Aggregating responses to structured questions often helps identify trends on concerns amongst respondents. However, structured questionnaires may also miss important details of concern to the respondents because there is no place for them to express themselves. A good questionnaire will normally mix structured and un-structured questions.

Requirements  
elicitation methods  
(292)



Figure 11.5: methods for requirements elicitation

A disadvantage of questionnaires is that poorly worded questions cannot be remedied after distribution.

### 11.3.2 Observations

Making observations is a good way to understand the organisational and cultural context of the problem. The documented work flows and processes of an organisation are often very different from how tasks are actually carried out. Work practices that have evolved over time to improve organisational efficiency may be disrupted by the introduction of a new system with a design that doesn't acknowledge their presence. The results of observations may be as much a surprise to the customer organisations's management as to the project team.

Observations (293) System actors who haven't been identified through other elicitation activities may be revealed by observing the 'real' work practices. Observations may also help to reveal assumptions that prospective users have made about the features of the proposed system.

One method of conducting observations is to *instrument* the work place in order to automatically gather data. Data sources can be video feeds, or logs from existing systems, for example.

A more active approach to observation is to *shadow* (follow around) a member of the organisation during their day to day tasks, building up an understanding of the organisational context from their perspective. In some situations, it may be appropriate to actually do the task normally undertaken by the member of the organisation. The observer takes copious notes, sketches, photos and even audio/video recordings. Some software engineers work with professional *ethnographers* to obtain detailed observations of work practices Viller and Sommerville [2000]. A disadvantage of observations is that it is sometimes challenging to draw general insights about the organisation from the results.

### 11.3.3 Interviews

An alternative to distributing questionnaires is to ask questions in person in an interview. Interview styles can range from completely structured to unstructured in a similar way to questionnaires. Figure 11.6 illustrates this range of options. The advantage of using interviews is that they permit interaction between the interviewer and the interviewee, and in particular, allow the interviewer to identify and correct mistakes in pre-defined questions.

Interviews (294) A good approach for interviews is to prepare a number of *relevant* questions to ask beforehand that cover the things you want to learn, but be prepared to depart from these if an interesting line of questioning emerges. Remember, you may be able to obtain answers to missed questions from other interviews, e.g. *those conduct by other software development teams on the same project*.

Conducting interviews (295) When conducting the interview, take some time at the start to explain your purpose and what you hope to learn. Use warm up questions to get the

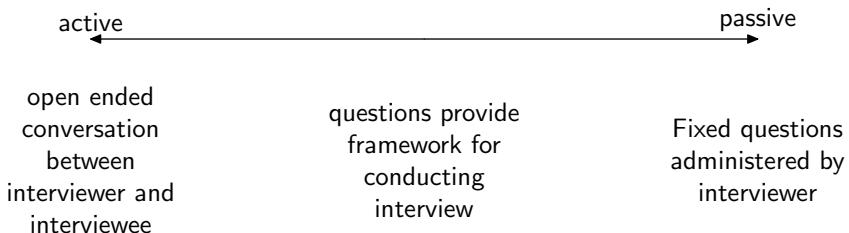


Figure 11.6: different approaches to structuring interviews

interviewee comfortable with the environment. Leave time at the end to debrief the interviewee. This involves presenting an initial summary of the interview, allowing the interviewee to identify any mis-understandings. Also, allow time for the interviewee to ask any questions of you - this may also help uncover issues that haven't been addressed.

Generally, the more documentation taken during the interview, the better. You can always discard useless information later. Designate a note taker for the interview who won't be involved in asking questions. You can also take an audio recording with the permission of the interviewee, but you shouldn't rely on it for documentation. After the interview, write up your notes immediately and summarise the key findings. Try to verify the findings with those from different interviews if possible.

#### 11.3.4 Walk throughs

The elicitation and validation of requirements from users is often easier if the discussion can be centered around a potential design solution for the problem to be addressed. This can be useful during the later stages of the process, when the broad specification of the system has been established, but the project team has discovered a more detailed aspect of the system that needs to be addressed.

A *scenario* is an example of the execution of a use case for the proposed system. The scenario documents the specific steps taken by a user in order to achieve a goal.

Walk throughs (296)

During a *walk through*, users are tasked with following a particular scenario in order to enact the use case. Alternatively, walk throughs can be videoed using actors and shown to users.

Typically walk-throughs can be conducted using a *mock-up* or a *prototype* of the proposed system. A mock-up is a crude user interface for a system with no underlying functionality. The mock-up may be little more than a collection of hand drawn sketches of system dialog boxes. Prototypes are similar, but often have some functionality, and in some cases may be developed into the final system.

Walk throughs provide a shared environment for users and the project team

### **Interview report**

Postgraduate students are to be treated separately from undergraduates, since the central library has different lending policies for them, and they can also be employed as staff within the department. The administrator suggests that a librarian will be employed by the department to manage the branch and any interactions with the central library.

The administrator also believes that the central library are concerned that books will go missing from the branch, because borrowers will not treat lending policies with the same respect.

Figure 11.7: a report of an interview with a department administrator

to investigate and evaluate the requirements for a system. Walking through a scenario helps users to understand the structure of the proposed activity and make changes if necessary. They also allow exceptional situations to be explored with respect to the system.

## **11.4 Documenting Requirements as Use Cases**

Once the requirements have been elicited from the users they need to be documented in a form that is understandable by users, customers and system developers. This eases the process of organising and analysing the requirements to ensure consistency and completeness. We will use *use case descriptions* and *use case diagrams* to document the requirements identified for the branch library system.

Figure 11.7 is an extract from a report of an interview with a departmental administrator. Using the new information, we can identify the different classes of *actors* who will interact with the system. Actor classes are sometimes called *roles* because they are partly defined by how they interact with the system and the use cases they may invoke. Figure 11.8 updates the collection of actors for the branch library. Actors are denoted by stick figures with a label. A short description of the actor also helps to document their role in relation to the system.

Documenting the actors helps to identify the scope of a system, which defines the scope of the project and identify constraints. Figure 11.9 shows the relationship between actors and the system. Things inside the system boundary must be created in order to support user activities. Use cases define the interface between the system and the actors.

The next step is to begin to document the activities that the actors will undertake with the system. Figure 11.10 illustrates a use case diagram. A use case diagram shows the relationships between several different use cases and the branch librarian actor. Use cases are shown as ovals with a label. As for actors, it is useful to accompany each use case with a short description, explaining its

Actors and roles (298)  
Identifying the system boundary (299)

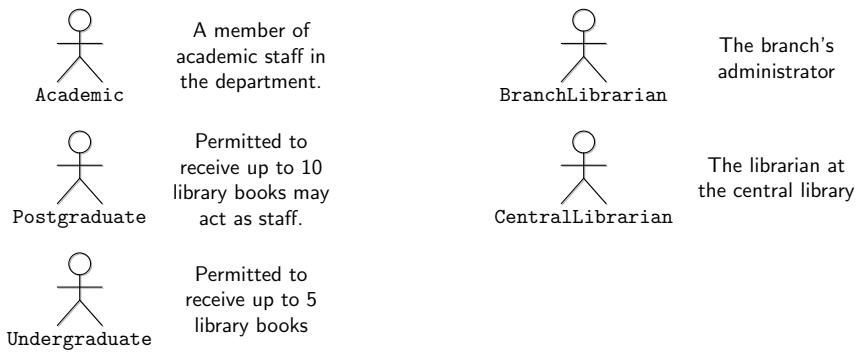


Figure 11.8: revised set of actors for the system

purpose. Sometimes the description can just be the requirement itself.

The figure shows the use cases identified so far for the branch library. There is a branch librarian, who will manage the lending and collection of books to and from members of the department. In addition, members of staff can use the system to request a book be transferred to the library. The branch librarian will use the system to process the request, passing it on to the central library. Notice that Staff has replaced the Academic actor, since we have discovered from the interview report in Figure 11.7 that non-academics may also be staff. We also need to give short descriptions for each of the use cases:

Use case diagrams (300)

**request book** staff borrowers are entitled to use the system to generate requests for books hosted in the central library are moved into their departmental branch

**process request** the book requests made by staff are reviewed by the branch librarian prior to initiating the request with the central library

**check book out** books in the branch library can be issued to borrowers. The

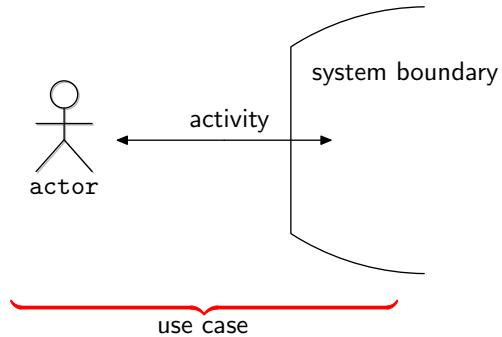


Figure 11.9: actors and the system boundary

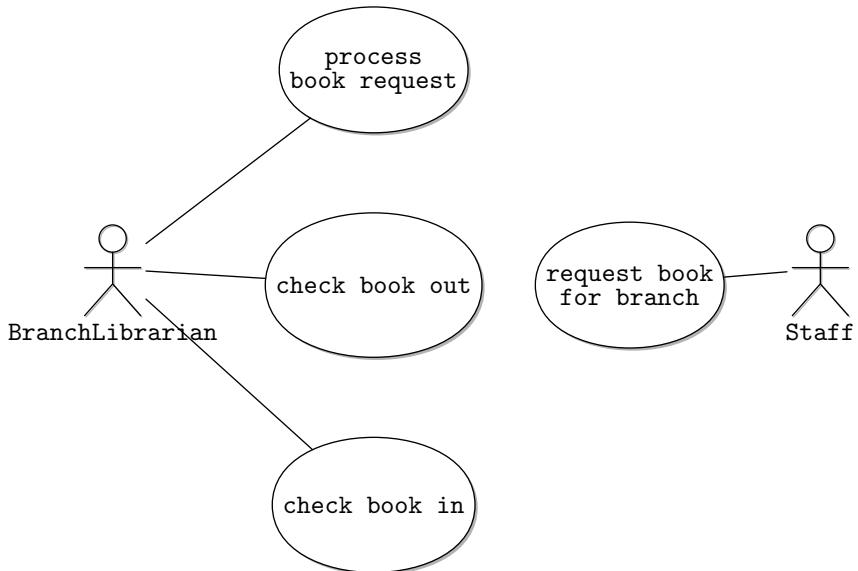


Figure 11.10: a use case diagram for some of the the branch library use cases

system needs to track which books have been issued to which borrowers and ensure that the central library's borrowing policy for that class of user has not been violated

**check book in** books that are borrowed can be checked back in to the branch library

External systems as actors (301)

From the problem description and requirements elicitation it is clear that the branch library system will need to interact with the central library system, so that requests for books to be transferred can be processed. We can show external systems as actors in the normal way on a use case diagram. Figure 11.11 includes the central library system as an actor that will interact with the branch library system for the process book request use case. Note that this again helps to define the scope of the system, by encapsulating central library system functionality in an actor. Actors do not have to be human users of the system, they are simply external entities in the domain that have some interaction with the target system.

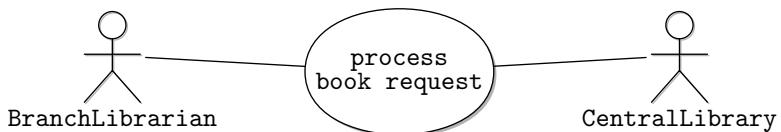


Figure 11.11: the central library system as an external actor on the branch library system

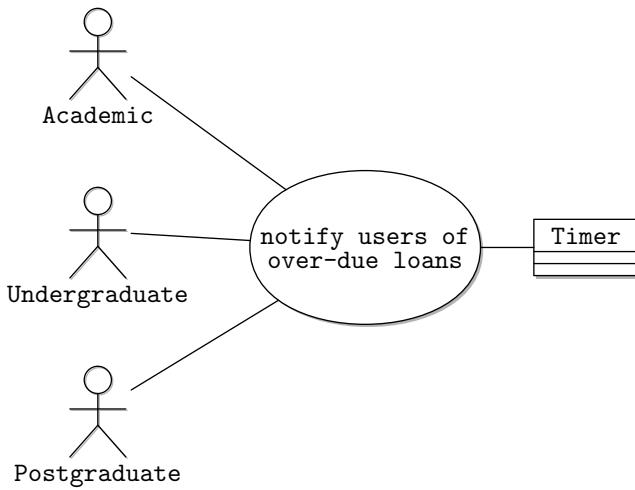


Figure 11.12: a timed notification use case

So far, we have only considered interactions with the system that are initiated by external actors. For these cases, the system is essentially passive, waiting for and responding to user input. However, there are some use cases that are better represented as being initiated by the system itself, even though they describe interactions with external users.

Recall from the interview report that the administrator was concerned that some users might treat branch library lending policies as more flexible than the central library. Although we can't expect the branch library system to alter human behaviour, we might be able to 'nudge' it a little. Figure 11.12 illustrates the use of an internal system timer to initiate a notification use case. All the borrower actors (staff, undergraduate, etc) are passive. When the timer is triggered the system checks all borrower accounts and issues notifications to any borrowers who are over-due in returning books.

Now that the functional requirements are starting to become clear, it is necessary to consider the non-functional requirements for the system. Non functional requirements refer to the *emergent* properties of a system that cannot be tested by the presence or absence of a feature. Non-functional requirements may be associated with one use case, several, or the entire system.

Non-functional requirements can be organised in a variety of ways. Sommerville uses a hierarchical taxonomy of *product*, *organisational* and *external* requirements. Table 11.2 lists different categories of non-functional requirements given by Mynatt, with some questions that can be used as a check list for a requirements document Mynatt [1990].

We can begin to identify some non-functional requirements associated with the branch library use cases we have identified, and classify them using the categories given in Table 11.2. Figure 11.13 illustrates the request book use case extended with some non functional requirements.

Timed activities (302)

Non-functional requirements (303)

Branch library non-functional requirements (304)

<b>usability</b>	What type of user will actually be using the system (novice, expert, frequent, casual)? Are alternative interfaces required? What are the desirable learning times for new users?
<b>error handling</b>	What is an acceptable error rate for a trained user? How should the system respond to input errors (recovery, report)? How should the system respond to extreme conditions?
<b>documentation</b>	Are user manuals, on-line help features, or tutorials required? What audience is to be addressed by each document?
<b>performance</b>	What metric will be used to measure throughput, response time etc?
<b>storage capacity</b> <b>compatability</b>	What are the current and future storage needs? Where are the interfaces with other systems? What data formats must be supported?
<b>availability</b>	What is the expected life time of the system? What periods of time does the system need to be available? How long can the organisation survive without the system? Are fall-back or fail-safe options necessary? What is the maximum acceptable downtime per day, month, year and frequency of outages? How often should the system data backed up? Who will be responsible for back up?
<b>environmental</b>	Are there any unusual operating conditions for the hardware?
<b>security</b>	Must access to the data or system be controlled? Is the data subject to the provisions of the Data Protection, or Freedom of Information Act?
<b>resources</b>	What is the budget for the system? Are there any personnel constraints?

Table 11.2: categories of non-functional requirements [Mynatt, 1990]

**F6.3.1 request books from central library**

A user shall be able to use the system to request that a book be transferred from the central library to the branch

**NF6.3.1.1**

Only staff members are permitted to request books

**type:** security

**NF6.3.1.2**

A response to the request shall be sent to the user within 3 working days

**type:** performance

Figure 11.13: the request book use case extended with non-functional requirements

## 11.5 Requirements Validation and Analysis

The process of checking that the requirements document is acceptable is called requirements validation. The task is to ensure that the document describes what the customer wants and is of a sufficient standard that the project team can begin to design the system. Table 11.3 summarises the key criteria for validating requirements.

The first criteria is to consider whether the requirement should be included in the system at all. If a requirement contributes no value to the system's business case, it should be discarded. One way of evaluating the importance of a requirement is to consider whether the customer would care if it wasn't present in the system, given that they are paying for it to be implemented. Also, the requirement should be traceable to the evidence gathered during requirements elicitation. If the requirement can't be justified, then it should be removed.

The second criteria is whether requirements can be realised within the projects current budget and resources. If a requirement cannot be realised, then the project scope may need to be adjusted. The realism of requirements reflects the fuzzy distinction between requirements and design. In principle, requirements should be specified in isolation, without concern for how they are to be satisfied. However, this risks specifying a system that cannot be imple-

Criteria for  
'appropriate'  
requirements (305)

**necessary** would the system be inadequate without the requirements?

**realistic** can the requirements be realised by a design?

**testable** can the requirements be observed in the deployed system?

**quality** are the requirements complete and comprehensible?

Table 11.3: criteria for evaluating requirements

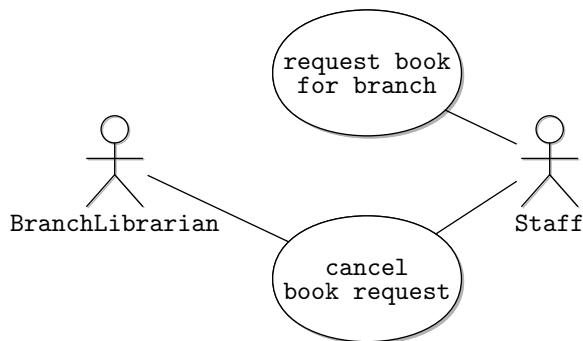


Figure 11.14: the new cancel book request use case

mented at all, due to the constraints that have been identified in the problem domain! In practice, a judgment needs to be made as to what belongs in the requirements specification, and what belongs in the design.

For the third criteria, if requirements are not testable, then it cannot be shown whether the system meets the requirements specification or not. A useful indicator of untestable requirements is the presence of phrases like 'user friendly', 'robust', or 'good' in the statement. Instead, consider the metrics you can apply to testing requirements.

The quality of the requirements documentation should also be reviewed during requirements validation. In particular, the completeness of the requirements can indicate the readiness of the project to proceed to the design and implementation activities. If parts of the system are still unspecified, more requirements elicitation may be required. The organisation and comprehensibility of the requirements specification should also be reviewed. Requirements that are poorly written may be interpreted differently by different customers and users.

After reviewing our existing use cases we decide to return to the request book use case. Writing use cases facilitates a dialogue between the customer and software team, and help us to decide what the system will and will not do. Use case descriptions can be added to and changed as our understanding of the problem improves.

After further discussion with the branch librarian, we discover that it becomes necessary to add a new use case for cancelling a book request. This will allow staff to correct mistaken requests before the book arrives in the branch library. The cancellation will generate a notification for the branch librarian. Figure 11.14 shows the addition of the new use case to the diagram. We may also need to add use cases for returning books to the central library.

We next proceed in reviewing some use cases developed by another member of the project team. We discover one use case (Figure 11.15) that reads better if it is satisfied by the actor rather than the system.

In the diagram, the database actor is responsible for storing the request information. This is often a good sign that the use case is prematurely committing

Refining the problem  
description (306)

Identifying premature  
design decisions (307)

<b>must have</b>	the system will not be successful without the feature
<b>should have</b>	this feature is highly desirable,
<b>could have</b>	this feature is desirable if everything else has been achieved
<b>would like to have</b>	this feature does not need to be implemented yet

Table 11.4: MoSCoW rules prioritisation scheme for requirements

the team to a design decision. In this case, the use case commits the system to storing requests in a database, which is clearly a design decision, because it is concerned with the internal implementation details of the branch library system. It may turn out that requests can be stored in a file, must be sent to a logging server or do not need to be retained on the system at all.

As a result of reviewing the existing requirements set, a number of new functional requirements have been identified. However, new features are not always desirable. When identifying potential new use cases, consideration should be given as to whether the new features are something the system should ‘naturally’ do, or could they be done by one of the existing actors.

Adding many new requirements may alter the project scope substantially, meaning that we have to re-consider the risk analysis and business case. All projects have a finite budget and duration, so it may not be possible to include all the desirable features in the system, or at least not in the first instance. Instead, the use cases identified for the project should be *prioritised* and an estimate established for implementing each of them.

Table 11.4 shows the MoSCoW rules, a typical scheme for prioritising use cases. Broadly, use cases should be ordered into a priority list from highest to least. Following the prioritisation, all of the ‘must have’ and most of the ‘should have’ use cases should be achievable within the allocated budget and duration. If this is not the case, it indicates that the project is not realistic and either the budget and scope are adjusted, or the project should be abandoned.

Prioritising use cases  
(308)

## Summary

The requirements engineering process is concerned with eliciting, documenting and validating requirements *about the system*. There are numerous methods for

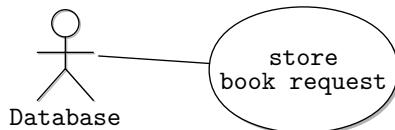


Figure 11.15: a pre-mature design decision expressed as a use case

eliciting requirements, depending on the type of information to be learned about the proposed system and the stage of the requirements elicitation process.

## 11.6 Exercises

1. Consider the problem description in Figure 11.16, and then complete the following tasks.

A museum requires a system that will enable museum staff to keep track of guided tours of the museum. When a party of visitors arrives, a staff member must be able to record the date, start time, details of the visitors and one of the pre-defined routes for the tour. The age of each visitor is used to calculate the total entrance fee to charge for the tour.

Also, a staff member must be able to assign a guide to the tour, from available museum tour guides (i.e., those not currently conducting a tour or performing some other duty); the assigned guide will be notified of the assignment.

During a tour, its guide must be able to record special incidents that might occur, such as a visitor becoming ill. Each such incident will be related to the tour on which it occurred and will include a description and a time. Incident reports must be communicated to the Museum's Safety Office.

Figure 11.16: museum problem description

From the description:

- (a) Identify the actors (including all stakeholders) in the system.
- (b) Write an initial collection of use cases to describe the functional requirements for the system.
- (c) Draw a use case diagram to illustrate the relationships between actors and use cases.
- (d) Write a list of non-functional requirements for the system. Organise the list by the categories given in Table 11.2 (you don't necessarily need requirements in every category).

Once you have an initial set of use cases, consider the following questions.

- (a) Are there any functional or non-functional requirements that you think have not been explicitly identified in the description?
- (b) How would test the non-functional requirements you have identified?
- (c) How do your requirements measure up to the criteria described in Section 11.5?



## Chapter 12

# Requirements Refinement and Planning

### Recommended Reading

PLACE HOLDER FOR bennett06object

#### 12.1 Recap - Branch Library

Recall from Chapter 11 that we are working on a project to develop a system to manage the movement of books in a branch library. We have established an initial problem description and business case and begun, on the basis of some requirements elicitation, to document the use cases and associated non-functional requirements.

We have now received the results of an interview with a Head of Department (HoD) from one of our colleagues. The department will host one of the branch libraries. Figure 12.1 is a partial transcript of the conversation with the HoD.

The results of the interview are not encouraging. The requirements engineer was hoping to use the interview to finalise some details regarding system infrastructure. However, the questions have un-covered a hidden assumption in the system specification. The use cases we have established so far were on the assumption that there would be a space allocated in the department for the branch and that a branch librarian would be appointed. According to the HoD, neither of these things will happen. We are going to have to re-think the features of the system to fit within the organizational constraints. The good news is that at least we have uncovered this assumption before proceeding to system design.

The HoD has also offered a small amount of administrative time for the system and has also suggested a different system specification. We need to develop a specification for a distributed branch library management system, rather than the centralised approach we had adopted thus far.

Interview with the head of department (313)

**Requirements Engineer(RE)** thanks for agreeing to talk to me... I need to just ask you a few questions about the systems infrastructure in the school.

**Head of Department (HoD)** okay,... but you might be better talking to the technicians.

**RE** right, I'll arrange that, but I would like to confirm a couple of basic things with you first.

**HoD** okay...

**RE** first, of all, I just need to check that there is network cabling in the branch library?

**HoD** well, there is networking throughout the department!

**RE** so... where exactly is the branch library going to be based,... within the department?

**HoD** I'm sorry, I don't understand...

**RE** where is the branch library going to be physically located, in the department, which room are you planning to allocate to it?

**HoD** oh, I see. There must be some mis-understanding!... space is very tight here at the moment. I don't have any room to allocate for a branch library... it wouldn't be attended anyway.

**RE** oh, so where will the branch librarian be based?

**HoD** who?

**RE** we thought you were planning to appoint a librarian to manage the branch.

**HoD** no, there's no money for that! Who told you that?

**RE** are you planning to make branch librarian an administrative duty for one of your academic members of staff?

**HoD** I don't think that would work at all. A branch librarian would probably be a full time task. I could probably ask the existing library representative to take on a slightly larger administration load, but nothing more than say an hour a week.

**RE** so who do you envisage managing the books in the branch then?

**HoD** I'd assumed you were planning some sort of distributed system, with books held and lent out by individual academics.

**RE** Right, well thank you very much for your time - I've got a lot to talk to the team about!

Figure 12.1: interview with the Head of Department (HoD)

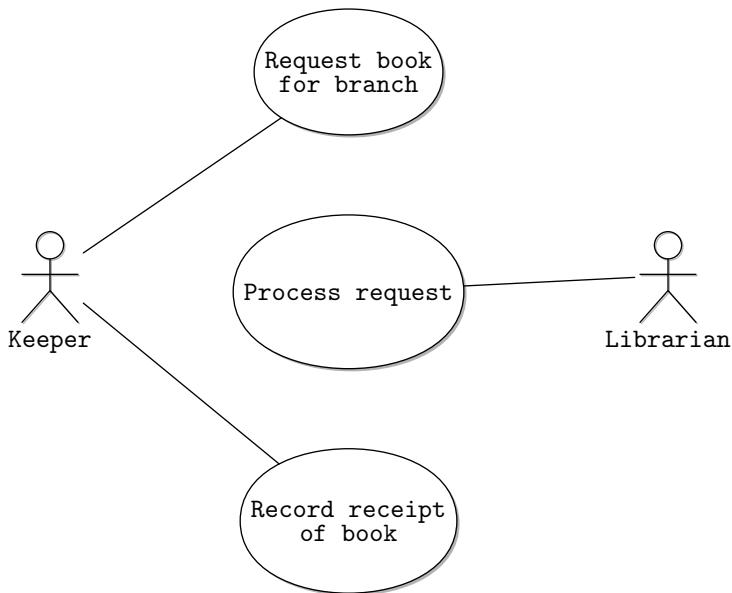


Figure 12.2: the use cases related to transferring a book to the branch

Although a distributed branch library system may be more complicated to implement, it does provide users with some useful features. In the distributed branch library, books are held by members of staff throughout the department. Each academic requests books to be transferred to the 'branch', and the member of staff is then responsible for the book while it is in the department. The member of staff is able to lend the books they hold out directly to students or other members of staff.

Recall the discussion in Section 11.2 on the relationship between requirements and design, and the constraints imposed by brownfield development. Notice how the new organisational design that we have adopted has *constrained* the specification of the branch library system. In this context, the branch library system is just one component in the larger organisational system consisting of the University, the Central Library and the constituent departments. Thankfully, we haven't begun to design the branch library system itself yet, so the specification is still open to alteration.

The new organisational model means that we are going to have to revise the use cases and actors for the branch library system. Figure 12.2 shows the revised use case diagram for transferring a book to the branch from the central library.

The diagram shows three use cases (request book **for** branch, process request and record receipt of book) and two actors (a Keeper and the Librarian in the central library), much of which is new. The Keeper actor represents a member of the department who is entitled to hold books on behalf of the branch. There may be many keepers appointed within the department because the books will be distributed rather than held in a central location. As

Revised use cases  
(314)

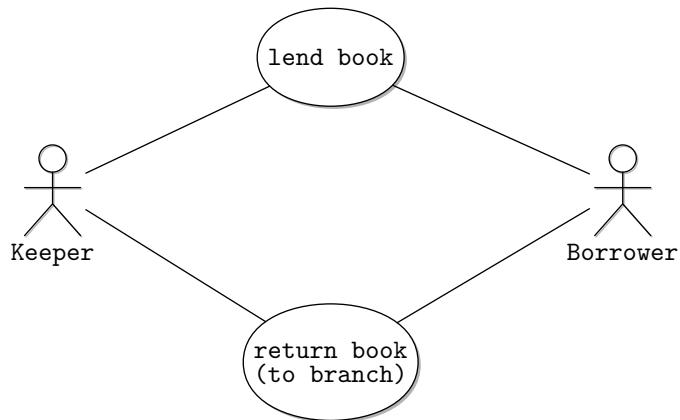


Figure 12.3: use cases for lending books in the branch

before, a librarian in the central library will approve and process requests. There is also a new use case for a Keeper to record receipt of a book. We have adopted a looser model of book management, so we need to record on the system when a book has been received into the possession of a keeper.

Having arranged for books to be delivered to the branch library, we need to model the lending of books by keepers to borrowers. Figure 12.3 shows the use case diagram with the collection of use cases for lending books in the branch.

Next, we need to consider how members of staff get appointed as keepers. Recall from the interview with the Head of Department that a small amount of administrative work load can be allocated to a member of departmental staff. Figure 12.4 illustrates the use case diagram containing the two use cases for adding and removing keeper accounts from the system.

The altered use cases also means we should revisit the non-functional requirements for the system. In particular, the decision to opt for a distributed

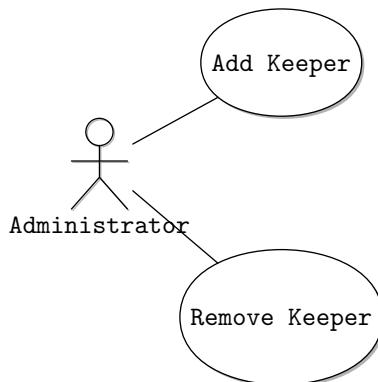


Figure 12.4: use cases for keeper administration

NF1 The system must interact with the Central Library's database of books.

NF2 The system must interact with the Central Library's database of users. Access control must be implemented with reference to this database.

Figure 12.5: non-functional requirements for the branch library system

system means that the management application will need to be available at a larger number of locations than previously considered, because each registered keeper will need to access the system from their office. This has implications in a large number of categories of non-functional requirements, in particular, user interface and security requirements. Figure 12.5 documents some system wide non-functional requirements.

Notice how the system-wide specification of non-functional security requirements implies new functional requirements. In particular we will need to implement use cases for logging on to and logging off from user accounts to ensure proper authentication and access control.

Non-functional requirements (317)

## 12.2 Planning

Now that a revised system specification has begun to take shape, we need to begin planning the elaboration and construction phases. Requirements elicitation has identified a large number of use cases for the system (see Figure 12.6), not all of which are going to be possible to implement within the time scale for the first iteration of the project. In addition, some of the use cases have been added late and are not fully documented yet.

All use cases (318)

We can use the MoSCoW rules method to decide which use cases to prioritise. Figure 12.7 illustrates the use cases grouped according to priority. Notice that the must have category includes only the essential use cases for the system. Use cases for searching for books, viewing status and so on have been placed in the should have category, because the system can function without them, just

- |   |  |   |
|---|--|---|
| <ul style="list-style-type: none"><li>● login</li><li>● logout</li><li>● search borrower</li><li>● search book</li><li>● view status</li><li>● add keeper</li><li>● remove keeper</li></ul> | <ul style="list-style-type: none"><li>● request book</li><li>● process request</li><li>● record receipt</li><li>● lend book</li><li>● request return (to branch)</li><li>● record returned (to branch)</li></ul> | <ul style="list-style-type: none"><li>● request return (to central)</li><li>● record sent</li><li>● record returned (to central)</li><li>● send reminder</li><li>● apply fine</li><li>● record fine payment</li></ul> |
|---|--|---|

Figure 12.6: all use cases identified for the branch library

<b>Must have:</b> login logout request book process request record receipt lend book record returned (to branch)	<b>Should have:</b> search borrower search book view status add keeper remove keeper
<b>Would like to have:</b> send reminder apply fine record fine payment cancel request	<b>Could have:</b> request return (to branch) request return (to central) record sent record returned (to central)

Figure 12.7: moscow rules categorisation of the use cases

Planning (319)

not at all well. The use cases for returning books to the central library have been relegated to the could have category. Books will need to be transferred back at some point, but this is likely to be required once the body of books in the branch is established.

With the use cases organised by priority we can begin to plan the overall project as a series of iterations. We will aim to implement the must have and should have use cases during the first iteration.

## 12.3 Describing Use Cases

The next step in the elaboration phase is to begin to add more detail to the use cases so that they are in a state to be translated into a system design. To do this, we need to document the sequence of events that occur when a use case is executed and how variations and exceptions are managed. In addition, we need to document the state the system must be in before a use case can be started, and the state it is in after the use case has finished. These are called the pre and post conditions for the use case.

The method we will adopt for this process will be to develop a number of scenarios for each use case. A scenario uses specific information (perhaps fictional) to add detail to a use case. Figure 12.8 illustrates the *primary* scenario for the request book use case. A primary use case describes the situation when all the 'normal steps' take place, with no unusual exceptions. Primary scenarios are sometimes referred to as happy day scenarios, because everything goes right in them.

Primary scenario  
(320)

John logs into his account with the branch system. He is a registered keeper and wants to request that a copy of the book 'Software engineering' by Sommerville is transferred into his posession in the branch library. He selects the 'request book for branch' option from the main menu, and enters 'Sommerville\*' into the search field. The query return several possible options, including 12 copies of the book on software engineering (in the 6th, 7th and 8th edition) and another on requirements engineering. John chooses the software engineering book (9th ed). Finally, he clicks the button to confirm the request and logs off.

Figure 12.8: the primary scenario for the request book use case

John logs into his account with the branch system. He is a registered keeper and wants to request that a copy of the book 'How to lie with statistics' is transferred into his posession in the branch library. He selects the 'request book for branch' option from the main menu, and enters '\*Statistics\*' into the search field. The query return a 4652 options, unfortunately, none of which are the book he is looking for. John logs off the system.

(a)

John logs into his account with the branch system. He is a registered keeper and wants to request that a copy of the book 'How to lie with statistics' is transferred into his posession in the branch library. He selects the 'request book for branch' option from the main menu. The system reports that the connection to the central library is unavailable. John logs off the system.

(b)

Figure 12.9: an alternative scenarios for the request book use case

Figure 12.9(a) illustrates an *alternative* scenario for the same use case. The scenario shows what happens when the book requested is not found in the central library. Alternative scenarios illustrate how exceptions or unusual events are handled. In principle, all possible alternative scenarios should be documented along with the use case. In practice, many trivial variations can be combined. Figure 12.9(b) illustrates a second alternative use case, this time documenting how the system should respond when the connection to the central library is unavailable.

Scenarios are useful for establishing the details of a use case with users and customers. However, the information contained is too specific to be directly transferred into a system design. Instead, the structure of the use case needs to be abstracted from the specific examples. The first task is to enumerate the steps of the scenario. Figure 12.10 shows the steps taken by the participant in the primary scenario in Figure 12.8

Alternative scenarios  
(321)

Enumerating the steps  
(322)

1. John logs into his account with the branch system by entering his username 'johnd' and password 'mypetcat84'.
2. He selects the 'request book for branch' option from the main menu
3. and enters 'Sommerville\*' into the search field.
4. John chooses the software engineering book (9th ed).
5. he clicks the button to confirm the request and
6. John logs off.

Figure 12.10: the steps of the primary scenario for the request book use case

Abstracting from the sample data (323)

Next, the specific steps taken in the scenario are abstracted to a general procedure detailing the use case. Figure 12.11 shows the steps of the use case in *pseudo code*, a semi-formal language for describing activities. Using pseudo code provides a briefer description of the steps taken, suitable for input to the design process.

Alternatives (324)

The alternative scenarios now need to be integrated into the use case description. To do this, it will be necessary to describe the conditions under which alternative routes are taken through the use case. Pseudo code can be extended to include descriptions of alternatives and repetitive actions. Figure 12.12 illustrates the use of conditions in pseudo code to integrate alternatives.

Repetition (325)

Repeated tasks can also be denoted in pseudo code using `foreach` and `while` loops, based on a condition. Figure 12.13 illustrates the use of repetition for describing the request book use case.

Complete Example (326)

Finally, all the separate parts of the use case can be integrated into a single description. Figure 12.14 illustrates the complete pseudo-code description of the request book use case.

```

login
select 'request book for branch' option
search for books
select books
confirm request
log off

```

Figure 12.11: the steps of the request book use case as pseudo code

```

if <condition> do
  <step 1.1>
  <step 1.2>
  ...
else if <condition> do
  <step 2.1>
else
  <step 2.2>
endif

if connection available
  search
  ...
endif

if book found do
  if not book in branch
    select books
    confirm request
  else if on loan
    request return
  endif
endif

```

Figure 12.12: denoting alternatives in pseudo code

```

foreach <items> do
  <step 1.1>
  <step 1.2>
  <...>
done

while <condition> do
  <step 1.1>
  <step 1.2>
  <...>
done

foreach books found do
  select book
done
confirm request

while book not found
  search
done

```

Figure 12.13: denoting repetition in pseudo code

```

login
select request book
if connection available
  while not books found
    search
  done
  foreach book found do
    select book
  done
  confirm request
endif

```

Figure 12.14: the request book use case description in pseudo-code

## 12.4 Activity Diagrams

Pseudo code is an expressive notation for conveying system activities for later software design. However, for complex use cases, it can be difficult to gain an understanding of the flow of work from a pseudo-code description. In addition, the notation is less suited for explaining specifications to customers and users.

Activity diagrams are a graphical notation for illustrating activities. All the constructs available in pseudo-code are available in an activity diagram. In addition, activity diagrams can represent activities that occur in parallel, which are difficult to denote in pseudo-code. Activity diagrams (as of UML 2.0) are based on the concept of petri-nets, which can have multiple *flows* passing through them simultaneously. This can be a powerful formalism for modelling complex concurrent activities. However, we will be using activity diagrams to represent relatively simple purposes.

Figure 12.15 illustrates the notation for activity diagrams. Activities are represented as labelled lozenges with flows between activities indicated by arrows.

Decisions are indicated by diamonds accompanied by the appropriate condition (the *guard*) on each of the out-bound flows. Some variants of the activity diagram notation place the guard on the diamond itself, if it is obvious which flow to follow based on the condition.

Activity flows can diverge or merge in parallel at forks and joins respectively. When a fork is reached, all proceeding flows can be followed immediately (allowing any connected activities to also begin). When a join is reached, all preceding activities must terminate before the flow can proceed beyond the join.

Activities can be decomposed into finer levels of detail if necessary. Activity diagrams have entry (begin) and final (flow and activity) points, indicating the interface between the diagram and activities represented at higher levels of abstraction. A flow final node indicates the end of a single flow. Multiple flows may be active in a diagram as a result of one or more fork nodes being reached. The activity final node indicates the end of all flows in the activity.

Figure 12.16(a) shows the same request book activity described in pseudo

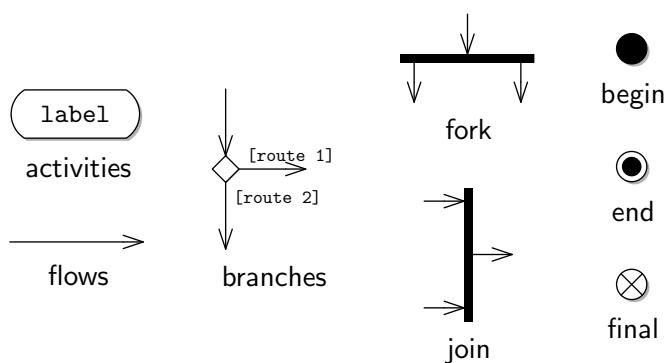
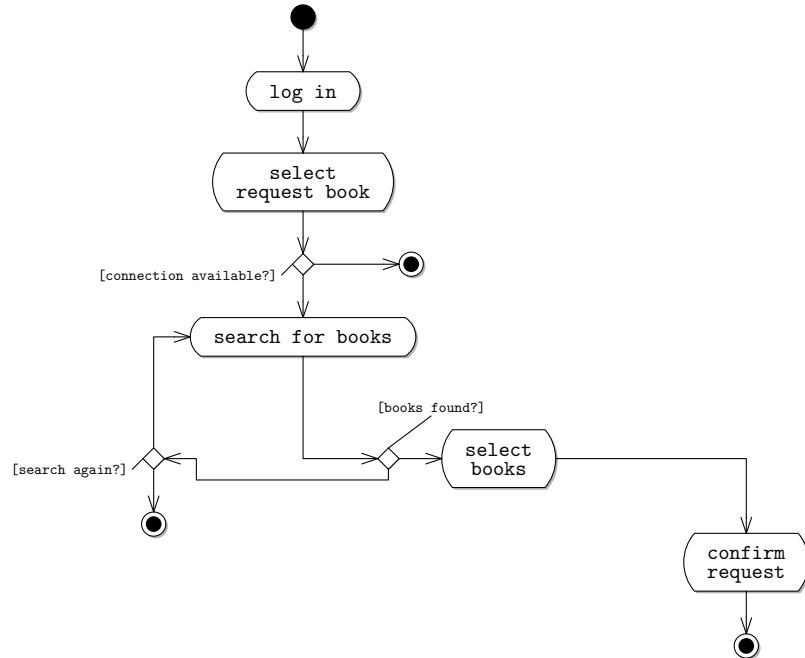
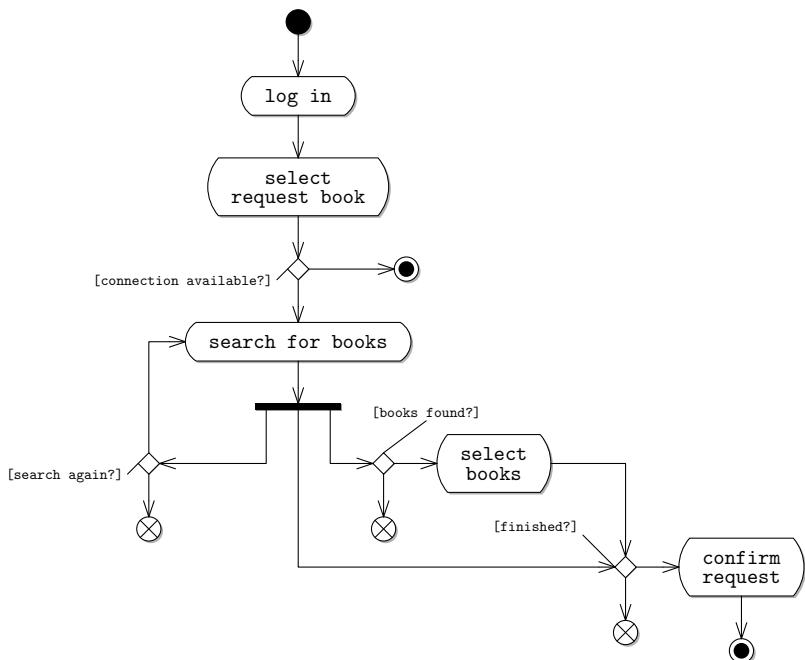


Figure 12.15: notation for activity diagrams



(a) pseudo code-equivalent



(b) using parallelism

Figure 12.16: the request book for branch use case illustrated as an activity diagram

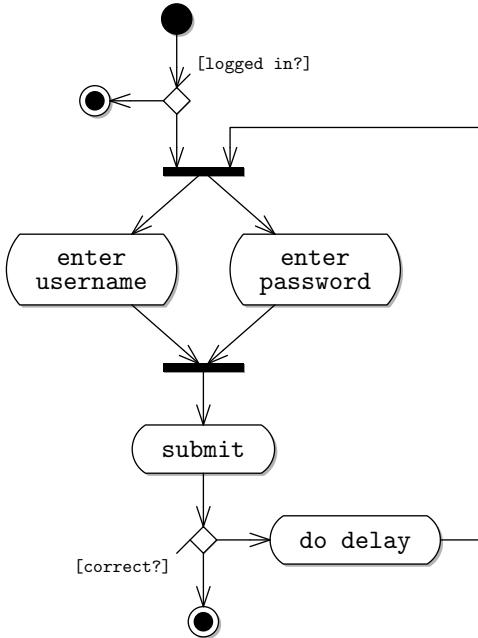


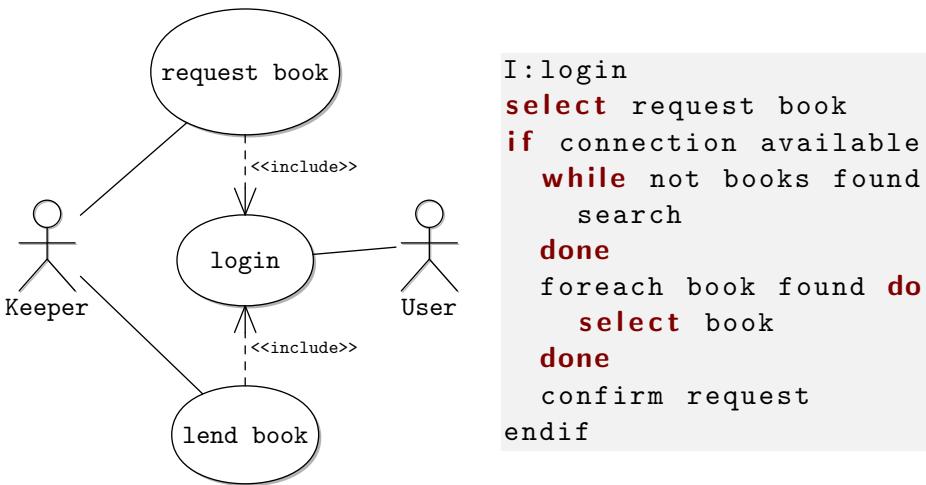
Figure 12.17: the login use case

code in Figure 12.14 as an activity diagram, using the notation in Figure 12.15. A user logs in and (if a connection is available) searches for books to request for transfer to the library. If the search has produced the desired results, the books can be selected and the request confirmed.

On reviewing the flow of activities with a user, it becomes apparent that the arrangement is rather awkward. In particular, the work flow requires that a user either get their search exactly right before selecting the books to be requested, or alternatively submit separate requests for each search.

Figure 12.16(b) refines the activity diagram by using a parallel arrangement of searching for and selecting books. Once a user has searched for books, they may do any of three things in parallel: search again, select some books or confirm the request. Note that activity final nodes have been replaced with flow final nodes for each of the decisions following the fork. These allow the relevant flow to be terminated, without halting the entire activity. We could also extend the activity diagram further to allow a user to cancel a request.

We can also add further refinements by drawing activity diagrams that describe a single activity at a finer level of detail. Figure 12.17 shows the login activity used in the request book use case description as a full activity diagram in its own right. Notice that we have used fork and join nodes to exploit parallelism again in representing the login use case. A user must enter both a user name and a password for authentication, but the order in which they do this is not restricted.



```

I : login
select request book
if connection available
  while not books found
    search
  done
  foreach book found do
    select book
  done
  confirm request
endif

```

Figure 12.18: including login in the request use case

## 12.5 Advanced Features

There are some additional features of use case diagrams that can be used to simplify requirements documentation. The UML was designed to be extensible, which means we can define special classes of entities and relationships with particular semantics. Extensions in UML are defined using *stereotypes*, which can be denoted using a label surrounded by guillemots, «like this». Stereotypes can also be denoted by changing the appearance of the artifact. Actors for example are special types of object classes.

Stereotypes in UML  
(330)

There are two standard stereotypes for use case diagrams. Sometimes it is convenient to show a commonly occurring activity as a use case which is part of a number of other use cases. Included use cases are usually complete and can be used in isolation (a user may login and do nothing else for example).

We have already seen how the login use case is represented as an atomic activity in the request book use case activity diagram in Figure 12.16. We can also represent this on a use case diagram with an «include» relationship, as shown in Figure 12.18. In the diagram, login is included in both request book and lend book use cases. The figure also shows the equivalent modifications to the pseudo-code description of request book to indicate where the login use case is included, the login statement is preceded by an 'I:'.

Including use cases  
(331)

Alternatively, it may be the case that a use case description has become complicated by too much detail. To manage this situation, some details of the use case can be encapsulated in an extending use case, denoted by an «extend» relationship. Typically, an extending use case is an encapsulation of a fragment of activity that isn't 'naturally' part of the main use case (e.g. for managing unusual situations or alternatives) and cannot be used as a use case in its own right.

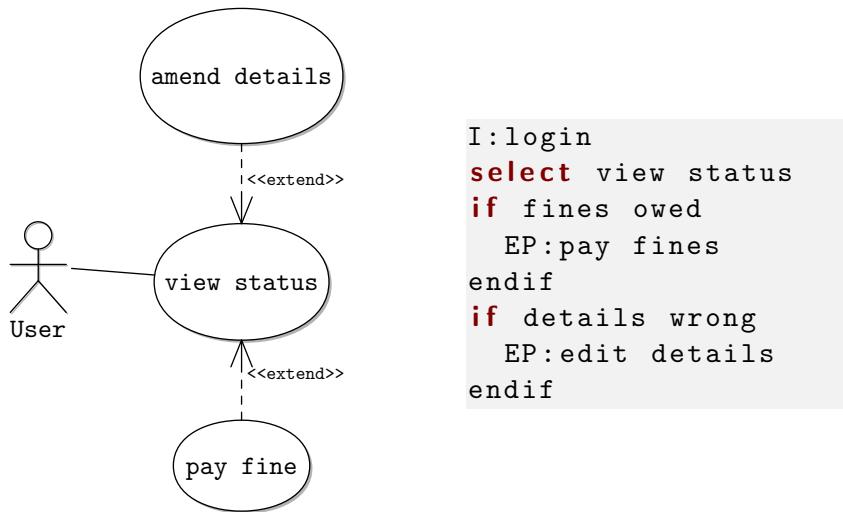


Figure 12.19: extending the view status use case

Figure 12.19 illustrates the view status use case extended by two further use cases, pay fine and amend details. The figure also illustrates the equivalent notation in pseudo-code, the extending statements are preceded by an 'EP:':

Note the distinction between included and extending use cases. Included use cases can be thought of as 'common' utility activities that occur frequently as part of other use cases. Extension use cases are fragments of activity that add extra activity to a 'main' use case. This difference is emphasised by the stereotype relationships. Included relationships are drawn from the including use case to the included. Extended relationships are drawn from the extended use case to the extending.

We can now see the recursive relationship between use case and activity diagrams. Use case diagrams are used to denote the functional specification of a system and the interface between the system and external actors. Use case diagrams also show where one use case is used as part of another. Activity diagrams are used to explain the structure of activities within each use case. Where a use case is used as part of another, it is represented as an activity.

We now need to revise the descriptions of the actors on the system to take account of the changes made to the use cases. The change to a distributed system has meant that the former role of branch librarian has been split between a branch administrator and the book keepers who request and lend books. Keepers are academics who can also borrow books like other student borrowers. This can make use cases quite complicated with many different actor classes associated with each of the use cases. Figure 12.20(a) illustrates this problem with the lend book use case with all of the relevant actors present. It isn't clear from the diagram whether all the different actors play different roles in the use case, or whether they all (or some of them) play the same role.

Extending use cases  
(332)

Actors and Inheritance  
(333)

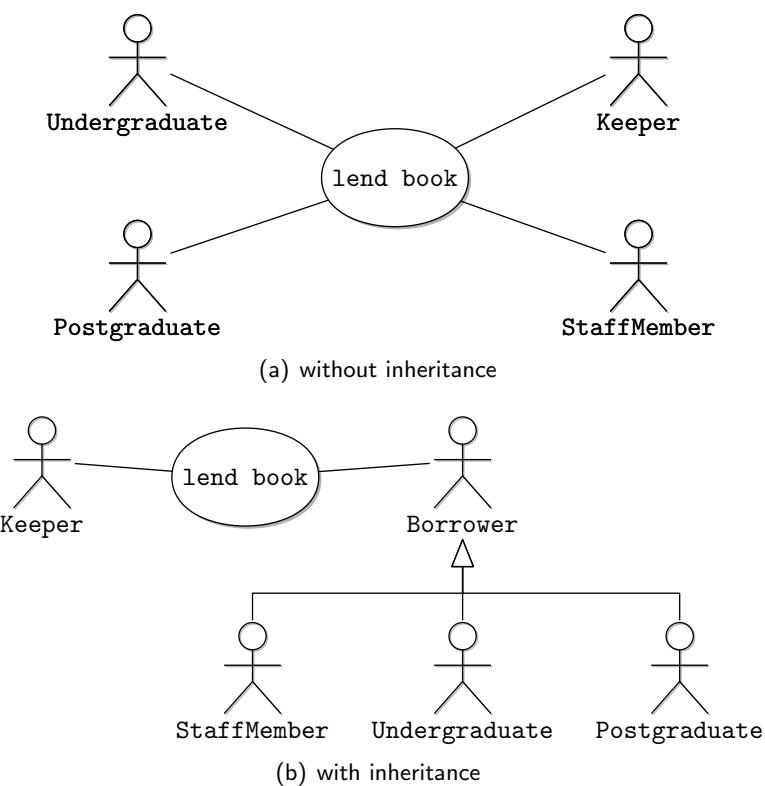


Figure 12.20: the lend book use case and actor inheritance

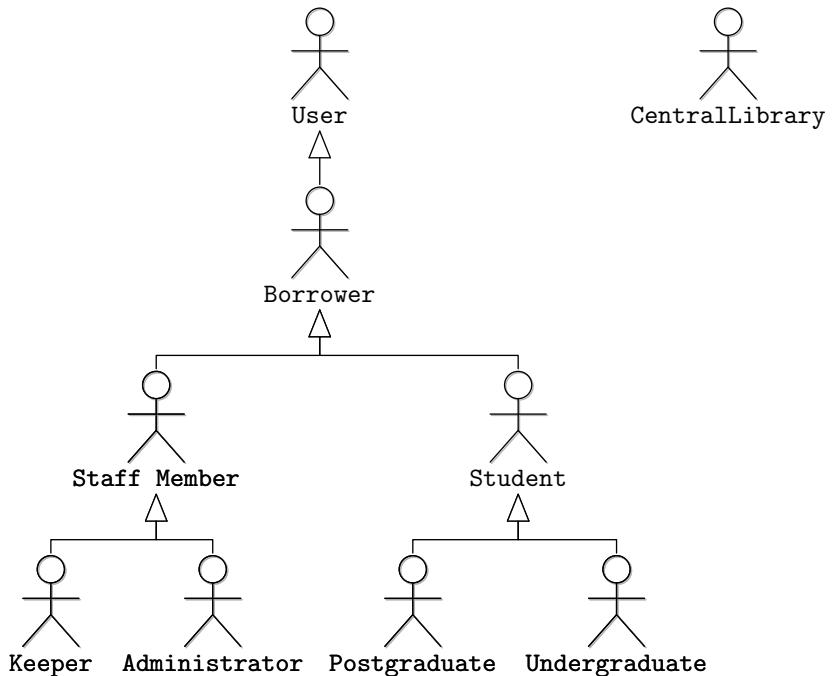


Figure 12.21: revised actors for the branch library system

To simplify the diagram, we can use *inheritance* between different actor classes. We can see now that actors are just a special object classes, so the 'is a' rule of inheritance applies to them in the normal way. Figure 12.20(a) shows the lend book use case with inheritance among the actors. The diagram introduces a borrower class. Now, the only two actors associated with the lend book use case are the book's keeper and the prospective borrower. All other actor roles who are able to borrow books are sub-classes of the borrower.

In practice, it can be useful to show inheritance between actors on a separate diagram from the use cases. Figure 12.21 shows the revised actors for the branch library system with the inheritance relationships between them. Notice we have decided to include an extra student role in the hierarchy - it may be that some aspects of the undergraduate and postgraduate roles are the same.

Revised actors (334)

## 12.6 Completing Use Case Descriptions

We have now covered the different approaches to describing the details of use cases, and need to fill in the final few details.

Use cases must be associated with both pre- and post-conditions. *Pre-conditions* describe the state that the *system* (note the emphasis) must be in before a use case is executed. Pre-conditions describe the constraints on the system state that are not handled by the internal logic of the use case. Pre

<b>rationale:</b> Keepers need to be able to request books to be transferred into the branch library.
<b>conditions:</b>
<b>pre</b> none
<b>post</b> a request for the transfer of the requested books is stored for processing; or
<b>post</b> no request is stored if not confirmed

Figure 12.22: pre and post conditions for the request book and lend book use cases

conditions must be checked and satisfied by the system before the use case is started.

Post conditions describe the state that the *system* must be in once a use case has finished. There may be several alternative states for the system, depending on the possible outcomes from a use case.

Finally, a *rationale* is needed which gives the justification for the use case. The rationale supports traceability checking for use cases to the original source documentation for the requirements.

Figure 12.22 illustrates the pre- and post-conditions and rationale for the request book and lend book use cases. Note that there are two possible post conditions because the use case can terminate with or without a request being confirmed. Also note that there is no pre-condition, indicating that the use case can be executed from any system state. This is acceptable, because we have documented which users can invoke the use case separately (keepers), and have included the login process as part of the use case.

We now have all the elements necessary to document the system requirements as use cases, so we now need to collate the information into a single requirements document. Appendix A contains a *LATEX* template for recording all the information related to use cases discussed above.

Each use case for a system should be described using a separate template. Use cases are best grouped together into related functions, accompanied by a use case diagram to describe the relationships between use cases and actors. Where appropriate, use cases can be repeated on different use case diagrams. For example, the login use case may need to appear on each diagram where it is included in another use case.

Figure 12.23 illustrates the overall structure for a use case document. The document consists of a problem definition; a description of system actors; a set of use case diagrams illustrating relationships between related use cases and actors; and a set of use case descriptions, developed using the template described above. In addition, the requirements document should have an appendix containing the

Conditions and  
rationale (335)

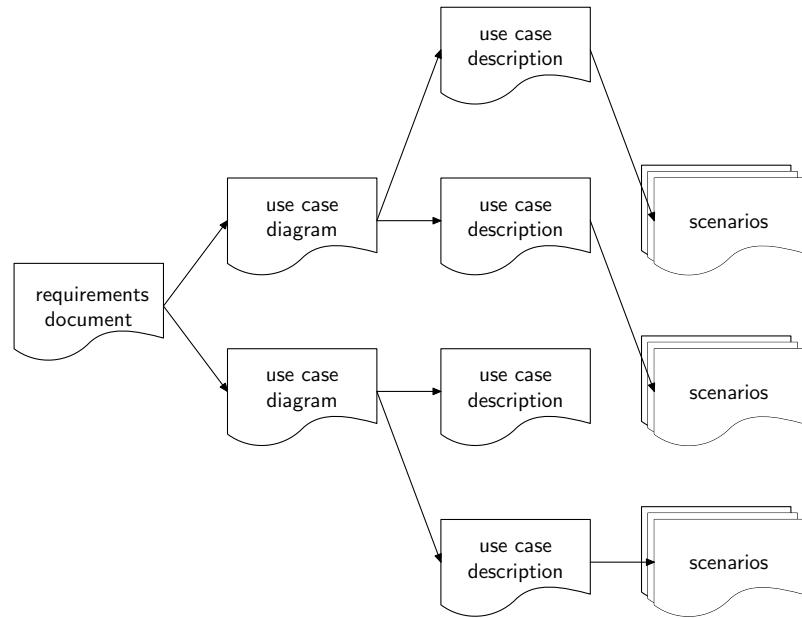


Figure 12.23: the component parts of a use case document

Use case document  
(336)

key scenarios for the use cases.

An example use case document for the branch library system is available on Moodle.

## Summary

Use case descriptions document a single use case of a system. This can be done via a variety of notations, including natural language, pseudo code and activity diagrams. Use case diagrams show the relationships between use cases.

## 12.7 Exercises

1. Investigate following CASE modelling tools:
  - StarUML (Level M) [http://staruml.sourceforge.net/docs/user-guide\(en\)/toc.html](http://staruml.sourceforge.net/docs/user-guide(en)/toc.html)
  - Umbrello (Level 3) <http://docs.kde.org/stable/en/kdesdk/umbrello/index.html>
2. Recall the scenario in Figure 11.16 and then complete the followings tasks.
  - (a) Elaborate primary and alternative scenarios for each of the use cases identified in the previous tutorial.
  - (b) Draw an activity diagram for each of the use cases you have identified.
  - (c) Indicate any omissions or ambiguities in the description and state any assumptions you have made.

Develop the detail of your use cases using the  $\text{\LaTeX}$  (Level 3) or Word (Level M) template available in appendix A and on Moodle. In particular, for each use case give a rationale and describe some scenarios to illustrate the possible variations and exceptions, the pre and post-conditions, extension and inclusion points and any non-functional requirements.



## Chapter 13

# Object Oriented Analysis with UML

### Recommended Reading

PLACE HOLDER FOR lethbridge05object

Recommended reading  
(339)

### 13.1 Introduction - The Unified Modelling Language

A system is a mental construct, denoting some aspect of the world with state and/or behaviour. A system is distinguished from its environment by a system boundary. We may be also be able to discern the purpose of systems we identify, for example, the purpose of a car engine is to provide kinetic electric power to the car's wheels.

A *model* is an abstraction of some aspect or perspective of a system. Models can serve two purposes:

- analytical models are used to describe and better understand the properties of existing systems; and
- constructive models are used to describe proposed systems, in order to predict their future properties. In some cases it may also be possible to derive an implementation directly from a sufficiently detailed constructive model.

Reasons for modelling  
(340)

We can also distinguish between a modelling notation and a method:

- modelling *methods* are descriptions of systematic procedures that can be followed to construct a particular type of model of a system; and
- modelling *notations* are standardised syntax used to represent the results of applying modelling methods.

Categorising  
modelling notations  
and methods (341)

Clearly, there is not a tidy distinction between the two purposes of modelling. Many modelling notations can be used for both analytical and constructive purposes. For example, an analytical models may be extended to show how a new system would improve the performance of systems in an existing environment.

Modelling notations can be categorised in a variety of ways:

- by system aspect
  - structural
  - behavioural
- or both; and
- by format
  - informal
  - graphical
  - algebraic.

A little history (342)

Graphical notations are particularly common in software engineering, since they are useful in managing the complexity of software problems. The *Unified Modelling Language* (UML) is a graphical modelling notation derived from three object oriented modelling methods:

- object oriented design [Booch, 1982];
- the object modelling technique, including state and class diagrams [Rumbaugh et al., 1991]; and
- the ObjectOry method including component and use case diagrams, as well what became the phases in RUP [Kruchten, 1997].

Graphical modelling  
with UML (343)

UML is said to be a *semi-formal* notation, since it has a well defined syntax and semantics, with some informal extensions. Some of the features of UML are:

- a notation used in many modelling methods;
- well defined semantics, implemented in different syntactic flavours;
- extension mechanisms via *stereotypes*; and
- an algebraic formal constraint language (OCL) supporting design by contract.

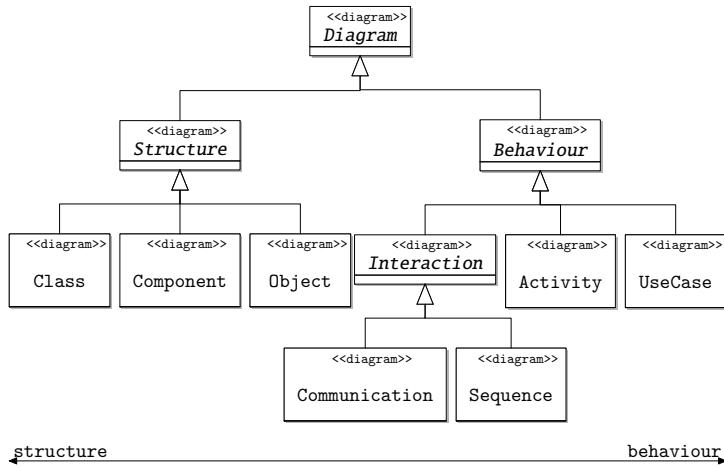


Figure 13.1: UML diagram type hierarchy

Figure 13.1 illustrates some of the different types of diagram provided in the UML. UML diagram types are broadly categorised as either structural or behavioural, with behavioural diagrams having a further sub-category of interaction diagrams.

However, the figure illustrates that the different diagram types exist on a spectrum from purely structural to purely behavioural, with those types in the middle exhibiting elements of both. We will principally focus on the use of class diagrams in this chapter, for the purpose of developing object oriented models of the problem domain for a software application. Other diagram types may be introduced throughout these notes, as UML can be used for describing many different software engineering concepts. For example, Chapter 21 shows how to use activity diagrams to represent the flow of events in a system to develop test cases and the use of component diagrams to represent a system connected to a test harness.

## 13.2 Developing Object Oriented Domain Models

A key challenge in software engineering is the to understand and accurately represent the problem domain and then translate the resulting requirements into a system design.

During problem *domain analysis*, activities are undertaken to:

Stages of modelling  
(345)

- identify key domain artifacts, their attributes and relationships that must be represented in object oriented system;
- allocate behavioural responsibilities to domain entities; and
- provide initial inputs to software design processes.

During design, the domain model is translated into a software design model which:

- documents the architectural decisions made by the software engineer; and
- identifies the sub-systems and components that constitute the overall system.

Domain modelling process (346)

The domain modelling process is approximately:

1. Analyse the available problem domain documentation
2. Develop an initial collection of 'high priority' classes with attributes
3. Identify relationships between classes
4. allocate types to attributes and extend the set of classes with new types as necessary
5. Simplify model by applying appropriate generalisations and abstractions
6. Identify and allocate further responsibilities and behaviour where appropriate using behavioural diagrams
7. Review domain model with stakeholders and identify missing or incorrect aspects
8. repeat!

In this chapter, we will examine methods for constructing domain models in the UML *class diagram* notation. We will be using the sports league management system problem description, introduced in Figure 2.1 in Chapter 2 as a motivating example.

### 13.3 Class Diagrams

In UML, a class is denoted as a rectangle, which can be divided into compartments by horizontal separators, as shown in Figure 13.2. Every class must be given a type label, shown in the top rectangle of each stack. In addition, classes may have attributes and/or operations listed in the lower two compartments. If both attributes and operations are present then the attributes are given in the middle compartment.

The first step in producing a domain model using the UML class diagram notation is to elicit and categorise the key artifacts from the problem description. Table 13.1 summarises the key artifact types to be identified, and what an analyst should look for in the problem description in each case.

Problem domain analysis (347)

A first pass collection of classes for the sportster application is shown in Figure 13.2. Notice that the attributes and operations of the sportster classes

artifact	look for
classes	the nouns (things) in the environment
attributes	simple characteristics of nouns that can be represented as a primitive or 'utility' type
relationships	ownership statements (has, part of, contains) between subject and object nouns, complex characteristics of already identified nouns
operations	descriptions of behaviour, things that can be done <i>to</i> class instances, descriptions of responsibilities

Table 13.1: identifying problem domain artefacts

have not been assigned types in Figure 13.2. In addition, many of the classes have not been assigned any behaviour at this stage. The priority during domain analysis is to identify the classes, attributes and relationships. Type information and more operations will be added as the class diagram is refined, during later stages of the design process.

Figure 13.3 illustrates how classes are annotated with type information for their attributes and operations. Types are placed at the end of the member identifier, after a : colon. Operation parameters can also be added to a class inside the operation signature's parentheses, and typed in the same way as attributes. It is common to omit attributes and operation types during early domain modelling, while the overall class structure is established. However, we will need to use this feature later on.

Attributes and operations also need to be annotated with their *visibility* as the class diagram is gradually refined. Visibility is particularly important during

Denoting classes in UML (348)

Denoting classes in UML (349)

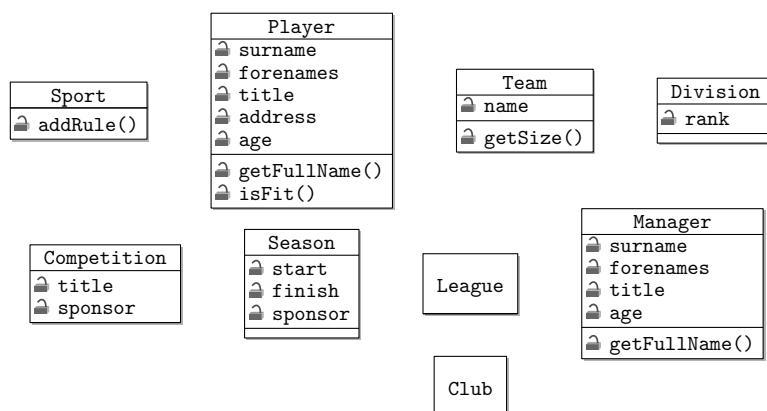


Figure 13.2: UML classes with un-typed attributes

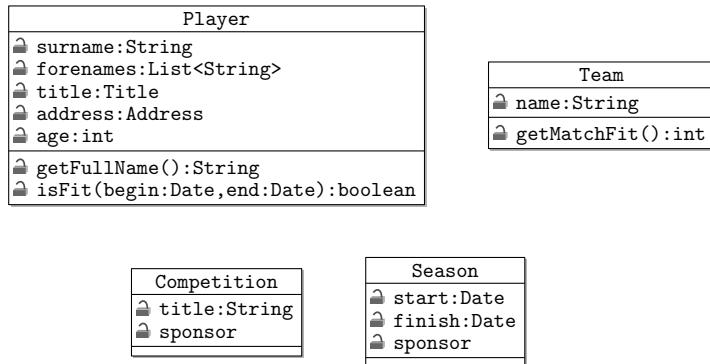


Figure 13.3: extending classes with types

Member visibility  
modifiers (350)

Denoting association  
constraints between  
classes (351)

Labelling associations  
(352)

the later design stages of an object oriented development, when considerations for encapsulation, information hiding and abstraction become important. Table 13.2 summarises the different visibility modifiers for class members.

### 13.3.1 Associations and Constraints

Associations are used to denote relationships between classes. Associations are solid lines drawn between classes. An undecorated association can be read as a 'has a' relationship. Figure 13.4 illustrates some example associations between sportster classes.

Constraints on associations are placed at the ends of a relationship denote how many *instances* of the classes may be a member of the relation at run time. Figure 13.4 illustrates the one-to-one, one-to-many and many-many constraint combinations on relationships. Constraints are read at the far end of the relationship from the class of interest. For example, "a sports division has many seasons".

Constraints can be used to indicate ranges of legal instance numbers in a relation. By default, an undecorated relation means one-to-one. Placing a \* on a relation means "one to many" and an explicit range of values can be denoted by x..y, which means any number between x and y, inclusive.

Associations can be labelled with identifiers. These can then be used to provide specific interpretation of the association, instead of a default 'has a' or 'has some'. Figure 13.5 illustrates different uses of labels for the sportster class.

Associations are implemented as attributes in an object oriented program-

	<b>public</b>	globally visible to all other classes
	<b>protected</b>	visible to child classes
	<b>private</b>	only visible internally in the owning class

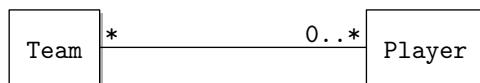
Table 13.2: member visibility modifiers



a player has an address, each address is associated with one player  
(one-to-one)



a division has 1 or more seasons and a season is for one league  
(one-to-many)

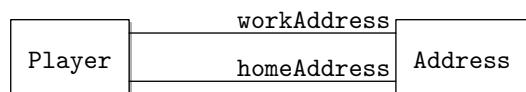


a team consists of zero or more players and a player can play for  
many different teams (many-to-many)

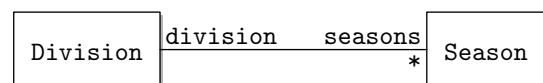


a team has at most one manager and a manager can manage no,  
one or two teams

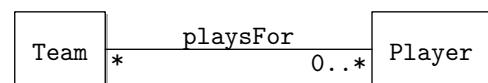
Figure 13.4: associations and constraints in UML



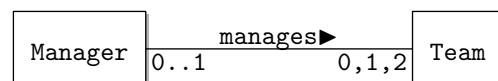
a player has a home address and a work address



a division has 1 or more seasons and a season is for one division  
(one-to-many)



a team consists of zero or more players and a player can play for  
many different teams (many-to-many)



a team has at most one manager and a manager can manage no,  
one or two teams

Figure 13.5: associations and constraints in UML

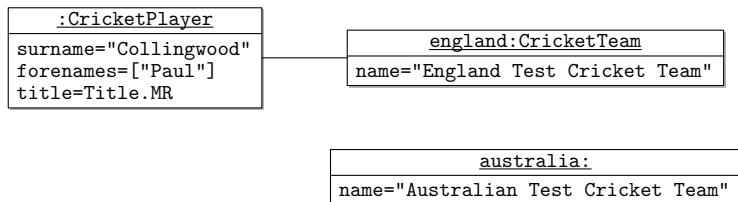


Figure 13.6: denoting class instances in UML

ming language.

### 13.3.2 Validating Class Associations

Take care when you are constructing class diagrams to read the associations denoted between classes carefully to make sure they make sense for the problem domain.

UML *instance* or *object* diagrams are useful for denoting the state of object oriented systems during execution (remember that an object oriented program's state is the objects, their attribute values and the associations between them). Instance diagrams are used for similar purposes as component diagrams, except they tend to be used to denote smaller scales of detail. Figure 13.6 illustrates the instance diagram notation.

An instance is denoted as rectangle and can be labelled with a name, a type, both or neither. In the diagram there is one instance of type CricketPlayer, one instance of type CricketTeam labelled england and another instance labelled australia without a type (although we can infer it is probably a CricketTeam). The state of instance attributes can also be denoted in slots, for example, the surname of the CricketPlayer instance is "*Collingwood*".

Associations between instances can also be shown on an instance diagram. For example, Collingwood's membership of the England cricket team is shown as an association in Figure 13.6. If necessary, instance associations can be labelled with the class associations they instantiate. Notice that unlike class diagrams, instance diagram associations are one-to-one between concrete instances.

Instance diagrams can be useful in determining the multiplicity of relationships between classes on a class diagram. Figure 13.7 illustrates a sketched instance diagram for the state of the sportster framework for managing cricket teams. From the diagram we can infer that:

- there is a many-many relationship between cricket players and teams (a team may play for more than one team and a team consists of many players); and
- there is a one-many relationship between teams and sides (each side is drawn from a particular team).

Instance diagrams  
(353)

Using instance diagrams to understand class relationships (354)

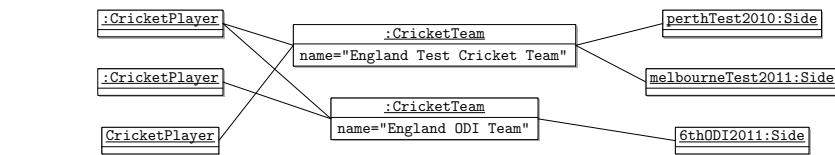


Figure 13.7: using UML instance diagrams to understand class relationships

Be wary of taking too many design decisions before the full class structure has been identified, as this can mean lots of re-working as the architecture of the system to be built is better understood. In particular, allocating types to attributes too early may prevent necessary further classes being identified in the domain model. In general, attributes are simple data items such as boolean values, numbers or strings. More complex data types should be represented as classes in their own right. Determining the difference between a primitive and complex attribute can be difficult early in the modelling process.

For example, we may choose during design to represent a player's address as a separate class or as a String, depending on the needs of the application and the complexity of the data to be stored. Figure 13.8 illustrates these options.

In addition, 'utility' classes such as dates, colours, coordinates or locales are usually best represented as attributes because their internal design is not of interest for the design of the system under consideration.

Deciding between relations and attributes (355)

Association classes (356)

### 13.3.3 Decorating Associations

There are several decorations that can be added to associations which have formal meaning in UML.

Sometimes, it is convenient to place a class between a many-to-many relationship between two other classes. This is of particular use when class diagrams are used to model relational databases, which don't support many-to-many relationships directly, but instead use *link tables*. Figure 13.9 illustrates how a Match association class can be inserted into the many-to-many relationship between Season and Team.

The association class can either be portrayed as having many-one relations

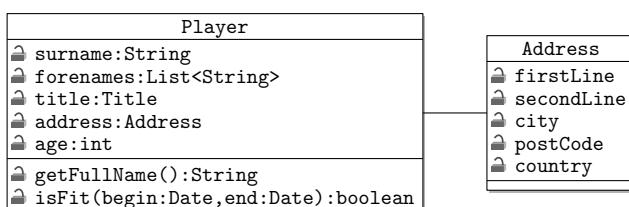
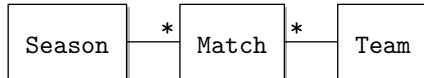


Figure 13.8: refining attribute types during software design



each team will play a number of matches during a season and a number of matches will be played in a season

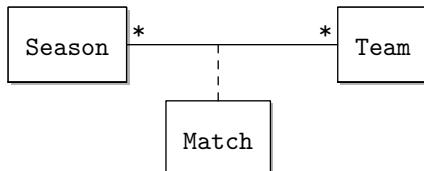


Figure 13.9: using association classes for many-many associations

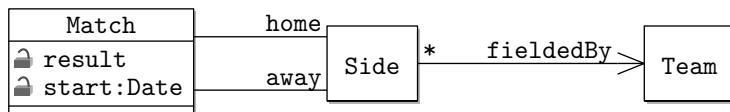


Figure 13.10: directed associations in UML

to the two main classes in the relationship, or alternatively by drawing a dashed link from the association class to the association. The two portrayals of the relationship between Season, Team and Match are equivalent.

By default, associations are bi-directional, meaning that in an implementation of the model both classes involved in the association relationship will have an attribute of the other type. Sometimes it is unnecessary for one class to have access to the other class, and indeed, it is desirable to remove unneeded attributes in order to reduce coupling. *Directed* associations indicate that the relationship can only be navigated one way, i.e. only class 'knows' of the existence of the other.

Figure 13.10 illustrates how to denote directed associations on UML class diagrams. The *fieldedBy* association can be navigated from the Side class to the Team class, but not vice-versa. This means it is possible to discover which team a side was drawn from, but that a team is not used to track the sides that it fielded.

Sometimes, classes maintain associations to themselves, which means that objects of the same class will have references to each other. This can be useful when recursive algorithms must be defined on a class, for example. Figure 13.11 illustrates how to denote a reflexive relationship on a class diagram.

The Season class has a reflexive relation showing that each instance of the class is preceded by previous season and succeeded by a next one. This is a similar data structure to a doubly linked list.

Similarly, a Player class can be denoted with a reflexive relationship indi-

Directed associations  
(357)

Reflexive associations  
(358)

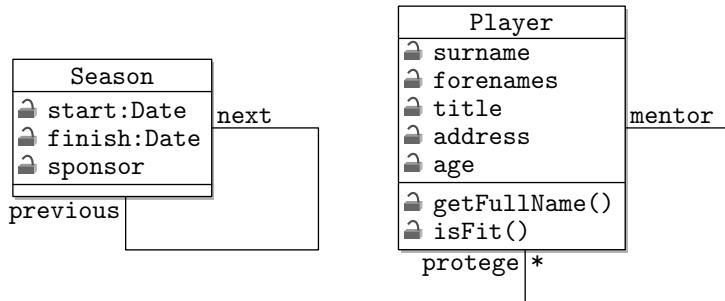


Figure 13.11: reflexive associations in UML

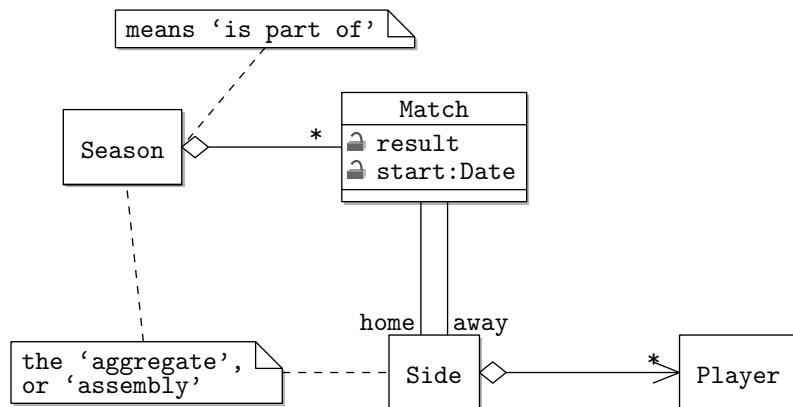


Figure 13.12: aggregation decorations of associations in UML

cating that a player may have a number of proteges who will learn from their experience, but that each player can have only one mentor. This illustrates that multiplicities can be applied to reflexive associations in the normal way.

Some classes serve as groupings of instances of other classes. Each of the constituent classes can be said to be a 'part of' the grouping class. This role can be made explicit on a class diagram by using an aggregation association. Figure 13.12 illustrates the use of the aggregation decoration.

The class which acts as a grouping of the instances of the constituent class is referred to as the 'aggregate' or 'assembly'. The 'is part of' relationship is indicated by decorating the association with a white diamond head at the aggregate end. In the figure, a Match is denoted as being 'part of' a Season. Note that multiplicities can be added to aggregate associations as normal.

Sometimes, aggregation relationships can be denoted as being stronger than just a 'is part of' relationship. When one class is said to have *life-cycle* control over the instances of another class, this can be denoted on a class diagram as a *composition*. Life cycle control means that should the instance of the composite class be destroyed, all the instances of the constituent class in the composition are destroyed as well. Figure 13.13 illustrates the use of the composition rela-

## Aggregations (359)

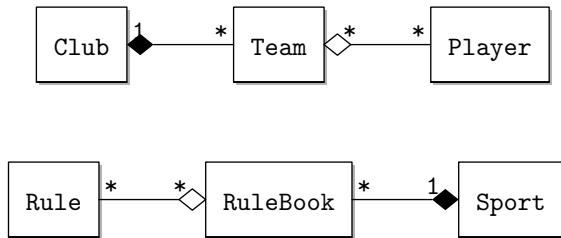


Figure 13.13: composition decoration of associations in UML

tionship.

Compositions (360)

In the figure, a Club is denoted as being composed of a number of Teams. Should a Club instance be destroyed (perhaps because the club closes) then all the teams drawn from the club will be destroyed as well. This contrasts with the relationship between Team and Player. Although a Player is denoted as being part of a Team, should the Team be destroyed, the Player instances won't be, because the Players could find other Clubs to join and Teams to play for.

Similarly, a Sport has a RuleBook that will be destroyed if the Sport instance is destroyed. However, the Rules that constitute the RuleBook won't be destroyed, as they may be used in more than one RuleBook. Note that an instance can only be grouped into one composite at a time - if the instance is a member of two composites, then either of the composites might try to destroy it when they were disposed with.

The overall functionality of aggregates (and composites) can often be re-alised by distributing the computation amongst the member elements. This can be particularly useful when the member elements of the aggregate have different concrete implementations of the task to be computed. This arrangement is called *propagation*, because the computational task is propagated to each of the member elements of the aggregate, without the composite knowing of the precise implementation details of the member elements.

Figure 13.14 illustrates the propagation of functionality for calculating the total runs scored by a cricket team during a single innings. A SideInnings instance is composed of a number of PlayerInnings. The SideInnings class propagates responsibility for storing the total runs scored be a player to the PlayerInnings class. Then, the SideInnings only has to iterate over the number of individual PlayerInnings classes to produce a total number of runs for the side. In addition, the second method `getRunsByBattingOrder()` can also be largely delegated to each of the individual PlayerInnings instances.

The code below shows how propagation of work in the `getTotalRunsScored()` method works from the SideInnings to PlayerInnings classes.

Propagating  
Functionality (361)  
SideInnings (362)

```

Map<CricketPlayer, PlayerInnings> innings;

public Integer getTotalRunsScored(){
    Integer result = 0;
}

```

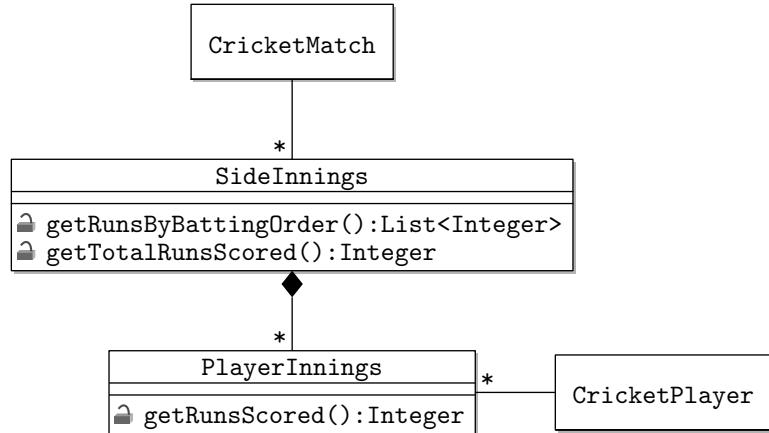


Figure 13.14: using compositions for functional propagation

SideInnings (363)

```

for (PlayerInnings players: innings.values())
    result += players.getRunsScored();

return result;
}

List<CricketPlayer> battingOrder;

public List<Integer> getRunsByBattingOrder(){
    List<Integer> result = new ArrayList<Integer>();

    for (CricketPlayer player: battingOrder){
        PlayerInnings pInnings = innings.get(player);
        if (pInnings != null)
            result.add(pInnings.getRunsScored());

    }
    return result;
}

```

*Delegation* is a similar technique to propagation, and is also associated with composites. Sometimes the desired functionality for a class can already be found to have been substantially implemented in another. Recall the `Map` interface (and implementing classes) from the collections framework covered in Chapter 2. The `java.util` package also contains a `Properties` class used to manage configuration options for applications and for the Java Virtual Machine. A single property is a key/value combination that can be stored in a `Properties` instance, or serialised out to an XML or text file stream. Rather than re-implementing the functionality necessary to store key/value combinations, the `Properties` class delegates responsibility for this to an implementation of the `Map` interface. The `Properties` class provides the extra functionality for serializing properties to the

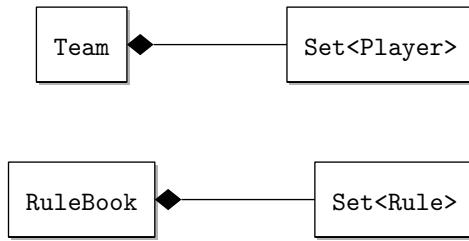


Figure 13.15: using compositions for functional delegation

various available formats.

Figure 13.15 illustrates another example of delegation. The `Team` class is responsible for storing the `Player` instances currently associated with the team. Rather than re-implement (and hard code) the functionality for storing the `Player` instance, the model is arranged so that responsibility for storage is delegated to the `Set` class. The `Set` class already enforces the rule that a player can only be in the team once, and provides methods for accessing and iterating over the `Player` members. Every `Team` instance then has a `players:Set<Player>` attribute that is used to store players. This form of delegation happens so often, that usually it is not made explicit.

Delegating  
Functionality (364)

### 13.3.4 Polymorphism during analysis

The initial stages of the object oriented modelling process are concerned with the identification of concrete types from the problem domain and their features and behaviours (represented as attributes and operations). As the model is refined towards an object oriented software design, we need to begin applying good design principles to the model. In particular, we need to avoid replicating implementation details shared between different classes in the problem domain. We have already introduced the notion of polymorphism in Section 2.4, including concepts such as generalization/specialization and inheritance.

There are several guidelines for appropriate use of generalisation relationships when refining an object oriented model:

- generalise by grouping common operations and attributes in the super-class;
- check the *is a* rule for all specializations;
- specialize the behaviour of sub-classes by over-riding explicit abstract operations in the super class;
- don't over-specialize, particularly for situations where only attributes are different; and
- avoid multiple inheritance for one class where possible.

Guidelines for using  
polymorphism during  
analysis (365)

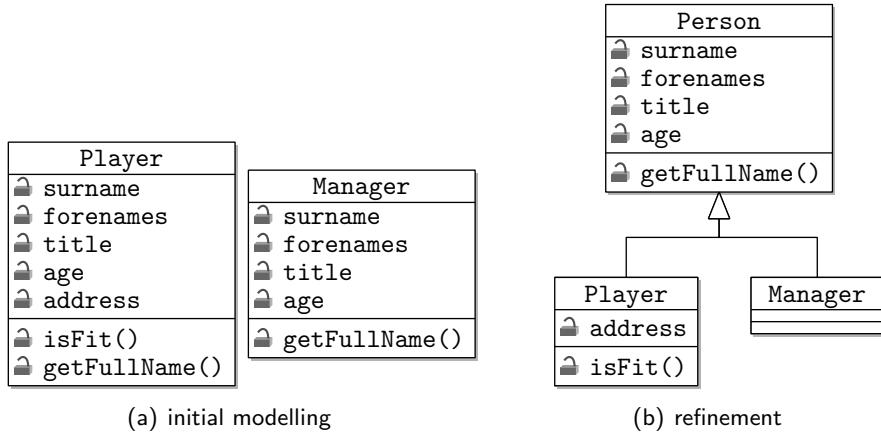


Figure 13.16: grouping common operations and attributes in abstract classes

Lets consider these guidelines in more detail.

The first step during object oriented model refinement is to identify areas of duplication in the domain model. Duplication may occur because the same features need to be represented in different concrete classes. When attributes which store the same data, or operations that exhibit the same behaviour are identified in two different classes, it may be appropriate to group the common features together in a more abstract general class.

Figure 13.16 illustrates the collection of common attributes and operations from the Player and Manager class into a more abstract Person class. The Player class retains the player specific features, the address attribute and theisFit() operation.

The Manager class is retained as a separate entity for now, since further operations or behaviours may be added in the future. This illustrates a further issue during modelling. Sometimes, a class may appear temporarilly redundant and it may be tempting to remove it from the model to reduce complexity. Consideration must, however, be given to the possibility that the features of the model are in-complete. In general, it is better to consider removing classes from the model later in the development cycle, when the scope of the model is better understood.

Notice that in the example above, the members that are grouped into the common Person class have the same *semantic* meaning in the Player and Manager classes. A manager's surname, for example has the same meaning as a player's surname.

It would even be appropriate (with a little refactoring) to group together attributes or operations with the same semantic meaning, even if they had slightly different labels or signatures. For example, if the Manager and Player class were identified by two different developers during problem domain analysis, it is possible that one developer would define an attribute Player.forenames and

Generalise by grouping  
common operations  
(366)

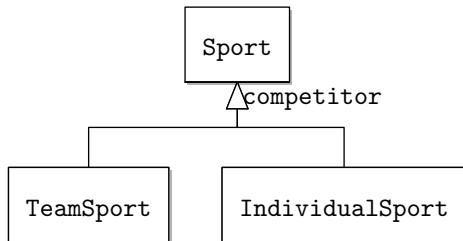


Figure 13.17: inheritance discriminators

the other defined an attribute `Manager.givennames`. In this case, the attributes could still be merged into the more general `Person` class, using one of the two labels.

However, it would not be appropriate to merge two classes based on members with the same label, if the two attributes have different semantic meanings. A refinement of this kind would violate the *is-a* rule, because the member in the super class would be used for different purposes in the sub-classes of the generalization relationship.

Complementary to this is the need to identify a valid *discriminator* for the specialized sub-classes. A discriminator is a label for a generalization relationship that can be used to justify the difference between each of the sub-classes of a super-class. Figure 13.17 illustrates a discriminator for the `Sport` class sub-classes.

In the figure, the `competitor` discriminator justifies the existence of two sub-classes, `TeamSport` for sports played by teams (football, rugby, cricket and so on) and `IndividualSport` for sports played by individuals (marathon running, javelin etc.).

A related problem in refinement is when instances within a single class share attribute *values*. When the value of an attribute changes in one instance, the value must change in all the other instances. This is effectively a one-many relationship between attributes within a single class.

In this situation, it can be tempting to generalise the commonly shared attribute into a super class to avoid duplication. However, this is inappropriate, because:

- the generalization will probably violate the *is-a* rule; and
- the value of the attribute will still be shared between all the concrete instances, even though it is defined in a super class.

Figure 13.18(a) illustrates an example of this situation. The developer has identified that sub-groups of instances of the `Side` class share the value of the `name` class. For example, if the name of a football team should change from 'Caledonian Thistle' to 'Inverness Caledonian Thistle' then the change will need to be reflected in all instances of the `Side` class drawn from that team.

Inheritance  
discriminator (367)

Abstraction-  
occurrence design  
pattern (368)

The developers have attempted to remedy the problem by generalizing the `name` attribute into the `Team` class. However, this is inappropriate for the reasons given above.

Figure 13.18(b) illustrates the use of the abstraction-occurrence pattern (see Section 18.2.1) as a better way of solving the refinement problem. The one-many relationship between `name` and the other attributes of `Side` is represented by a one-many relationship between `Team` and `Side`. Reading the relationship to validate it says, “A team fields many sides and a number of sides are drawn from the same team”. The `Team` is called the *abstraction* because it stores the values that remain unchanged. The `Side` is called the *occurrence* because it stores the things that will change each time a team fields a side to play a match.

Sometimes it is useful to specify a general abstract class before any of the concrete classes have been identified. This is more likely to occur during the later stages of refinement, as considerations regarding software re-use and re-usability begin to emerge. Sometimes, abstract classes are used to provide the general functionality for realizing an interface, with the final implementation details left to concrete classes. The `AbstractCollection` class, for example, provides almost all the functionality for accessing the elements of a `Collection`, leaving only the very low level methods regarding adding, removing and iterating over the collection to be provided in the concrete class.

When specifying abstract classes for later implementation, it is good practice to specify the signatures of abstract methods that any sub-classes must provide. This:

- proscribes the functionality of the sub-class;
- allows instances of the sub-classes to be fully treated as instances of the super-class.

Figure 13.19 illustrates this practice. The `Match` class is associated with the abstract `Result` class. Notice that `results` is denoted as abstract by using

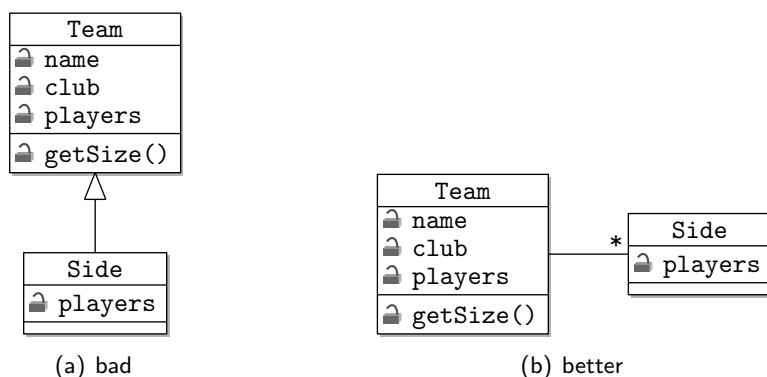


Figure 13.18: a bad generalization violating the *is a* rule

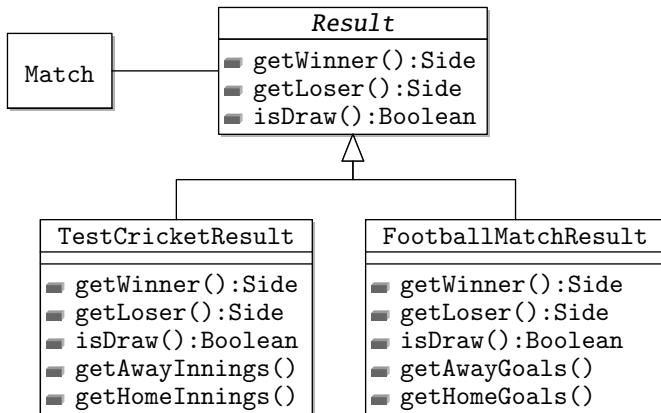


Figure 13.19: making specializations explicit with abstract operations

an italic font for the class name. The *Result* class is sub-classed by two concrete class's *TestCricketResult* and *FootballMatchResult* which provide the implementation details for deciding the outcome of a test cricket and football match, respectively. The implementation of these two classes is enforced by the specification of abstract method signatures for deciding whether the match was a draw and for identifying a winner and loser in the match.

By arranging *Result* as an abstract class, whose concrete sub-classes provide sport specific implementations, the *Match* class can utilise the behaviour of the general class, without knowing precisely how each of the abstract methods is implemented.

When no methods are over-ridden in sub-classes, it is worth considering whether the generalization is needed at all. Similarly, avoid over generalization for attributes only, particularly when the generalization is used to manage the partition of attribute values, rather than the identification of extra attributes in sub-classes. Figure 13.20(a) illustrates an example of excessive specialization. The *Player* class is sub-classed into *ProfessionalPlayer* and *AmateurPlayer*. This specialization appears to have no role, except to distinguish the career status of a sports player. Consequently it is probably better to remove the specialized classes, and introduce, an attribute to record whether a *Player* instance represents a professional or not, as shown in Figure 13.20(b).

Specialized classes that only have different attribute values from other sub-classes may be better represented as instances. In addition, over-specialization may mean that an object's concrete class may need to change over its life-time. This can be error prone and should be avoided.

The final guideline concerns the use of multiple inheritance, a feature that is not supported by all object oriented languages. Multiple inheritance is available in UML, and, as illustrated in Figure 13.21(a) the usual rules for validating the generalization rules should apply. The *PlayerManager* represents a person who is both a *Player* in, and a *Manager* of a team.

Specialize by explicit over-riding (369)

Don't specialize by only attributes (370)

Avoid multiple inheritance (371)

Despite its availability, multiple inheritance should be avoided if at all possible, because:

- of the additional complexity that is introduced into an object oriented model. In particular, a software designer must manage how identical members are inherited by a sub-class from two or more super-classes; and
- multiple inheritance often represents situations where an object is obliged to change its type over time. For example, a Person may begin their sporting career as a Player, progress to being a PlayerManager and finally end their career as a manager of a team only. This process of copying attributes from an instance of an old type to an object of the new type must usually be done manually and can be error-prone.

The *player-role* design pattern (see Section 18.2.2) can often be employed to solve the problem of multiple inheritance. Figure 13.21(b) illustrates the use of the pattern. The Person class acts as a *player*, which has a one-many relationship with a number of *roles* which can be played in the object oriented system. The roles assigned to a player may change over time without changing the type of the player object. In the example, a Person instance can be assigned a number of SportRole roles (which is an interface), such as as a Player, Manager.

The player-role pattern example above introduces the notation for *interfaces* in UML class diagram. Recall that interfaces are used to specify the operations that a concrete class must implement as methods.

Figure 13.22 illustrates two variants of the notation. In both cases interfaces are denoted in a similar way to classes, except that an interface does not by default have a compartment for attributes. The interface on the right is classified explicitly using a *class stereotype*. The interface on the left is denoted using a *visual stereotype* - the italic font for the interface class name.

Note that interfaces cannot have instance attributes, since interfaces do not provide implementation for instances). Consequently, it is not appropriate to draw an association from an interface to another entity (although the reverse is fine). However, it is possible to denote an interface as having a class attribute.

Denoting interfaces  
(372)

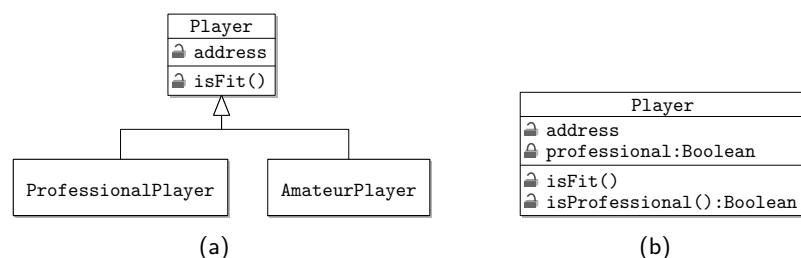


Figure 13.20: merging excessive specializations

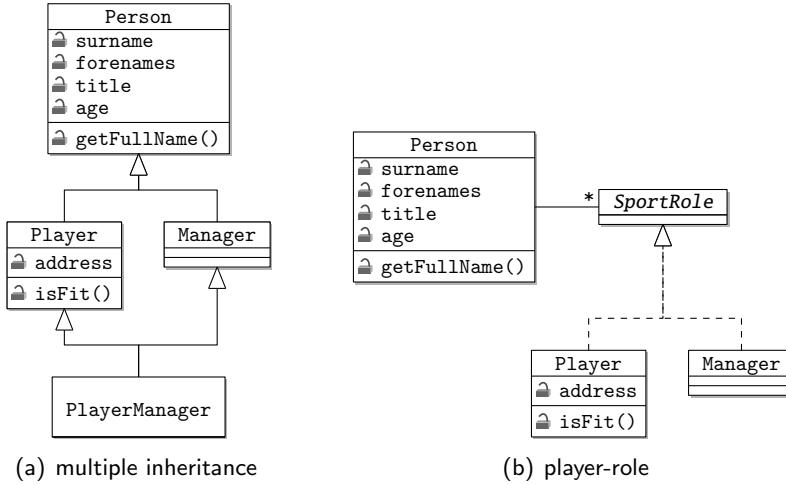


Figure 13.21: using the player-role design pattern to avoid multiple inheritance

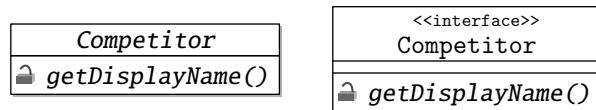


Figure 13.22: denoting interfaces in UML

For example, it may be necessary to specify the flags that an interface operation can accept.

Figure 13.23 illustrates the two ways that the realization of an interface by a class can be denoted on a class diagram. Figure 13.23(a) shows the use of a realization relationship between a class and an interface. Realization relationships are similar to generalizations, except that the line is dashed.

Figure 13.23(b) illustrates the use of an exported interface from a class. We have already seen exported interfaces on component diagrams - a 'lollipop' attached to the realizing entity and labelled with the name of the realized class. This form of interface notation is generally less common, although it can be useful for denoting the realization of 'common' interfaces that occur throughout an object oriented program.

Realizing Interfaces  
(373)

### 13.3.5 Extending Class Diagrams

The descriptive capabilities of UML class diagrams can be extended with:

- informal documentation to aid understanding, in the form of notes; and
- formal extensions using the *stereotype* mechanism.

Notes are a visual representation of informal documentation on all types of UML diagrams. Notes can be used for:

Using notes (374)

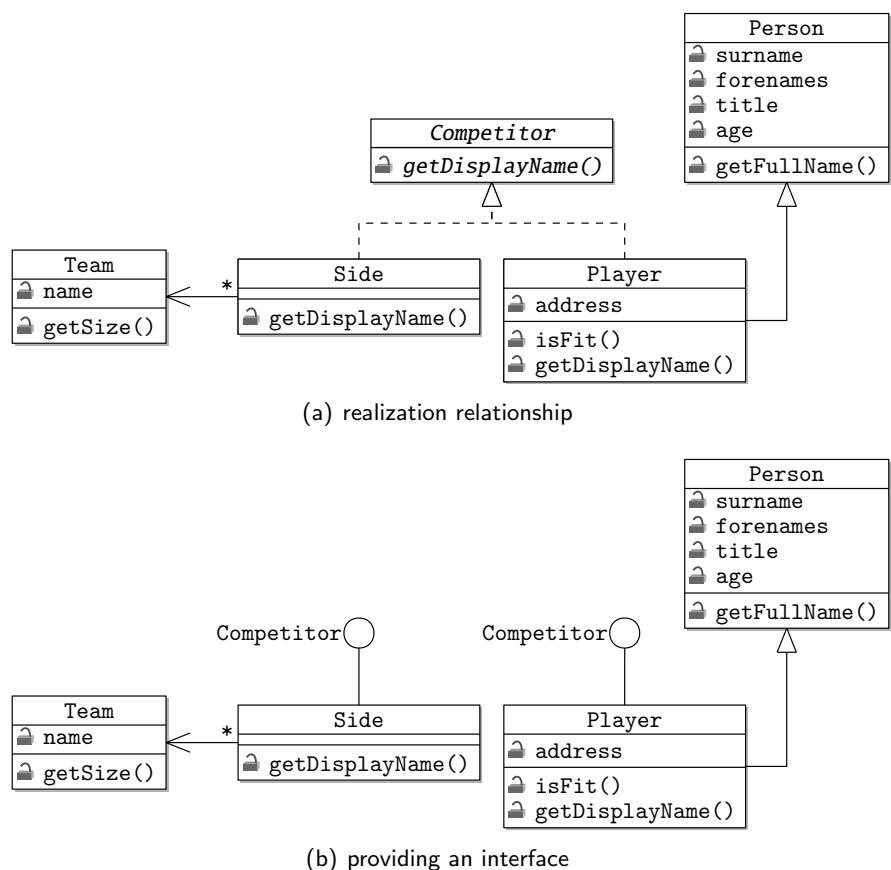


Figure 13.23: realizing interfaces in UML

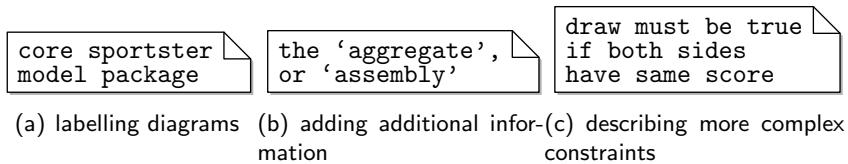


Figure 13.24: uses of notes in UML

- labelling diagrams to provide a reference identifier. This can be particularly useful for linking between diagrams;
- adding additional explanation to explain a particular design choice or link with a particular requirement that a design feature satisfies; and
- informally describing more complex constraints that cannot be expressed using multiplicities.

Figure 13.24 gives some examples of the various uses of notes on a class diagram. In fact, we have already seen how to associate notes with particular elements of a class diagram in Figure 13.12, by drawing a dashed link between the note and the element of interest.

Notes can also be used to annotate diagrams with formal constraints in the Object Constraint Language (OCL) Constraints. Chapter 25 describes the use of formal methods, including OCL, in the specification and design of software systems.

In some cases, the standard relationships for class diagrams are not appropriate to denote the particular kind of relationship needed. In this case, a general *dependency* relationship can be used, indicating that one class is dependent in some way on the other.

Dependencies can be annotated with a *stereotype*, a formal meaning, which extends of the core UML standard. The stereotype label is depicted between a pair of *guillemets*.

Figure 13.25 illustrates the use of a stereotyped dependency. The figure shows that the Main class is responsible for creating instances of the League class. The Main class is probably an entry point into an application, responsible for parsing any command line options. The League class is probably a *singleton* instance representing the front end of the functionality of the sportster framework.

In fact, stereotypes can be used to annotate general dependency links and other UML diagram elements such as classes. Some standard dependency stereotypes for classes are:

- <>, which denotes that the class is an intermediary between the object oriented system and its environment. Examples of boundary classes include user interfaces and classes that interact with external system resources;

Dependencies and stereotypes (375)

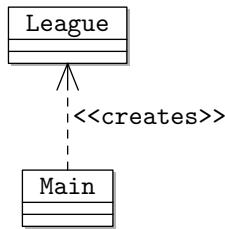


Figure 13.25: using dependencies and stereotypes in UML

- <<control>>, which denotes that the class manages the flow of control between other classes;
- <<thread>>, which denotes that the class defines an independent *thread* of execution, including its behaviour and state;
- <<table>>, which denotes that the class represents a table in a relational database. Class diagrams are suitable for describing entity relationship models; and
- <<enumeration>> which denotes that the class has a set number of pre-defined and labelled instances.

Some standard stereotypes for dependencies are:

- <<creates>>, which denotes that instances of one class are created by another; and
- <<import>>, which denotes that one *package* is dependent on the classes in another. We will look at package diagrams in Chapters 18 and 19.

In fact, much of the UML standard is defined using the stereotype mechanism. A stereotype may have an equivalent visual representation.

Two or more dependencies with the same stereotype should be interpreted in the same way. Stereotypes can be used throughout the UML. Many common stereotypes are given a graphical depiction - many of the relationships we have covered in this chapter are actually stereotyped dependencies.

## Summary

This chapter has looked at the use of UML class diagrams for documenting the key elements of a problem domain. Class diagrams can be used to denote the key elements in the domain as classes, and the relationships between the elements as associations.

We also looked at how class diagrams can be refined and enriched as a software design is developed. This process exploits the polymorphic features

of objects. When refining class diagrams it is vital to validate the models constructed by checking the meaning of the different relationship types used. The two high level types of relationship on a class diagram are associations, which should be read as *has a* and generalizations, which should be read as *is a*.

Finally, we looked at using interaction diagrams to translate from descriptions of the interactive features of a system into structural representations, using sequence and communication diagrams.

## 13.4 Exercises

1. Consider the problem description given in Figure 13.26.

A forensic software company is developing an application for acquiring forensic evidence from mobile devices. The software must work with a unknown number of different types of mobile device, including different brands and manufacturers of mobile phones, media players and smartphones. Mobile devices have a model and manufacturer. A model has a memory capacity and a form factor (candy bar, touch screen etc.).

A typical *use case* for the application is for the mobile device to be connected via a USB cable to a personal computer. The software application will then utilise a device driver (a software library for interacting with hardware) to interact with the mobile device via the USB cable. Some manufacturers provide device drivers for their mobile devices, while others must be custom written by the company's development team. Many generic device drivers are suitable for use with a number of different models.

It is assumed that each device stores its data on one or more NAND chips. The entire image of the device must be transferred to the desktop application for analysis via the device driver.

The memory image of any phone is organised into a particular file system arrangement (such as TFAT32, iOS or YAFFS) Once the image is transferred, the file system must be decoded and the contents presented to the user. The file system may be flat or organised into a hierarchy of directories. There may be many different types of file that must be presented to the user, including images, word processor documents, voice recordings, sound tracks, call records and SMS message databases. Each file type must also be decoded so that the contents can be presented to the user for further analysis.

Figure 13.26: mobile forensics application problem description

- (a) Identify the key concrete classes in the problem description
- (b) Identify the attributes of the classes.
- (c) draw a class diagram showing the initial associations between classes that you can identify.
- (d) List the responsibilities in the description.
- (e) Draw a sequence diagram for the acquisition of memory image from a mobile device. If your diagram identifies new classes of instance add them to the class diagram.
- (f) Refine the class diagram by identifying potential areas where generalization relationships could be applied.
- (g) Allocate the responsibilities to classes as *collaborations* and decide what class operations are needed to fulfill them.

- (h) Refine the diagram further by adding types and arguments to operations, and identify new classes.
2. Consider the problem description given in Figure 13.27.

The Sock and Piddle micro-brewery needs a new system for tracking inventories of beer in a warehouse. Beer is sold to local public houses (pubs) and shops. There are a number of different brands sold by the brewery, each with a name (e.g. 'Piddle's Best'), hop (brown beer, golden or stout only), an alcoholic strength (expressed as a percentage of alcohol in the beer by volume), and a price per litre sold.

Beer is stored in bottles and barrels. Both bottles and barrels of beer have a sell by date. Bottles of beer have a fixed volume of one litre. The cost of a bottle of beer is the cost of one litre of the beer and the cost of the bottle.

The brewery sells several different barrel sizes by volume. Barrels are returned to the brewery when empty so the price of a barrel of beer is calculated by multiplying the price per litre of the beer by the volume of the beer (i.e. the barrel itself is not charged for).

Pubs only buy barrels of beer and shops only buy bottles. All sales (to pubs and shops) are recorded on an invoice, which is used to prepare the delivery and adjust stock levels in the warehouse. The invoice includes the date of the sale, the name and address of the purchaser and the barrels or bottles to be sold.

Figure 13.27: the Sock and Piddle micro-brewery domain description

Model the problem domain with a UML class diagram. You should include the key classes, their attributes and any operations that can be identified from the description. In addition you should show the relationships (including any multiplicities) between the classes in the diagram.

3. Consider the class diagram shown in Figure 13.28 of part of a music player.

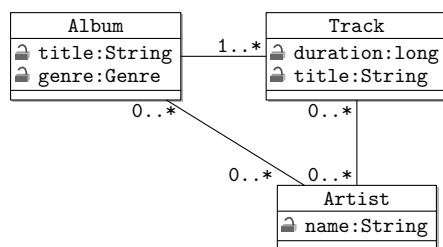


Figure 13.28: class diagram of a music player

Draw an instance diagram that represents the state of the music player when a single music track called “Drinking Piddle’s Best” from the album “Sock and Piddle Nights” is stored in the application. The track is a collaboration between two artists called “Barman” and “Landlord”.

4. Consider the instance diagram shown in Figure 13.29.

The figure shows a collection of objects representing newspapers, magazines, cities and countries

- (a) draw a class diagram to represents the structure of an object oriented system that could contain the objects and relationships shown in the diagram. Include the necessary associations and multiplicities.
- (b) Review the class diagram you have drawn; could it be refined by generalizing some of the classes?
- (c) A decision is made to record every edition of each newspaper and magazine.
  - i. What *design pattern* would you apply to achieve this?
  - ii. Add the changes you propose to your class diagram.

5. Take another look at the problem description shown in Figure 13.26

- (a) Draw a sequence diagram that represents the series of interactions described for recovering a memory image from a mobile device.
- (b) Translate the sequence diagram into an equivalent communication diagram.
- (c) Draw a class diagram of the types shown on your communication diagram.
- (d) How does the class diagram compare to the structure you developed for the exercise above?

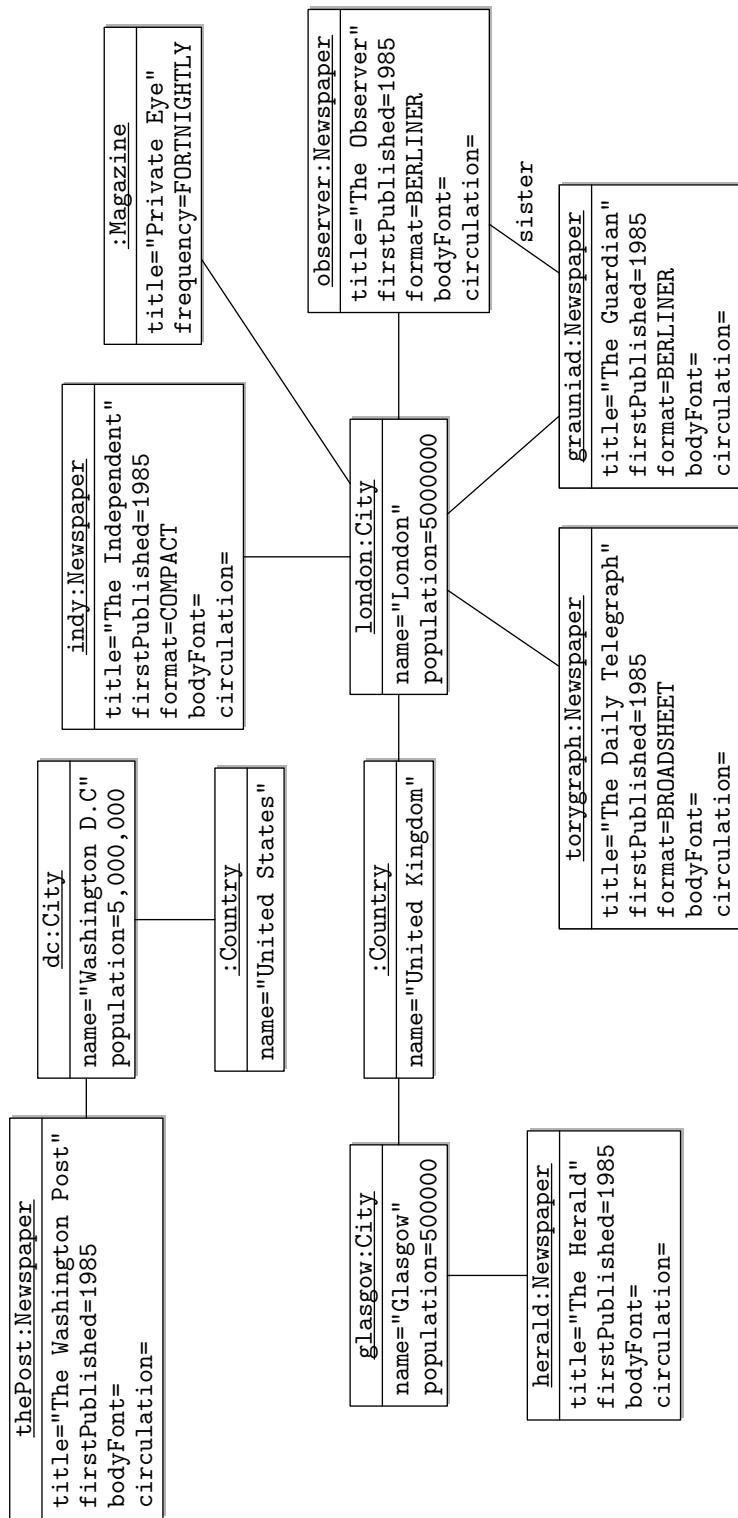


Figure 13.29: an instance diagram showing, newspapers, magazines, cities and countries



# Chapter 14

## From Domains to Designs

### Recommended Reading

PLACE HOLDER FOR sommerville10software

PLACE HOLDER FOR bennett06object

### 14.1 Introduction

Chapter 3 introduced the elaboration of *use cases* and *domain models* as key methods for documenting the requirements for a system. Use cases (described in Chapters 11 and 12) are used to specify the functionality of a system, i.e. what the system will do. Domain analysis and modelling were explored in Chapter 13. Domain models describe the entities in the environment that are to be represented in the system in some way. This chapter explores the concept of domain modelling further, using the class responsibility collaboration (CRC) method to elicit UML class diagrams and to begin to extend domain models into object oriented designs.

An object oriented approach can be adopted at several stages of software development. However, the application of object oriented techniques and the tools and notation adopted are different in each case, as illustrated in Figure 14.1.

Object oriented analysis and design

**during analysis** an object oriented approach is used to identify the entities in the domain, as referred to in the original problem description and use cases. These are documented as a high level design which identifies each domain entity type, their attributes and the relationships between them. Figure 14.1(a) documents the features of the Book class, identified for the branch library system. At this stage, it is identified that the book has an author, however, further details are omitted.

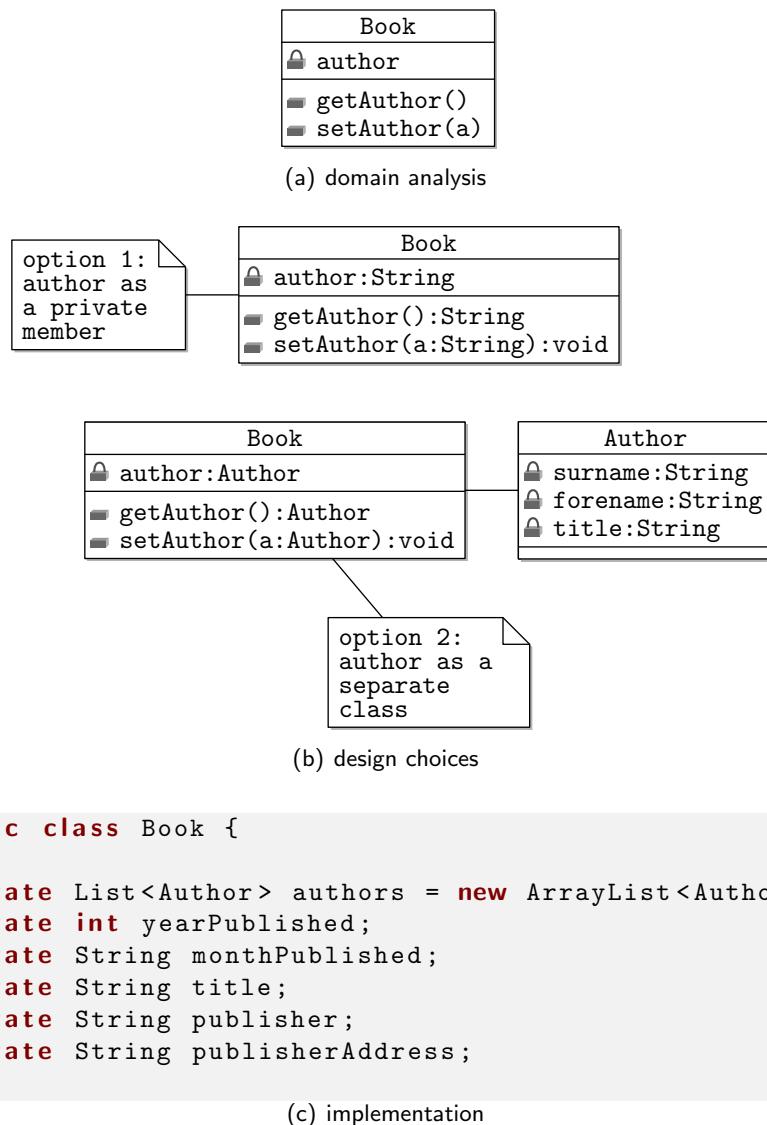


Figure 14.1: Object oriented analysis, design and implementation

The keeper selects the 'request book for branch' option from the main menu and enters a search term for books by either author or title. The keeper selects the desired books from the list and either searches again or confirms the request for the list of books.

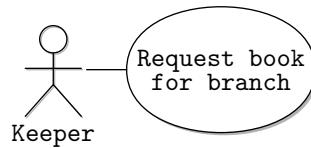


Figure 14.2: identifying entities in the request book for branch use case

**during design** the class models are refined and extra detail is added. Design alternatives are also identified and resolved at this stage. Further classes may be added and the architecture of the system gradually becomes established. Further refinements may be made to the design later on to accommodate the constraints of external libraries or platforms, or to leverage *design patterns*. Figure 14.1(b) shows two different design alternatives for the Book class. In option 1, the author is represented as a class attribute. In option 2, a book author is represented as a separate class, with its own attributes. Option 2 is selected, to allow for the possibility of a many to many relationship between authors and books.

**during implementation** the chosen design is translated into an executable object oriented system, using a language such as C++, Java, Python or C#. Figure 14.1(c) shows the implementation of design option 2 in Java. CASE tools such as Umbrello or StarUML can be used to translate class diagrams into object oriented program *stubs*, containing the attribute and method signatures of the class.

In summary, object oriented *analysis* is concerned with documenting the features of entities in the problem domain and the relationships between them. Object oriented *design* is concerned with translating the domain representation into a system architecture suitable to be implemented in an object oriented programming language. Design models document the software components to be implemented. This chapter is concerned with early stage object oriented analysis, we will return to object oriented design in semester 2.

## 14.2 Class Responsibility Collaboration

Problem descriptions and use cases are the primary source of information for domain analysis, since they implicitly describe the entities in the problem environment that must be represented in the system. As with other aspects of requirements elaboration, the domain model must be developed iteratively as the set of use cases becomes more refined.

Eliciting domain entities

class name:	
<b>Branch</b>	
responsibilities:	collaborations:
provide lists of books; handle requests	Book, Request

Figure 14.3: branch class CRC card

#### CRC card schema

A straight-forward method for identifying domain entities is to extract *nouns* in problem descriptions. Consider the use case description in Figure 14.2. Each of the nouns has been emphasised. Of immediate interest are the nouns *book*, *branch*, *keeper* and *request*.

We now need to translate from the domain entities we have identified to a UML class diagram. Class-responsibility-collaboration (CRC) is a method for eliciting and documenting domain classes during analysis Beck and Cunningham [1989]. Candidate classes are recorded on index cards that can be inspected and re-organised to elicit the structure of domain relationships.

Figure 14.3 illustrates the format of a CRC card, using the Branch class as an example. The name of the class is placed at the top of the card. The rest of the card is divided into sections used to describe the *responsibilities* of the class and *collaborations* with other classes.

Responsibilities can be categorised as:

**doing:** such as coordinating activity, invoking operations or providing services; or

**knowing:** the private data and associations held by instances of the class or how to calculate to a value.

#### Finding collaborations

The class descriptions are documented by hand on CRC cards should be informal and used for a basis for discussion. As well as recording them on the card, collaborations can also be visualised by laying out the cards and drawing lines between them, e.g. on a white board in the team's area. Figure 14.4 illustrates the collaborations between four of the different classes identified so far from the use case description.

The CRC cards are grouped together to investigate how they collaborate with one another. Lines are drawn between collaborating classes indicating the basic structure of the interaction.

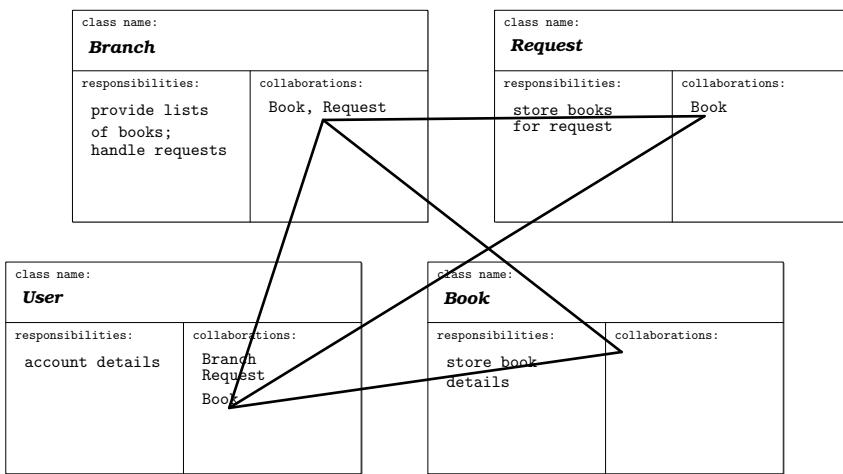


Figure 14.4: class collaborations

### 14.3 CRC Cards to UML Class Diagrams

We can now use the CRC cards to begin developing a class diagram. We need to define a class with a name, operations and attributes for each of the CRC cards. The name of the class comes directly from the name on the CRC card. The attributes and operations can be derived from the responsibilities identified for the class. Broadly, knowing responsibilities will be represented as attributes of the class and doing responsibilities will be represented as operations. Figure 14.5 illustrates an initial class diagram containing classes for each of the CRC cards developed from Figure 14.4.

Notice that much of the functionality is concerned with manipulating the information we wish to represent about the domain. An initial attempt at establishing the likely attributes and relationships has been made, however, further functionality will be added during later design stages as the architecture becomes more established. The domain model will also be extended further as each use case is examined for domain entities.

Attributes can be derived directly from use cases and the problem description; or from the identification of ‘knowing’ responsibilities and equivalent operations. Alternatively, the absence of details of the attributes for a class may indicate that further requirements elicitation activities are required. The initial analysis does not need to establish the precise representation of an association as this will be done at the design and implementation stages.

Returning to the class diagram introduced in Figure 14.5, we can now refine the attributes and relationships with further detail. Figure 14.5 illustrates the same class diagram with further detail. The association decorations from Figure 13.3.3 have been added, as well as the multiplicity relationships between the classes.

CRC cards to UML  
class diagrams

Revised class diagram

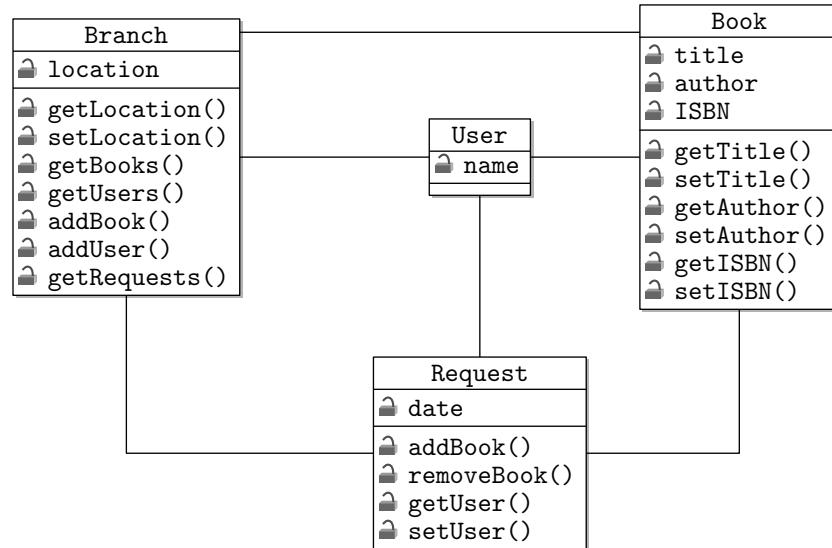


Figure 14.5: UML class diagram of the Branch, Request,User and Book classes

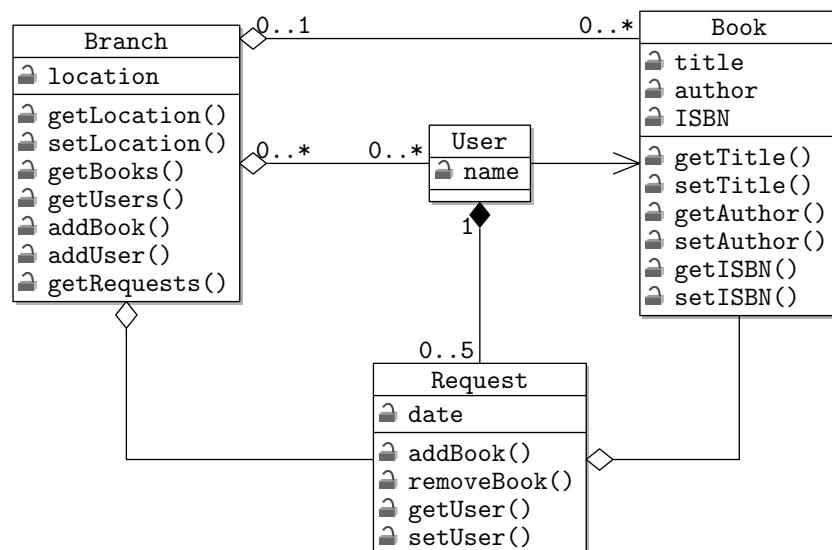


Figure 14.6: revised UML class diagram of the Branch, Request, User and Book classes

The diagram does raise the question as to whether the attributes identified for the Request and Branch classes are correct. We have assumed that branches have a location, but we have not yet decided how to represent this information in the system. We have also assumed that each request is associated with a date when it is made, in order to process requests in an orderly manner at the central library. However, we can't be sure from the current use case descriptions that this feature is necessary - perhaps requests are too infrequent to be concerned with ensuring a first-come first served ordering of requests. We may need to investigate these questions by developing a prototype and evaluating it with the customer.

## Summary

This chapter has reviewed the different uses of object oriented modelling for domain analysis, system design and implementation. The chapter focused on domain analysis using CRC cards and class diagrams for elicitation and documentation. We also considered the relationship between requirements elicitation and the domain model documentation process.

## 14.4 Exercises

1. Recall the problem description given in Figure 11.16
  - (a) create a set of CRC cards for the key candidate classes needed to realise the use cases (you will be provided with some index cards to do this);
  - (b) produce a class diagram showing the class names, attributes, operations and associations. Try creating two versions of the class diagram. The first version should just show the class names and associations between the classes. The second version should add attributes and operations to the key domain classes;
  - (c) identify the three most important analysis questions you would want to answer to improve or validate your domain model.

# Chapter 15

# Modelling Dynamic Behaviour

## Recommended Reading

**PLACE HOLDER FOR sommerville07software**

**PLACE HOLDER FOR bennett06object**

The Bennett et al. text is more useful for understanding the semantics of sequence diagrams.

### 15.1 Introduction

In Chapters 11 and 12 we developed a specification of system behaviour for the branch library based on a collection of use cases. Use cases are useful for describing dynamic interactions between actors and the system; a collection of use cases describe the system interface with the surrounding environment.

In this chapter we will look at a method for extending the models of dynamic behaviour from use cases into the system itself. To do this, we will need to use two new types of UML *interaction* diagram: *sequence diagrams* which show the flow of program control between instances in a system over time; and *communication diagrams* which show the groups of objects that collaborate during an interaction.

Figure 15.1 shows sequence and communication diagrams in the context of all types of dynamic UML diagram. We will use these dynamic representations of the system to establish a static architectural representation of the branch library system represented as an object class diagram.

UML diagram types  
(393)

We are concerned with modelling the dynamic behaviour of object oriented systems, which arises from the interactions of object instances, first introduced in Chapter 14. Instance diagrams are an intermediate representation between architectural and dynamic behaviour diagrams. Sequence and communication diagrams are used to show the flow of control and collaborations between object instances in a system.

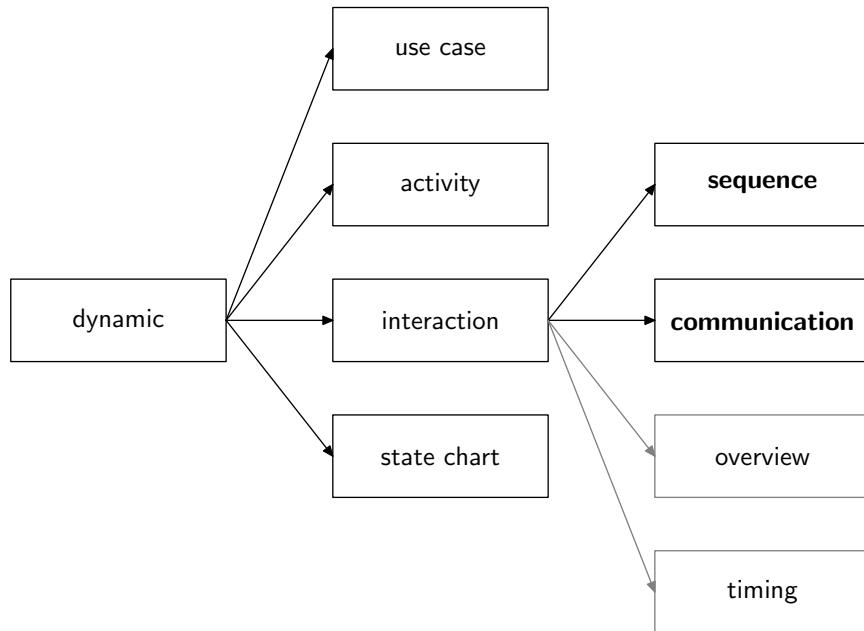


Figure 15.1: taxonomy of dynamic UML diagram types

## 15.2 Sequence Diagrams

Aside from showing the structural relationships between instances at run time, we also want to be able to denote the flow of program control between objects. Sequence diagrams can be used to show the passing of *messages* between instances over time. Messages on sequence diagrams may represent method or function invocations, remote procedure calls (RPCs), web service messages, network packets, or just about any other type of communication.

Figure 15.2 shows a simple sequence diagram for two object instances. Notice that the object instances have been extended with dashed *lifelines* that show time flowing vertically down the diagram. The instances have *active* periods denoted by a white rectangle on the lifeline. Program control transfers from an *active object* to a *passive object* when a message is passed. Messages have labels that may be equivalent to a method in the passive object. The passive object then becomes active and may pass messages to other objects in the system.

Messaging sequence  
and information flow  
(394)

Notice that sequence diagrams are concerned with the flow of control in a system, rather than information. However, it is possible to annotate message labels with parameters if desirable. In addition, the return of control from the passive instance to the active instance can be represented by a dashed arrow. However, in both cases, sequence diagrams can become rather cluttered if these are always used. Remember diagrams are useful for conveying meaning to a reader, as well as for literal documentation.

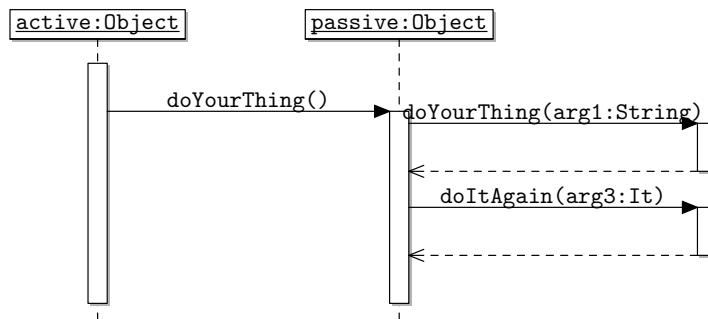


Figure 15.2: sequence diagram notation

By convention, users and boundary objects are placed on the left of the diagram, with *sequences of interactions* proceeding to the right where the internal instances of the system are represented. Instances which represent external systems that are part of an interaction are typically placed on the far right of the diagram. However, there is no requirement that messages are only passed to the right - it just tends to make the diagrams easier to follow.

Given that all instances presumably begin life in a passive state, we need to consider how to invoke sequences. Our model of systems is essentially *reactive* to external input, so invocation of a sequence must originate outside the system (or from a trigger from an internal timer object). Figure 15.3 illustrates the two ways to denote a use case being invoked.

Figure 15.3(a) denotes a GUI instance being activated by from the search for book use case. Each invocation from a use case on a sequence diagram should be associated with a step in the use case's activity diagram description. In this case, the first message, `selectSearch` is equivalent to the first step taken by the actor in the activity diagram which describes the use case. The activity diagram shows the steps an actor takes during a use case as the system produces output. The sequence diagram shows how the output from the system for each step is

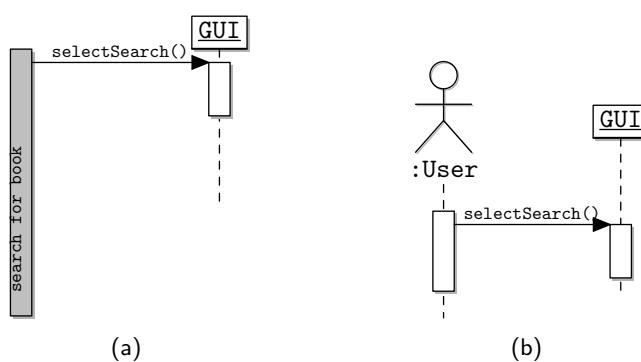


Figure 15.3: invoking sequences

Invoking sequences  
(395)

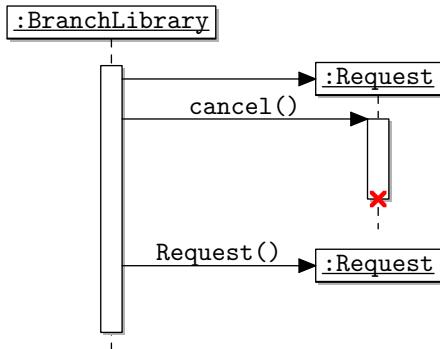


Figure 15.4: object construction and deletion

produced.

Figure 15.3(b) denotes an *instance* of an *actor class* invoking a sequence. Note that this is an instance of the actor classes denoted in the use case diagrams discussed in Chapter 11 and 12. This is convenient short hand where only a single actor class is associated with a use case.

Sequence diagrams illustrate dynamic behaviour of instances over their lifetime, so we need to represent object construction and destruction. Figure 15.4 illustrates the notation for construction and deletion of an object in a system.

Construction is denoted by a new message to the object to be instantiated, which you can think of as a class constructor message. Note that the new instance appears further down the diagram than others to indicate it is created later in time than the other objects. Some variants of UML use an explicit constructor label for instance creation, as shown for the instance in the bottom right in the diagram.

Instance deletion is denoted by a bold red cross at the end of the instance's final active period. Notice that instance deletion is explicit in the UML, unlike in memory managed OO programming languages, such as Java.

Sometimes it is useful to denote the messages that an instance passes to itself while it is active, sometimes referred to as self-invocation. Figure 15.5 illustrates the notation for self delegation. A new active period is nested on the instance's existing lifeline just after the point when the self invocation occurs. The message is drawn from the outer to the nested active period. Although the instance is not idle, nothing will happen in the outer active period whilst the nested invocation is active. This is denoted by shading the *idle* period in a darker colour.

Be careful when using this notation, as showing every internal message and indicating all periods of idling will quickly clutter the sequence diagram. Certainly don't include any messages that will be private members of the instance's class.

The flow of messages on a sequence diagram can be controlled using guards

Instance construction  
and deletion (396)

Self  
invocation/delegation  
(397)

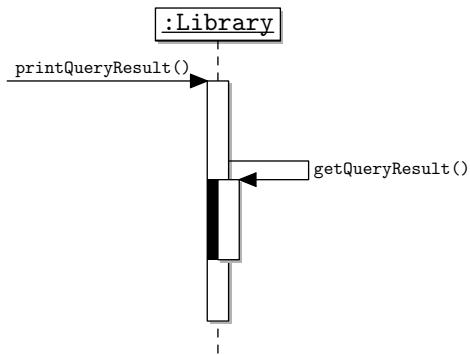


Figure 15.5: self-messaging in sequence diagrams

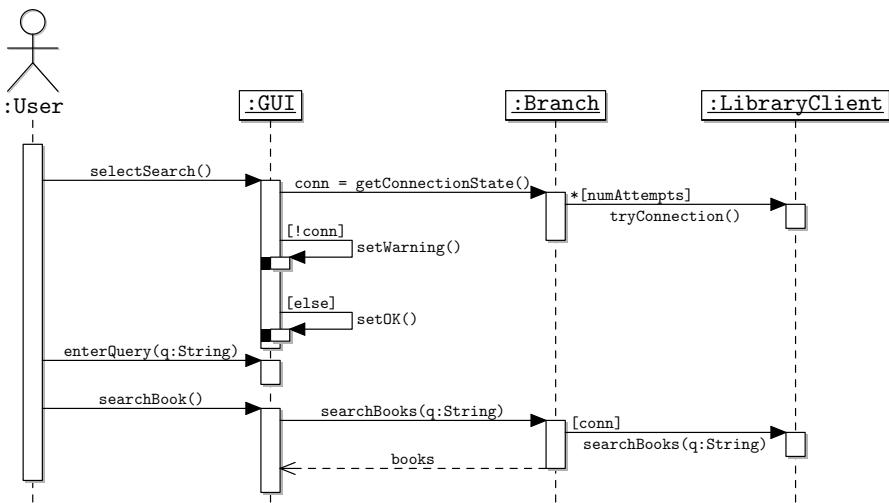


Figure 15.6: control structures using guards on sequence diagrams

in a similar way to sequence diagrams. Figure 15.6 illustrates how guards are associated with messages for both if-else and loop conditions.

The diagram shows the interaction between the Branch instance, and the component provided by the central library for interacting with their database, the LibraryClient. The type of connection status message set on the GUI depends on the result of the `getConnectionState()` message. If no connection to the central library is present, the branch library GUI shows a warning. Loop guards are decorated with a \*, as for the `tryConnection()` message.

If a guard is not satisfied then the message won't be sent. Note that this means that guards on sequence diagrams refer to the internal state of the system, whereas guards on activity diagrams refer to actor responses to system output.

In some variants of the UML sequence diagram notation, conditions and loops are encapsulated in sub-diagram boxes labelled with the guard. This notation can also be used to illustrate references to other sequence diagrams.

Control structures  
(398)

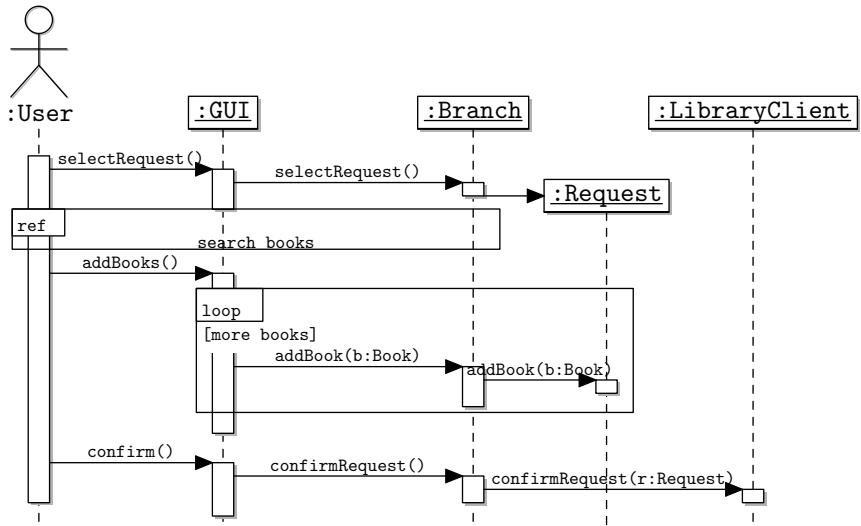


Figure 15.7: references to other sequence diagrams

## References (399)

Figure 15.7 illustrates the use of references for the request books use case.

We know from the use case description that the search books use case is included in lending books. This arrangement is shown in the diagram by a box extending across the lifelines of the instances involved in searching for books. The box is labelled with a reference to the search books sequence diagram, which we saw in Figure 15.6.

We have assumed so far that when an instance sends a message it waits in an idle state until a response is received. Similarly, we have assumed that the time taken to transmit a message from one instance to another is instantaneous. For method invocations between instances within the same process or run time these assumptions are probably reasonable.

However, there are many situations where these two assumptions are not appropriate, if:

- messages are to be transmitted across networks in which there is a noticeable delay; or
- processing of the message is expected to take the remote instance some time

## Asynchronous messaging (400)

it may be more appropriate for an instance to send a message and then continue processing other tasks while waiting for any reply. Figure 15.8 illustrates the use of *asynchronous* messages on sequence diagrams to denote when an instance sends a message and continues processing without waiting for a reply. The diagram is an alteration of the part of the search **for** books sequence diagram when the branch library system communicates with the central library.

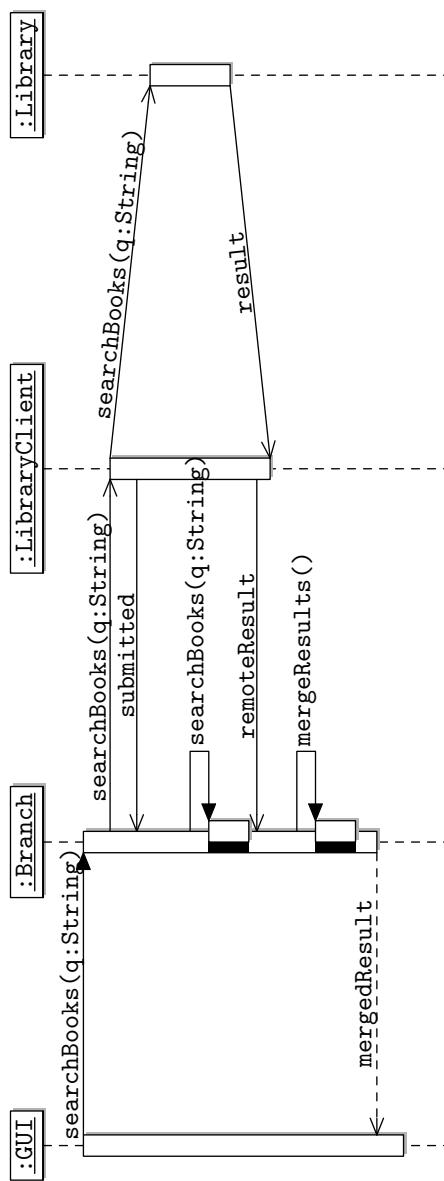


Figure 15.8: asynchronous messaging on sequence diagrams

An asynchronous message is denoted as an arrow with an unfilled arrowhead. On the diagram, the Branch instance sends an asynchronous message to the LibraryClient instance for a search to be made for books with the specified query. Once the message is sent, processing can continue. The Branch instance proceeds to perform a local search for the books requested that might be held in the branch. From the diagram, we can infer that the LibraryClient is responsible for communicating with the central library system from the branch. The LibraryClient forwards the query to the Library.

We have already seen how to use active periods on instance lifelines to denote periods of instance processing activity. Messaging delay is denoted by introducing a vertical gap between a message start and finish. On the diagram, the asynchronous message from the LibraryClient to the Library, and the reply is denoted as having a delay, presumably because this communication occurs over a network, rather than between instances in the same environment.

Once the message is sent, an instance can continue sending other asynchronous messages without waiting for any replies. The LibraryClient passes another message back to the Branch reporting that the query was successfully communicated. When the Library instance has processed the query it responds with a list of books. These books are then merged with the results of a local search to produce a final list displayed on the GUI.

Notice that asynchronous messages are not explicitly associated with corresponding return messages. Rather, any replies are themselves sent as asynchronous messages in their own right.

The time duration between delays in such a situation can be of concern to the specification and satisfaction of non-functional performance requirements. Sequence diagrams can be used to denote both the specification of performance requirements and the sources of performance problems in system designs. Figure 15.9 denotes the same sequence diagram in 15.8 annotated with timing constraints.

Timing (401)

The constraints are shown as measurements annotated with the appropriate in-equality. From the diagram, we can see that messages to and from the central library system can take up to two seconds, due to network delay, and processing of the request can also take up to one second. These limits may be problematic, since we have specified that the entire sequence of interactions should not take longer than three seconds for the user.

### 15.3 Communication Diagrams

Sequence diagrams are useful for denoting the flow of messages between instances over time (in sequence!) as a result of a particular event, which for our purposes, is typically the invocation of a use case. However, it is difficult to understand from a sequence diagram how communication between instances relates to the object-oriented structure of the system.

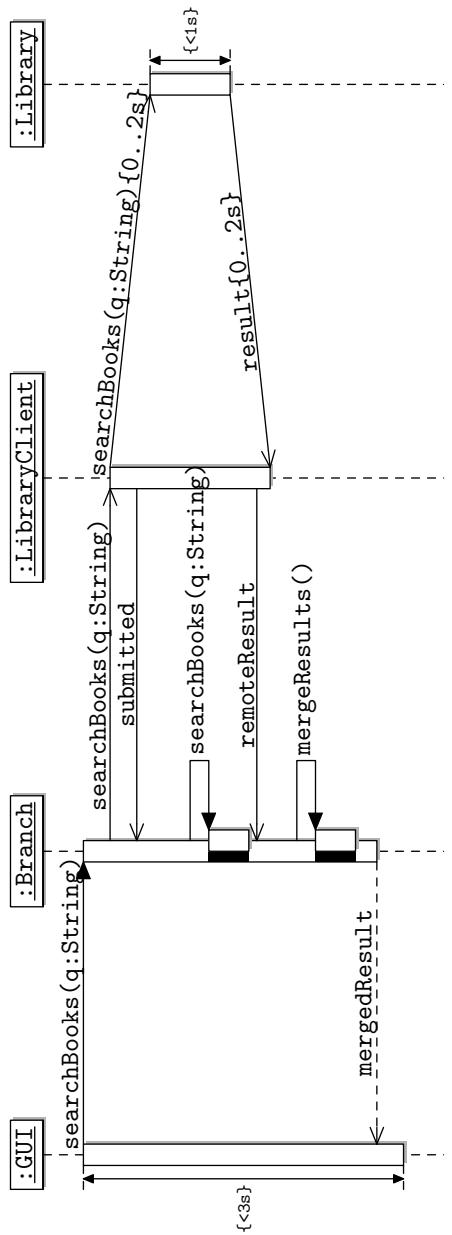


Figure 15.9: timing constraints on sequence diagrams

## Communication diagrams (402)

## Classes (403)

Communication diagrams are a hybrid of elements from instance and sequence diagrams, showing the sequence of messages passed between instances *and* the structural relationships that must exist between objects if they are to collaborate with each other. Figure 15.10 illustrates a communication diagram which is equivalent to the request book sequence diagram in Figure 15.7.

Communication diagrams are essentially instance diagrams annotated with sequences of messages. The messages are shown as arrows parallel to the associations between instances. Message identifiers are annotated with labels indicating message ordering. Timing information concerning message or processing delay is not shown on the diagram and generally, more complicated condition structures are also omitted.

Re-drawing sequences as communication diagrams begins to reveal the structure of the object oriented system we are developing. Drawing associations between instances that will be present at run time implies that the associations must also be present between the instance classes. Figure 15.11(a) illustrates a class diagram developed from the communication diagram in Figure 15.10.

The relationships between the classes generalise the specific example given on the instance diagram by introducing aggregations, multiplicities and so on. Notice that the classes don't immediately have any members (attributes or methods), since we've only identified the types from the sequence diagram. However, we can add the methods for each class as the messages that are passed to it on the sequence diagram. Figure 15.11(b) illustrates the full class diagram with class members added.

It is worth considering the two diagrams a little further here. UML diagrams used for specification and design of software systems serve two purposes which can be in conflict: a basis for the implementation of the final system and as a focus of discussion with customers and users.

In an ideal situation from an engineering perspective, there should be an automatic transformation from a UML design into an implementation in an object oriented programming language. When documenting design for this purpose,

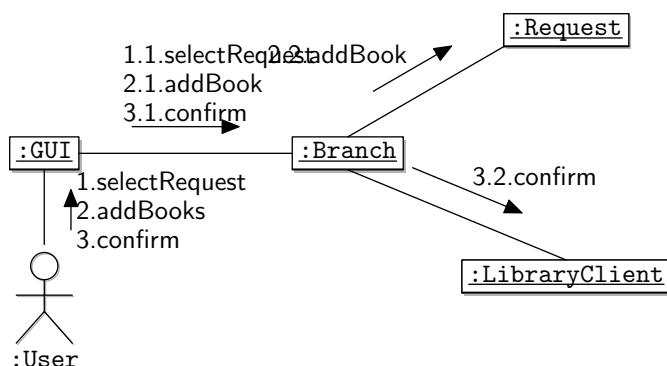


Figure 15.10: communication diagram for the request book use case

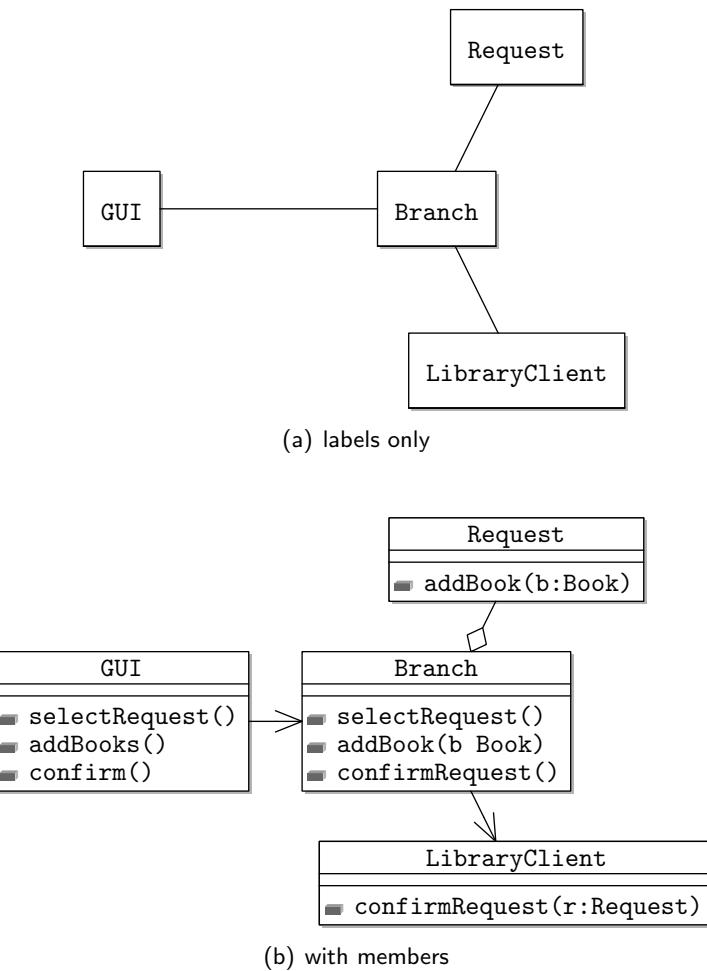


Figure 15.11: classes developed from the communication diagram in Figure 15.10

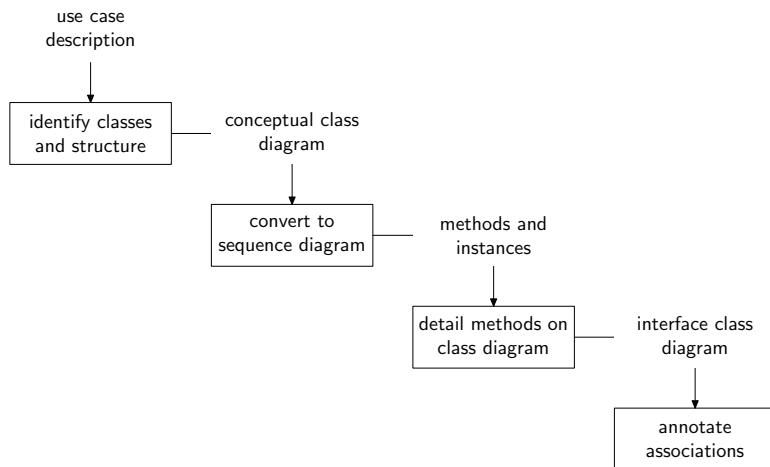


Figure 15.12: moving from use cases to architecture

diagrams need to be both complete and consistent, including as much detail as will be necessary to produce the implementation.

However, when using diagrams for discussion, too much detail can obscure the important aspects of the system that need to be discussed. A customer will probably not care what the identifier is for a particular label, but will be concerned if the overall system has insufficient throughput for their needs. In addition, it can be useful during discussions to introduce temporary inconsistencies while points of conflict or concern are being resolved.

Referring back to the diagrams in Figure 15.11, we can see that Figure 15.11(a) is more useful for early stages of the elaboration and construction of our system when discussion with the customer is important, while 15.11(b) provides a more precise basis for implementation by developers, and analysis for inconsistencies using CASE tools.

## 15.4 From Use Cases to Architecture

We have now reviewed a range of UML diagram types for modelling dynamic behaviour. Developing sequence, communication and then a class diagram has enabled us to translate the functional requirements for the system expressed as use cases to an initial structural representation of the system to be developed (recall in Chapter 3 that we were concerned with following rigorous methods for software development in order to reduce risk).

Method (404)

Figure 15.12 documents the method we have been following for developing architectural class diagrams from use case descriptions. The method begins by extracting a relatively simple conceptual class diagrams from the use case descriptions in the specification. Next, sequence diagrams are developed from the use cases, identifying the instances that are present in the system over time

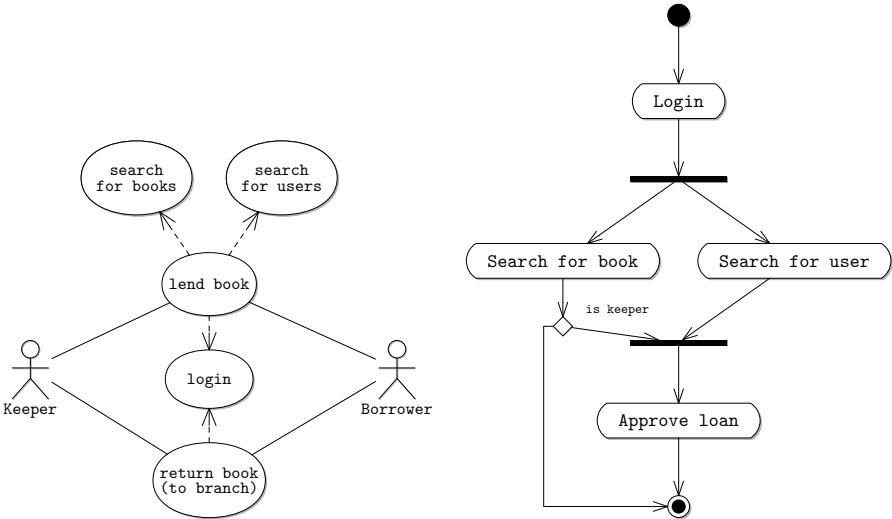


Figure 15.13: the lend book use case

and the messages that pass between them. These are used to annotate existing classes with methods and extend the class diagram as new types are discovered. Finally, the relationships between classes are identified from communication diagrams of interactions between instances.

We will now illustrate the method on a larger example, taking the collection of use cases concerning the lending of a book from the branch as a case study. Figure 15.13 shows an extended version of the use case diagram given in Figure 12.3, explicitly including the ‘common’ use cases: login, search **for** books and search **for** users. The figure also illustrates the activity diagram for the principal lend book use case. The first step is to develop a conceptual class diagram of the elements referred to in the use case. Figure 15.14 illustrates this diagram for the lend book use case, initially consisting of only the Keeper, Borrower, Book and Branch classes, with a few attributes for each. At this stage we haven’t added any methods or associations to the diagram.

Next, we need to model the steps of the use cases in a sequence diagram. Figure 15.15 illustrates an initial sequence diagram for the lend book use case. So far, we have only detailed the *boundary* messages that pass between the Keeper and the GUI instance (*selectLoan*, *selectBook* and *confirmLoan*), since

Lend book use case  
(405)

Conceptual class  
diagram (406)  
Initial sequence  
diagram (407)

<b>Book</b>	<b>Borrower</b>	<b>Branch</b>	<b>Keeper</b>
title author published	surname forename id	department	surname forename id

Figure 15.14: initial conceptual class for the lend book use case

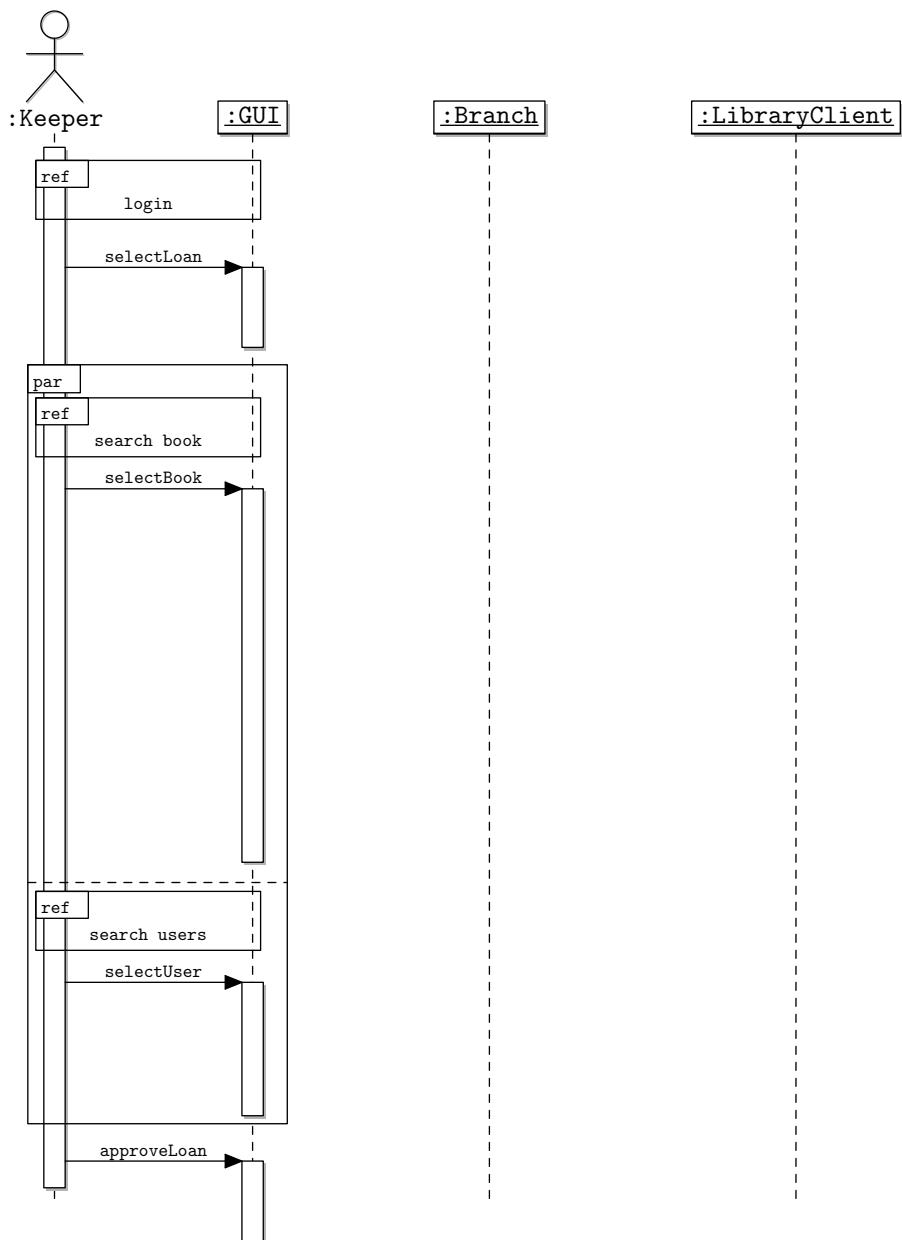


Figure 15.15: the initial lend book sequence diagram

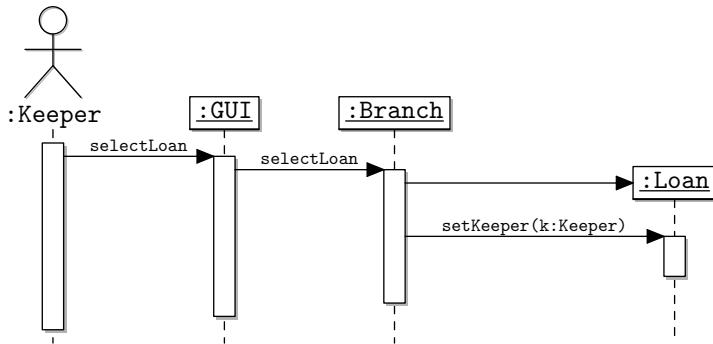


Figure 15.16: the `selectLoan` message

these can be directly inferred from the specification in the activity diagram. We can also show the position of the included use cases in the sequence of interactions for lend book. The diagram also introduces the notation for indicating that two sequences that occur in parallel for the search books and search user use cases. This is done in a similar way to references by encapsulating activity in a box, except that the parallel activities are divided by a dotted line.

The resulting sequence diagram isn't all that detailed. In one sense, this is encouraging, we have translated the encapsulation of specification details into the initial stages of design. However, we now need to begin extending the sequence into the interaction between instances within the system in order to uncover more detailed aspects of the design.

Initially, we can extend the `selectLoan` message. To do this, we need to create a new loan instance for representing the relationship between a book and a borrower. This gives us our first questions:

Extending `selectLoan`  
(408)

- how are loans to be represented in the system?
- what information must be stored about loans?

Figure 15.16 illustrates a first approach at addressing these questions for the `selectLoan` sequence. A new instance of the class `Loan` is created, so we will need to add this class to the conceptual class diagram. We also set the `Keeper` for the `Loan`, since we might need the information later, if for example, we want to check that the keeper isn't lending out books they don't hold in their possession.

But hold on, we haven't yet established which keeper is making the loan, which should be the current user. To do this, we will need to create a sequence diagram for the `login` use case. We can start this process as before by adding the boundary interactions between the user and the user interface which are already documented in the `login` activity diagram (Figure 12.17).

Next, we will need to explore the internal system interactions which result from a user submitting a username and password. In particular, we need to

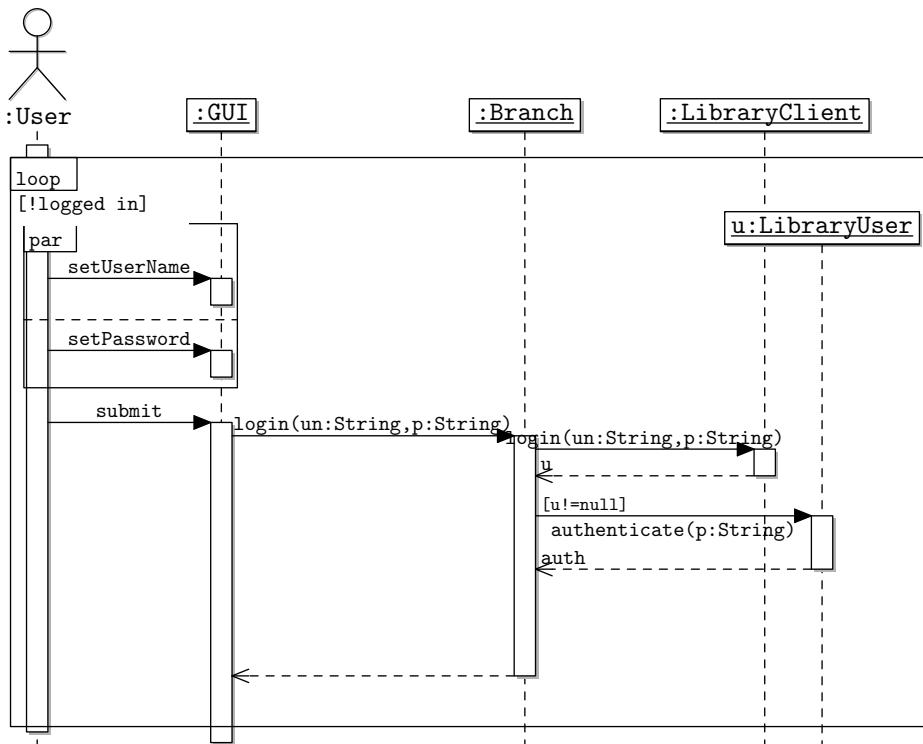


Figure 15.17: the login sequence diagram

establish how and where user credentials are stored. The requirements document states that the central library's database of users must be used to manage users. Consulting the documentation for the provided `LibraryClient` class, we discover that there is a method for retrieving users by username which returns a `User` class instance. The `LibraryUser` class has a method for authenticating a user from a supplied password. We can now complete the sequence diagram, as shown in Figure 15.17 by adding the additional messages between the `GUI`, `Branch` and `LibraryClient` instances. The sequence of interactions is completed by setting the currently logged in user in the `GUI` and changing the state of the branch to logged in.

login sequence diagram (409)

Now that we've completed the `login` sequence diagram, we need to return to the main `lend book` sequence. We will need to extend the `login` reference to encompass the `GUI`, `Branch` and `LibraryClient` instances. Now that we have established who is using the branch system, we also need to check whether the current user is a keeper, i.e. whether they are entitled to lend books at all. Figure 15.18 illustrates the `selectLoan` sequence diagram fragment that reflects these changes. We've added a guard so that a new loan instance is only created if the current user is a keeper. We've also revised the message itself so that the keeper of the loan is set when the loan is created, rather than via a separate message.

Revised selectLoan sequence diagram (410)

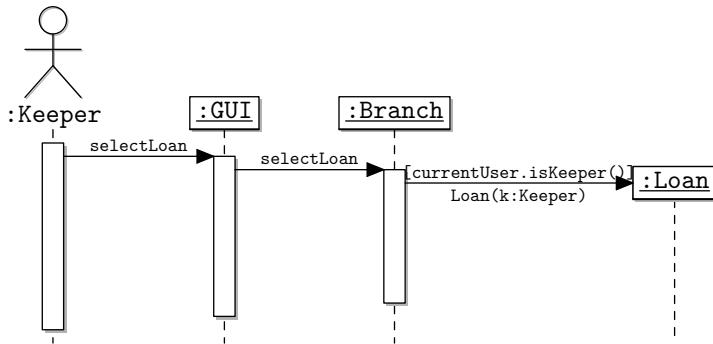


Figure 15.18: the revised `selectLoan` message

However, the diagram now presents us with a problem. We have included a guard to check whether the current user is a keeper or not. Unfortunately, a check of the documentation for the `User` class of the instance returned by the `LibraryClient` shows that the central library system doesn't store who are keepers in the individual branches. This information is going to have to be stored locally.

Figure 15.19 shows the revised login sequence diagram. Users are authenticated via the `LibraryClient` instance as before. However, once a user is authenticated, the `LibraryUser` instance is used to look up a `BranchUser` instance within the library. The `BranchUser` is then set as the `currentUser` of the branch. The `BranchUser` class will be implemented by us, so we can add a property to test whether the user is a keeper or not. Looking back, we will also need to add keeper information to the `request book` sequence diagram in the same way.

Lets return again to the `lend book` sequence diagram. We have already developed the `search books` sequence diagram and the `search users` diagram will likely be derivable from the messages sent in `search books` and `login`, so lets turn our attention to the `selectUser` and `selectBook` messages. At this point, we can ask:

- what happens to a book when it is selected by a keeper for lending?
- what checks need to be made before a book is lent?

Retrieving branch users (411)

`selectBook` sequence diagram (412)

Selecting a book should allocate that book to the `Loan` instance we have already created. We need to check that the keeper of the book is the current user and that the book is currently in their possession. Figure 15.20 illustrates the `selectBook` fragment of the `lend book` sequence diagram that implements these requirements. The diagram also introduces the use of `alt` alternative boxes. These are a convenience when two alternative message sequences can result from a guard, rather than annotating the guard on each message.

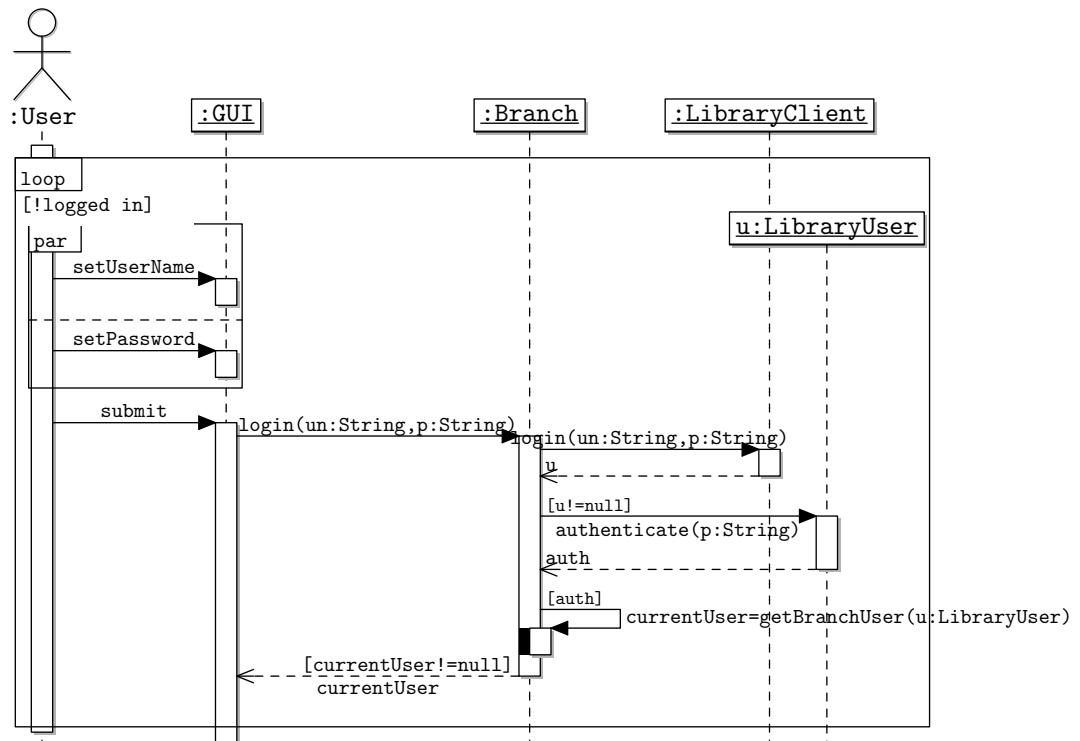


Figure 15.19: the revised login sequence diagram

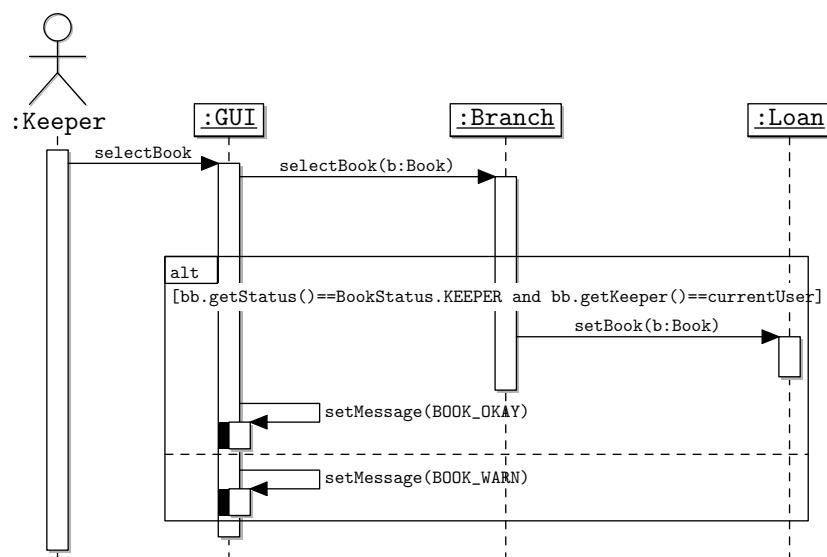


Figure 15.20: the selectBook sequence

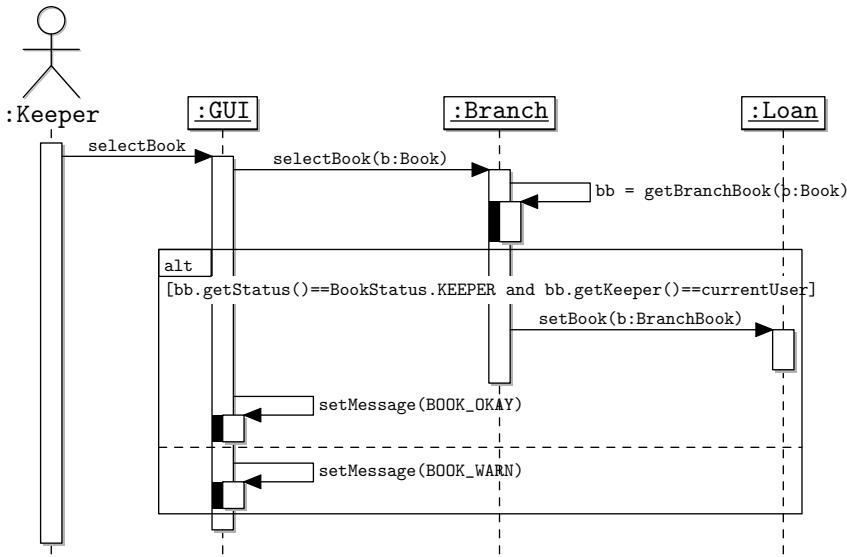


Figure 15.21: the revised `selectBook` sequence

Unfortunately, the sequence diagram introduces a similar problem to the one we encountered when checking that the logged in user was a keeper. The documentation for the `LibraryBook` class shows that the central library does not provide functionality for tracking the status of books in the branch or who the book's keeper is, so we will have to implement this ourselves as well. We can remedy the situation in exactly the same way as for the `BranchUser` by introducing a `BranchBook` class. When selecting a `LibraryBook` to be lent out, the `Branch` instance will first search for the equivalent `BranchBook` in its records. Figure 15.21 shows this revised sequence diagram.

This raises another question concerning the requirements:

- should book search results be optionally filtered by keeper and/or status?

Revising the requirements specification (413)

Doing this would save the keeper from having to try each book returned in a search. At the very least we should add a note to the search books use case description indicating that this would be desirable. We can now see how requirements specification may be altered as the design of a system proceeds, as discussed in Chapter 3.

We are now ready to complete the `lend books` sequence diagram by filling in the detail for the `approveLoan` message sequence. Figure 15.22 shows the complete sequence diagram for lending books, including all the sequence fragments we have developed so far. To complete the loan sequence the return due date is added to the `Loan` instance.

We can now represent the collection of messages that we have developed on a communication diagram. Figure 15.23 illustrates the communication diagram for all the sequence diagrams developed so far. Notice that we have aggregated

Complete sequence diagram (414)

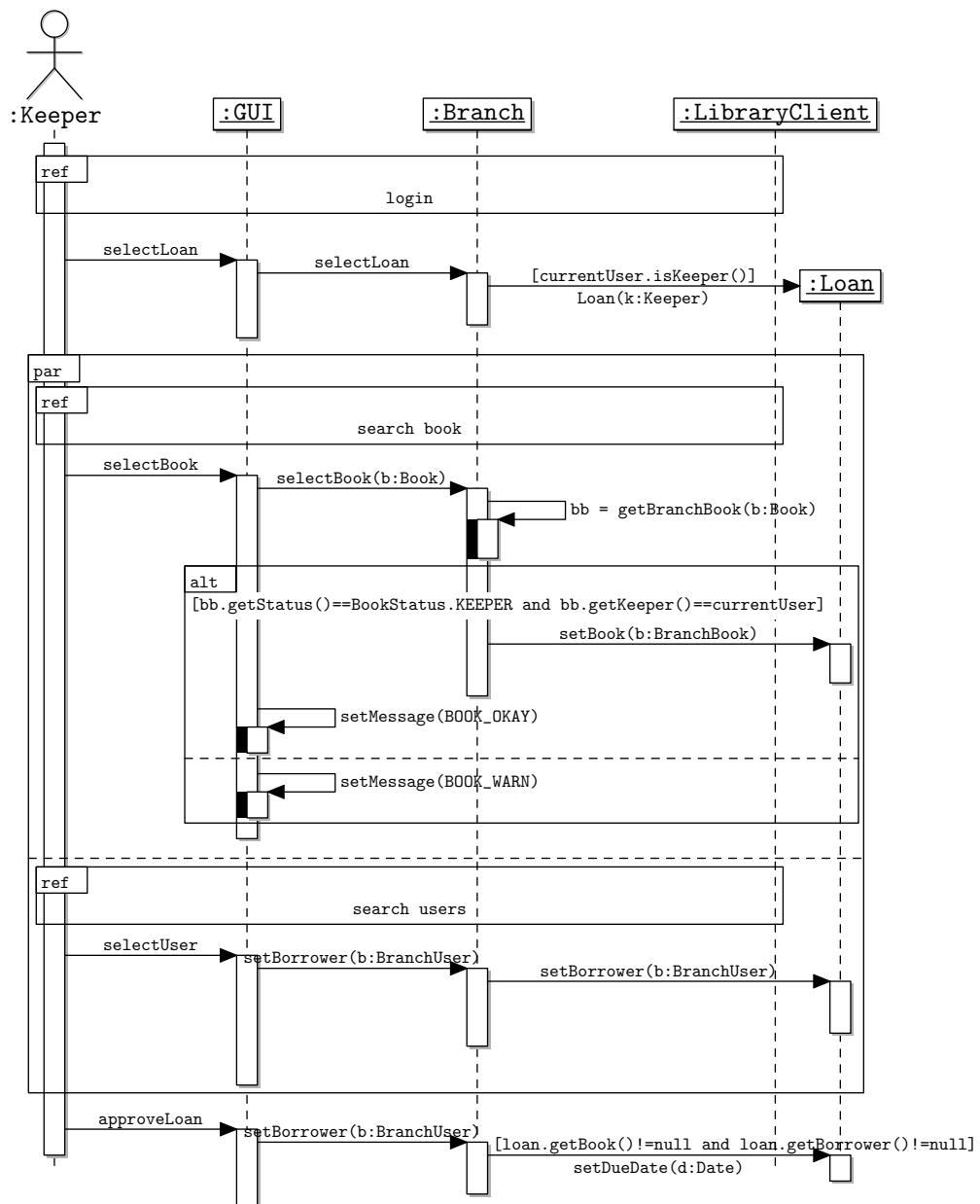


Figure 15.22: the complete lend book sequence diagram

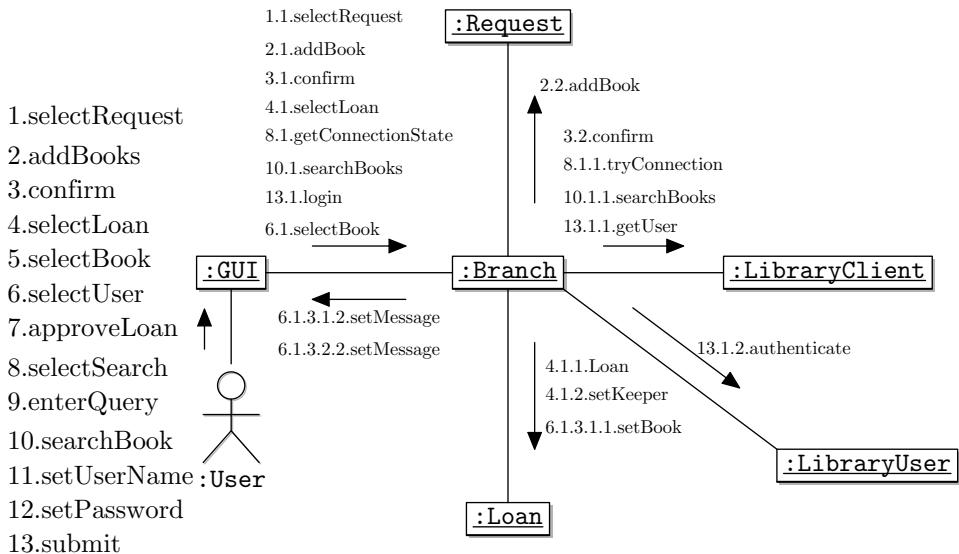


Figure 15.23: extended communication diagram

all the messages from different sequence diagrams. The communication diagram shows the general flow of messages between instances.

Notice that multiple sequences can be collated onto a single communication diagram to show the overall structure of collaborations in the system. Collating all sequences onto a single diagram is a good way of:

- revealing the overall set of pairwise collaborations between instances, giving an indication of the relationships between components and the extent of *coupling* in the system. We can see from Figure 15.23 that the Branch instance has been revealed as a central component in the architecture of our system;
- identifying components that may need to be divided into several smaller components during structural design activities. Instances that receive a large number of messages for different use cases should probably be broken down into smaller components later in the design process.

Note that collation onto a single diagram also tends to obscure the structure of message flow. This isn't necessarily a bad thing, since sequence diagrams are better at representing message flow.

We've added quite a few new instance classes to the system design, which need to be added to the class diagram we began in Figure 15.14. In addition, some instance classes have been replaced, as we have discovered more about the functionality provided by the LibraryClient component. Figure 15.24 illustrates the extended class diagram for the use cases we have investigated so far. The diagram shows the messages identified on the communication diagram as

Communication diagram (415)

Refine the class structure and detail (416)

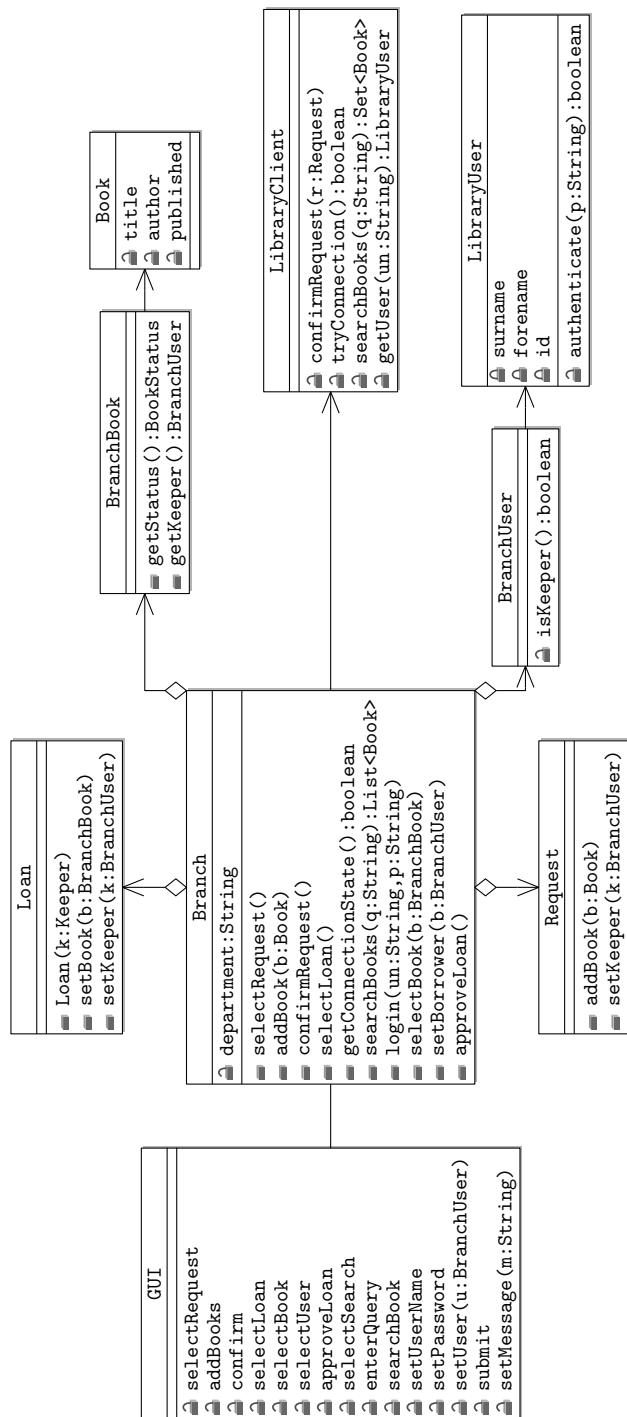


Figure 15.24: revised class diagram extended with method names

methods of the receiving classes. We've also identified the associations between classes, in particular the relationship between the Branch and the collections of BranchBook, BranchUser and Loan instances it maintains.

Despite the work we have done, the architecture of the system remains incomplete and relatively unrefined. Sequence diagrams need to be developed for the other use cases we have identified. The next chapter covers methods for improving the initial structure of the system developed so far.

## Summary

We have reviewed the use of interaction diagrams for modelling the dynamic behaviour of object oriented systems. Sequence and communication diagrams together can be used during the transition from the elaboration to construction phase of an RUP iteration. Following the method we reviewed above allows us to derive system architecture in the form of a class diagram from a system specification in the form of use diagrams and descriptions. Remember, use cases and activity diagrams describe the dynamic behaviour of actors *with* systems. Sequence and communication diagrams describe dynamic behaviour *within* systems as a result of external interaction.

## 15.5 Exercises

1. Recall the scenario described in Figure 11.16, describing a system for managing museum tours. Your tutor can provide you with a sample solution of the use cases and a domain model developed using the CRC method.

Using Umbrello or StarUML, construct a:

- a sequence diagram and a communication diagram to show an implementation of `create tour`; and
- a sequence diagram and a communication diagram to show an implementation of `assign guide to tour`, assuming that at least one guide is available to take the specified tour

by following the method described in this chapter for constructing interaction diagrams from use cases. Remember - the construction of sequence diagrams may reveal necessary changes to the use cases.

# Chapter 16

# Prototyping

## Recommended Reading

PLACE HOLDER FOR sommerville07software

### 16.1 Introduction

Prototype development typically occurs during the early stages of a software development project (during elaboration and construction in RUP), when there is uncertainty associated with some aspect of a system's requirements or design. *Prototypes* are small projects within a longer term software development process, which implements a part of the complete functionality required for the final system. Prototypes typically need to be developed and evaluated rapidly, so the resulting build quality and broader non-functional requirements are relaxed. The outcomes of the project are usually (but not always) not the prototype itself, but a better understanding of the system to be delivered.

What is a prototype?  
(420)

### 16.2 Why prototype?

There are several related reasons for producing a prototype, listed in Table 16.1. In each case, the underlying reason for developing a prototype is to expend a relatively small amount of effort in order to manage risk associated with a project caused by some uncertainty.

Why prototype? (421)

Developing a demonstrator can be useful in assessing whether a proposed business case really does offer value to an organisation.

Using prototypes during walkthroughs can help resolve ambiguity or inconsistency in requirements demanded by one or more stakeholders. As discussed in Chapter 11, customers often find discussions of how a system should work easier than a more abstract discussion of what a system should do. When requirements or design solutions are proposed it may be difficult to determine

---

<b>as a demonstrator</b>	a prototype can be used during the early stages of a project (initiation in RUP) to support a proposed business case
<b>support requirements gathering</b>	prototypes, particularly of user interfaces, can be incorporated into walkthroughs of scenarios. Customers and users can interact with a proposed solution, identify deficiencies and propose changes
<b>investigate the feasibility of requirements or design</b>	a prototype can be helpful in establishing the feasibility of requirements or proposed design solutions. A rapidly developed prototype may be useful for showing that a particular requirement can or cannot be achieved before too many resources are invested in the main development
<b>experiment with new technologies</b>	a prototype can be useful for assessing the feasibility of adopting a new technology to solve a part of the design problem for a system, or to understand a problem with an unfamiliar technology that has already been incorporated into the system
<b>as the main development for a project</b>	successive prototypes may be the central focus of a software development process, if many parts of the project are poorly defined

---

Table 16.1: purposes of prototyping

whether they are feasible, so a prototype can be developed before excessive resources are committed to an infeasible solution.

Prototypes are also useful for experimenting with new technologies to be incorporated in a project. The experiment can take the form of an investigation of the features and capabilities of competing equivalent products, or in order to resolve a problem concerning an already selected technology. In both cases, developing a prototype should reduce the uncertainty of working with the new technology without expending excessive resources.

Finally, prototyping may be adopted as the core software development process for the main product itself. This approach carries considerable risk, but may be necessary when the scope of the intended project is intrinsically uncertain.

A prototype can be developed to address more than one type of uncertainty at a time. For example, we could develop a prototype featuring the following subset of use cases for the branch library system: request book, process request, record book receipt. The prototype could then be used to:

- check that the branch library system will be of benefit to members of staff in the local departments (demonstration);
- validate the revised approach for managing books in a distributed branch with the central library;
- evaluate a web-based platform for hosting the system (evaluation); and
- as an initial iteration of the project.

Example: branch library system (422)

### 16.3 The Prototyping Process

Figure 16.1 illustrates the general process for developing a prototype alongside a longer term project. The overall intention is that some aspect of the main project should move from being assessed as high to low risk. Initially, a high risk aspect of the project is identified which represents some uncertainty concerning the intended systems requirements or design. A decision is made to investigate the uncertainty by developing a prototype, leading to an initial set of objectives being identified. The scope of the prototype (how much of the features of the final system will be implemented) is then identified, within the context of the overall budget for the project. The scope of the prototype should be constrained to minimise the resources expended on development.

The prototyping process (423)

Prototype design and implementation occurs rapidly, with less consideration for software architecture. Once the prototype has been designed and implemented, it is evaluated to determine whether the objectives have been satisfied. This may lead to the prototype or scope being altered, or alternatively, the objectives themselves may be revised (uncertainty from the project will affect all three aspects). Finally, if the objectives are satisfied, a report is prepared describing

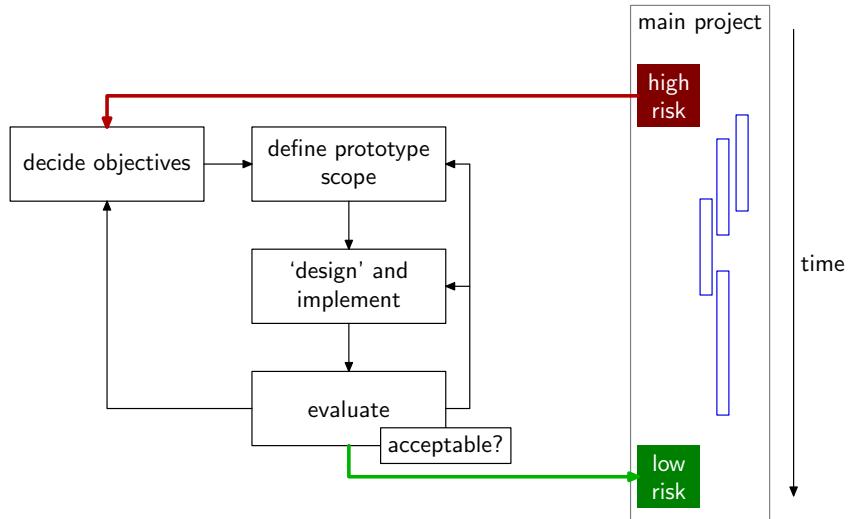


Figure 16.1: overview of the prototyping process

the outcomes of the project and how these address the original uncertainty that was identified as high risk.

## 16.4 Approaches to Prototyping

The specific types of prototyping are now discussed in more detail with respect to the general process in Figure 16.1.

### 16.4.1 Throw-Away Prototypes

Throw away prototypes are developed in order to resolve uncertainties about some aspect of the project, for example:

- what should the requirements for the system be?
- is the current requirements set feasible within the current budget and schedule?
- how should the system be designed?

Figure 16.2 illustrates the throw-away prototyping process. The objective for a throw-away prototype is to resolve uncertainty concerning the project, so the first step is to formulate the questions to be answered. The prototype is scoped, designed and implemented as before. The prototype is evaluated with respect to the original questions asked. If the uncertainties have been resolved, a report is prepared describing the lessons learned during the process and the implications for the main project. Alternatively, the outcome of the evaluation

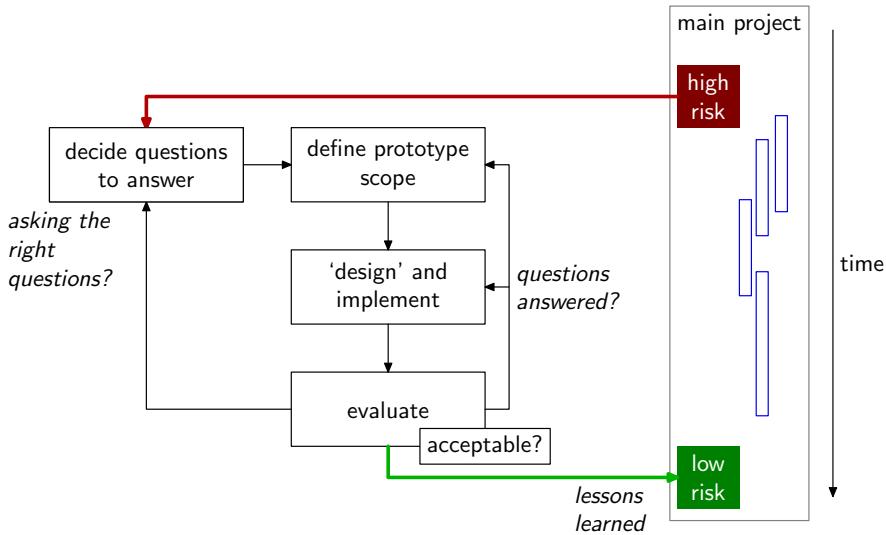


Figure 16.2: overview of the throw away prototyping process

may show that the uncertainty has not been satisfactorily addressed, so further prototyping may be undertaken, or the questions to be answered may be revised.

Prototypes can be a useful focus of discussion during scenario walkthroughs with clients, making abstract questions concerning the system requirements more concrete. Clients often have a pre-conception of what sort of system is to be built, as this is easier to formulate than the exact problem they expect the system to solve. Developing a prototype can help the client make these ideas explicit, by presenting them with a concrete example of a solution and then asking them to describe how they would like to change it. Abstract questions such as:

“What is the workflow for this use case?”

can be rephrased as:

“Is the workflow like this? How do you want to change it”

Throw-away  
prototyping process  
(424)

Prototypes in  
requirements  
gathering (425)

It isn’t necessary to present a complete system during requirements gathering, with the focus instead being on the functionality relevant to the questions to be explored. Functionality can even be ‘faked’ by, for example, only providing output for the inputs specified in the documented scenarios. Alternatively, a ‘Wizard of Oz’ interaction can be adopted, in which the inputs are processed manually and passed back into the user interface. Care should be taken not to ‘over sell’ the capabilities of the final system to the client based on the apparent features of the prototype. In particular, avoid including features that won’t appear in the next iteration of the final system.

A throw away prototype should not normally be used in a production system, because:

- the ‘design’ will not have been developed with consideration for the integrating with wider system;
- the architecture will become ‘hacked’ in order to demonstrate extra features during revisions;
- system non-functional requirements will have been relaxed, or even completely ignored;
- wider project quality assurance standards and conventions will have been relaxed;
- the design will not have been developed for re-use in other contexts; and
- documentation will be of low quality, or non-existent.

Instead, prototypes are often re-developed once criteria is established, with the lessons learnt from the throw-away prototype applied to the component of the final system. Outputs from prototyping should still be retained as a part of the documentation for the main software project to support traceability of requirements. The outcomes of a prototyping project represent the justification for revising the requirements specification or a design decision.

Prototypes can also be useful during testing of components or systems with well defined inputs or outputs, particularly if a *test suite* is developed for the prototype. In principle a production system should have the same outputs as the earlier prototype for any given input, so a prototype can be used as a *test oracle* for the production system (sometimes called a *reference implementation*).

Figure 16.3 illustrates this process. Input from the prototype test set is passed to both the prototype and the production system. These resulting outputs are then compared and a report is produced describing any differences. Differences between outputs between the two implementations should be investigated and resolved - the cause could be a defect in either implementation.

### 16.4.2 Evaluating New Technology

As discussed in the previous section, many software products are based on existing frameworks which provide the core functionality, encouraging software re-use. Table 16.2 describes a number of popular frameworks (the range of frameworks is constantly evolving).

The use of new or unfamiliar (to the development team) technologies represents two sources of uncertainty (and hence risk) to a software project:

Throw-away  
Prototypes in Testing  
(427)

Evaluating new  
technology (428)

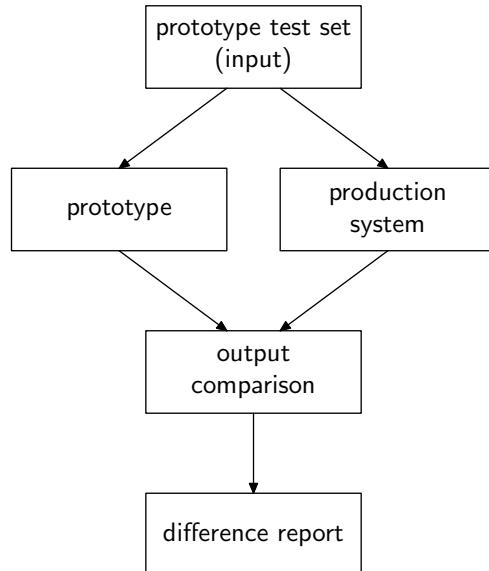


Figure 16.3: using prototypes during testing

---

<b>Common Object Request Broker Architecture (CORBA)</b>	is a specification for messaging passing between distributed services, implemented in a number of languages including Java and C.
<b>Knopflerfish</b>	implements the OSGi component management specification
<b>Apache Axis</b>	a suite of tools and libraries for implementing web services as specified by the Web Service Definition Language(WSDL), including a SOAP <sup>2</sup> protocol library
<b>Globus</b>	developed as a toolkit for deploying Grid computing applications. The fourth and fifth versions of globus Web Service Resource Framework (WSRF)
<b>Java Messaging Service</b>	communication occurs between components by allocating messages to queues, which are then accessed and processed by other components

---

Table 16.2: a selection of software frameworks

- it can be difficult to select an appropriate framework from a range of competing products, each developed around a number of (often implicit and hidden) assumptions;
- it may be unclear how to implement a particular feature or technique in a chosen software project, or resolve a deficiency identified with the framework.

A prototype development which uses the new technology can help address both of these issues, without diverting too many resources from the main development effort. A prototype project can be used:

- to evaluate a number of competing products, by implementing a small number of key features for the main system on each of the candidate platforms
- to isolate and resolve defects in a software system; and
- for developer training.

Evaluating new technology (429)  
Figure 16.4 illustrates an example of using a prototype to resolve a defect identified during the development of the `BranchClient` component, supplied for the branch library system. The defect was identified during preliminary testing of the search books use case. Some documents are held electronically by the library (PhD and MRes theses, for example), so it was decided these would be directly retrievable via the branch library system. The electronic documents are stored in a MySQL<sup>3</sup> database. A developer noticed that not all of the electronic documents were being downloaded to the user's machine.

The development process follows the same structure as for other prototype projects. To identify the cause of the defect, a prototype was developed which uses a shell client to the DBMS to upload a series of files directly (isolating the DBMS from the branch application). Inspection of the database and the DBMS logs revealed that some of the files were not uploaded because they are bigger than the default maximum packet size for the database (16mb). To remedy the problem, a recommendation was made that the maximum packet size is doubled.

#### 16.4.3 Proof of Concept

A prototype can also be useful for demonstrating the feasibility of a new concept or idea. Proof of concept demonstrators are often used to investigate speculative features for a new system or as part of research projects. The outcomes from the process provide insights into the benefits and risks of a potential investment during the early stages (initiation phase in RUP) of a software development project.

---

<sup>3</sup><http://www.mysql.com>

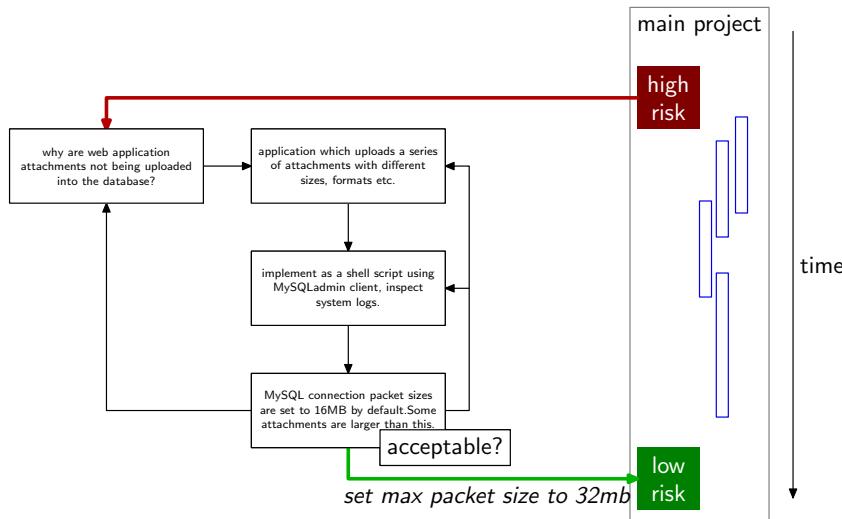


Figure 16.4: prototyping process for evaluating new technology

In some cases, the prototype can provide a basis for further development. For example, the management of the branch library system project might choose to deploy a single branch library system in one department, before committing all the resources necessary for a university wide system.

#### 16.4.4 User Interface Prototyping

User interfaces present a particular problem for requirements gathering because the specification for the design can be difficult to express. In addition, the development of rich user interfaces can be extremely time consuming for a development project<sup>4</sup>.

Prototyping of user interfaces is a natural solution to this problem, and it can also be used to represent the functionality of the underlying system. There are a variety of CASE tools available for developing a prototype user interface, particularly graphical user interfaces, such as the NetBeans IDE<sup>5</sup> for developing Java swing user interfaces, Microsoft's Visual Basic programming language<sup>6</sup> for developing Windows applications and Adobe's Dreamweaver<sup>7</sup> for web applications. A screen shot from a NetBeans development in shown in Figure 16.5.

Developing a graphical user interface using a CASE tool may mean that the client is able to make changes to the layout and functionality themselves. Recording these changes can help to understand the client's *mental model* of the

CASE Tools for UI prototyping (430)

<sup>4</sup>in my (highly limited, subjective) experience of projects it can consume well over 50% of resources

<sup>5</sup><http://www.netbeans.org>

<sup>6</sup><http://msdn.microsoft.com/en-us/vbasic>

<sup>7</sup><http://www.adobe.com/products/dreamweaver>

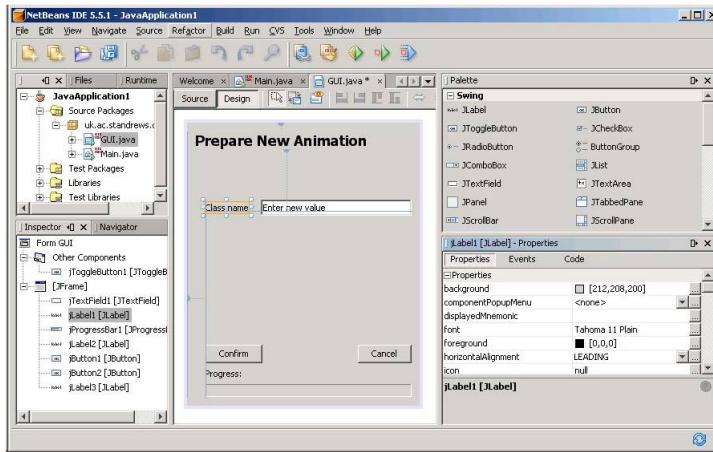


Figure 16.5: the netbeans GUI designer

underlying functionality of the system, because the changes may be a reflection of how the client believes the system functions, or should function.

Depending on the CASE tool, the prototype user interface may eventually be developed into the user interface for the final product. However, using CASE tools to develop user interfaces *can* mean that the user interface code is poorly structured and doesn't relate well to the functions of the underlying system. This can make maintenance costs for alterations to the user interface more costly later on in a project.

#### 16.4.5 Rapid Application Development (RAD)

Many business processes are primarily collecting, processing and analysing data. Automation of such processes often results in an *information system*<sup>8</sup> with a recognisable structure:

- a *data model* of the information to be managed;
- a *functional layer* for editing data and generating *reports*; and
- a *user interface* for entering data and presenting reports.

The common architecture of this class of systems mean that much of the core functionality can be pre-defined in a software framework or platform, consisting of:

- a database management system (DBMS) configured for the required data model, and supporting a query language, typically some flavour of Structured Query Language (SQL);

---

<sup>8</sup>there are far broader definitions of information systems which encompass pre-digital electronic information processing technology.

- a *report generator* links to office applications and CASE tools
- a GUI builder for designing and implementing the user interface; and
- a fourth generation programming language (4GL) for constructing applications.

Rapid Application Development (RAD) is a software development method which exploits the characteristic features of information systems in order to reduce software development effort Martin [1991]. The system is developed iteratively as a series of prototypes. The system is within a software framework, so the overall amount of source code to be written should be reduced because much of the core functionality is pre-provided. In addition, RAD encourages software re-use (and design for re-usability) so new components developed for one project should be available for others.

Since much of the core functionality of the system is pre-determined, requirements processes for RAD instead focus on:

- understanding current business processes;
- establishing the data model for the underlying system;
- the information needs of the various users (and who supplies it); and
- the necessary organisational changes to fit the new system.

More recently, *Enterprise Resource Planning* (ERP) platforms such as SAP<sup>9</sup>, Oracle E-business Suite<sup>10</sup> or SAGE<sup>11</sup> and web application frameworks such as Ruby on Rails<sup>12</sup> (for the Ruby language) or CakePHP<sup>13</sup> have been promoted for developing applications in a similar way to RAD. An ERP framework vendor may also offer consultancy services for the configuration of the framework for a client's application, rather than just selling the framework itself. The vendor may also offer to train and accredit developers to work on their framework with other clients.

Since much of the functionality for these systems has already been implemented by the vendor, development focuses on configuring and tuning the selected components for the target organisation. The term *configuration engineering* or *construction by configuration* has been used to describe this process Sommerville [2005].

The constraints imposed by RAD methods means that there are several trade-offs for development and associated risks:

---

<sup>9</sup><http://www.sap.com>

<sup>10</sup><http://www.oracle.com/us/products/applications/ebusiness>

<sup>11</sup><http://www.sage.com/>

<sup>12</sup><http://www.rubyonrails.org/>

<sup>13</sup><http://www.cakephp.org/>

- code is cheaper to development, but may be less efficient. For example, the framework may not implement the most efficient algorithms for performing a pre-defined processing task;
- functionality is quicker to generate, but underlying code may be poorly structured, increasing maintenance costs;
- re-engineering applications for a different platform may present challenges. This can cause vendor *lock in* for the client, preventing effective competition for services; and
- an organisation may struggle to adapt to the new system; Frameworks have to be designed with at least some minimal assumptions about how any organisation functions. These assumptions may not fit the culture of a particular organisation.

#### **16.4.6 Incremental Prototypes**

In some cases, the requirements for a system may be so poorly understood that the only way to make any progress is to adopt prototyping as the main focus of development effort. This approach is common in many research environments, where the expected result of the development activity is unknown and the primary outcome is new knowledge, rather than the software itself. In these situations, the software is developed in a number of prototype iterations, and the specification is refined gradually each time as the requirements become better understood.

Incremental prototyping does carry risks, not least because it can be used as a justification for low quality software development processes. A consequence of a continually changing application is that the overall structure may not be maintained, making long term maintenance and future changes more costly, as well as making non-functional requirements harder to implement and defects harder to identify and remove.

In addition, as the software base becomes larger and more complex, it may be necessary to initiate a number of iterations in parallel, making change management more complex. As a consequence, documentation and other accompanying artefact's (such as test suites) may not be maintained, or abandoned all together. Eventually, the entire system may need to be re-implemented anyway, costing much of the supposedly saved effort and resources.

Many *agile* software development methods advocate disciplined software practices to address the risks associated with software quality of incremental prototyping. Section 4.7 describes the Extreme Programming software development process and associated practices.

	<b>requirements</b>	<b>outcomes</b>
<b>throw-away</b>	high risk, poorly understood requirements	executable prototype; better understanding of requirements
<b>incremental</b>	well understood, low risk requirements	application for end user

Figure 16.6: throw-away vs incremental prototyping

## Summary

Software prototyping can be used for a variety of purposes in a software development process, from resolving a short term uncertainty to adoption as the main development activity itself. Figure 16.6 contrasts these two extremes in terms of the approach to risk management and the resulting outcomes.

Throw-away prototyping addresses the highest risk, poorly understood requirements for a software project first, with better understood requirements implemented in the main software development process. The outcome is potentially a executable prototype, but more importantly the lessons learned for the prototyping process. Incremental prototyping addresses the well understood requirements for a software project *first*, since these should be the cheapest to implement in the initial version. Higher risk requirements should become better understood ,as more functionality is added to the prototype in successive iterations. The outcome of incremental prototyping is the system to be delivered to the end user.

Prototyping approaches can also be differentiated by their longevity, as illustrated in Figure 16.7. The longevity of the prototype should be reflected in the effort and resources devoted to the quality during the prototyping process.

Throw-away prototypes have very short life-cycles as the prototypes themselves are far less important than the lessons learned from the prototyping process. Demonstrators are often anticipated to be short-lived artefact's, but may

Throw-away vs.  
Incremental (431)

Longevity of  
prototypes (432)

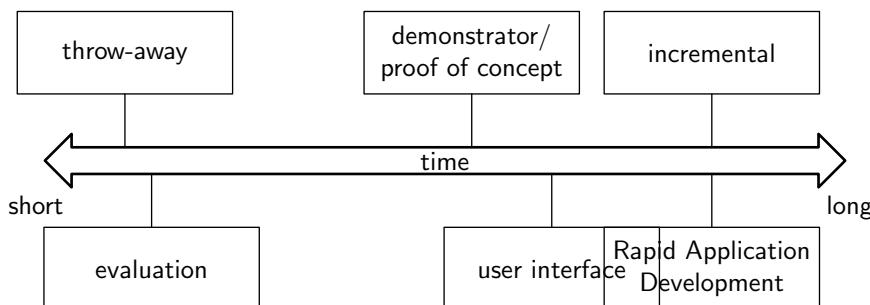


Figure 16.7: longevity of different types of prototype

Key Point (433)

go on to be part of a software project for a period of time, at least until there is opportunity to re-implement the software following more rigorous development standards. The first steps during elaboration and construction phases of a project based on a demonstrator may be to engage in a period of 'cleaning up' the architecture and code base before further features are added.

User interfaces can be developed most effectively with the involvement of the client as a series of prototypes. Finally rapidly developed incremental prototypes form the basis of the main software project, so last as long as it does. This approach to software development implies considerable risk, that can be managed to some extent by adopting disciplined practices from agile methods.

# **Part III**

# **Software Design, Implementation and Testing**



## Chapter 17

# Software Re-use and Frameworks

### Recommended Reading

**PLACE HOLDER FOR lethbridge05object**

Recommended reading  
(434)

**PLACE HOLDER FOR cox90planning**

The Apache Axis webpage:

<http://ws.apache.org/axis>

This lecture uses material originally derived from:

<http://zetcode.com/tutorials/javaswingtutorial/>

A typical computer, be it a general purpose desktop or laptop, a server within a large cluster, or embedded in a mobile device or washing machine will be running a variety of software programs that are also used elsewhere on other platforms. In addition, many applications incorporate, or depend on software components provided by other development projects to function. It is *extremely* rare for all of the software run by a computing platform to be custom developed for that purpose alone, or even for that particular hardware model.

Software re-use, then, is endemic in software development. Sometimes, parts of a larger system development are partitioned into a separate development activity, because it is realised that it represents features can be utilised by other systems. Alternatively, a software system can be deliberately designed to be used as part of a larger application, and not intended to function as an application by itself. The development priorities for such software is consequently different from that for end-user applications: the user of the software will be other software engineers.

Although beneficial, re-use often results in compromises in system features during the design process, and imposes additional responsibilities on those who

provide software for re-use. This chapter introduces some general concepts and features of software re-use and looks at two particular frameworks: the Java Swing GUI Toolkit and the Apache Axis web services middleware framework.

## 17.1 Re-usable Software Components

The concept of components was introduced in Chapter 19, as software artifacts specified in a system design, with well defined Application Programming Interfaces (APIs). Multiple components can be combined to provide larger applications for an end-user as part of the software design process. Software components are portrayed in component diagrams in the UML. These diagrams illustrate the relationships between persistent components with a long life time within the overall application design.

*Re-usable components* are software components that are intended to be incorporated in more than one software applications. A re-usable component implementation may be instantiated as several different runtime components, each combined in a separate application; or the same runtime component can be simultaneously incorporated in several different applications.

Re-usable components can be implemented in a homogenous component management environment, such as the Open Services Gateway initiative (OSGi)<sup>1</sup> or the Axis web service container (see Section 17.5). These environments provide life-cycle management, functionality discovery and orchestration facilities for hosted components.

A *software library* is a language specific collection of related re-usable software modules (routines, classes, resources and so on). Many development environments and programming languages specify formats for creating software libraries and mechanisms for integrating them into a large application. Libraries can be integrated into an application:

- *statically* during program compilation by a *linker*; or
- *dynamically* when the library is needed by the application at runtime, by a *loader*.

If a library is integrated into an application during compilation, it can result in a larger than necessary program binary (since the binary will contain a fresh compiled version of the library). However, it is harder to check a program for defects if integration only occurs at runtime.

Examples of library formats include:

- Dynamic Link Libraries (.dll extension files) on Windows platforms;
- the Java Archive format (.jar file extension), which is actually, a zip format file containing the classes of the library and classpath mechanisms for the Java Runtime Environment; and

---

<sup>1</sup><http://www.osgi.org>

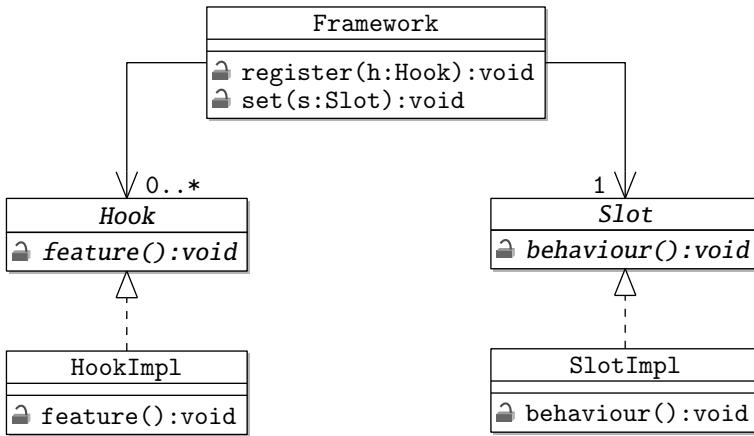


Figure 17.1: slots and hooks in frameworks

- the shared objects mechanism for \*nix platforms.

A library will typically contain implementations for a range of classes, functions or components of related purposes.

A *software toolkit* is a software library containing a collection of related components and software tools for building a particular type of application. A toolkit may contain re-usable software components, as well as additional tools to assist the development effort (but which won't be included in the final application). The Java Swing toolkit, for example, contains components and tools for building graphical user interfaces in Java.

Software toolkits  
(438)

A *software framework* is a re-usable, but incomplete software component, which provides much of the functionality required for a common software task. A software engineer provides the missing functionality for the framework to configure it for a particular context of use. A framework is often implemented as one or more software libraries.

Software frameworks  
(439)

A framework can contain:

- *slots* which specify missing functionality that must be implemented by a developer for the framework to function; and
- *hooks* defining optional extensions for a framework. A framework will continue to function whether a hook is used or not.

Slots and hooks (440)

Figure 17.1 illustrates the distinction between hooks and slots in a framework (note that there are actually several different ways of implementing hooks and slots in object oriented systems). In the figure, a hook is an interface specification. Zero or more implementations of the hook can be *registered* with the framework. The framework will invoke the hook's feature as appropriate. The framework also specifies a slot which must be filled by exactly one implementation. The slot defines part of the core *behaviour* of the instantiated framework – the component framework will not function without it.

## 17.2 Choosing and Deploying Re-usable Software

Examples of re-usable software (441) There are numerous examples of software developed for re-use. A small selection includes:

*information management system frameworks*, which can be configured to manage a variety of organisational processes in a common information system architecture. The Linux, Apache, MySQL, PHP (LAMP) pre-bundled operating system images provide the components necessary for a web-based information system. The Microsoft Access database management system provides features for developing applications on top of a Microsoft SQL database (typically the Jet engine);

*enterprise resource planning (ERP) software*, which is used to translate business process models of an organisation into an information system. Specialist ERP software or components is available for particular problem domains, such as payroll management (e.g from SAGE), and university administration (the Student Lifecycle Project at Glasgow is based on a platform called Campus Solutions);

*logging frameworks*, for management of de-bugging and other non-user output from an application. See appendix 29.6 for information about the log4j logging framework;

*test harness frameworks*, for managing the development of test suites. See Chapter 21 for information on test harnesses and the JUnit test suite framework;

*server application frameworks*, which provide input/output and application life-cycle management, allowing developers to concentrate on the business logic of an application. An example is the Tomcat application container, which supports applications developed for a variety of communication standards;

*General Circulation Models (GCMs)* in climate science are often configured, parameterised, adapted or incorporated into larger models by climate scientists. A model is a particular software implementation in this field;

*information retrieval engines* such as Terrier and PuppyIR developed at Glasgow. These engines allow users to specify data formats and search algorithms without needing to provide the low level implementation details;

*3D modelling and rendering toolkits*, such as OpenGL<sup>2</sup> implementations and Java3D<sup>3</sup>. These toolkits provide high level application programming

---

<sup>2</sup>[www.opengl.org](http://www.opengl.org)

<sup>3</sup>[www.java3d.org/](http://www.java3d.org/)

interfaces for manipulating objects in 3D space, without needing to fully understand the mathematics involved;

*software monitoring frameworks* provide features for obtaining runtime information about software applications. The Java Platform Debug Architecture<sup>4</sup> is a framework for building applications which track information from a Java Runtime Environment (JRE) virtual machine;

*software modelling frameworks*, such as the Eclipse Modelling Framework. Modelling frameworks provide the basis for developing modelling tools for particular languages, such as UML. The technique is often called meta-modelling, since models are developed which specify the rules of the language and the appearance of elements of the language as graphical artifacts;

*middleware frameworks* such as Axis for web service platforms (see Section 17.5), and the Java Messaging Service; and

*software as a platform (SaaP) cloud computing* provides frameworks for developing applications for deployment within cloud environments.

Wherever possible, it is generally preferable to re-use software provided by somebody else. There are several reasons that re-use is beneficial for a project:

- avoid wasting resources on expensive developments;
- utilise the experience and expertise of domain experts;
- improved software design for end-user applications; and
- concentrate resources on improving the maturity of widely used software.

Why re-use?  
Engineering vs. craft  
(442)

Lethbridge and Laganière [2005] argue that the repetitive development of the same features for different applications is *craft* rather than engineering. In a craft, a worker applies a range of specialist skills to create a product type from raw materials. You can see examples of craftsmen at work creating large tapestries at Stirling Castle, or on the Isle of Harris in the Outer Hebrides creating Tweed.

The craftsman will acquire their skills over a long period of time, often serving an apprenticeship with a master. Each craftsman will develop their own style of work, and 'way of doing things' which will leave tell-tale marks in the resulting product. Think of stone-masons who left their 'signatures' hidden on their works in cathedrals or castles, allowing historians to match work from place to place to the same craftsman. This approach to production can result in very high quality work. However, the resulting product will often be expensive, take a very long time to make and contain considerable variation.

---

<sup>4</sup><http://java.sun.com/javase/technologies/core/toolsapis/jpda>

## Component oriented engineering (443)

In contrast, engineers are concerned with the application of scientific principles in the construction of products. Engineers seek to select and use standardised methods, practices and tools to reduce the risks of a development project. An engineer seeks to identify and apply known solutions to well defined problems, so if a standard solution already exists to a problem, that should be used in preference to re-developing a solution from raw materials.

In *component oriented software engineering* much of the development work is concerned with selecting and orchestrating re-usable components for a particular application or task. There are several activities involved in component system engineering:

- Re-usable component evaluation and selection. This involves the identification and comparison of competing components. A common technique for evaluating components is to construct a *prototype* of the final system which can be used to determine whether the component will be suitable;
- component orchestration. This is sometimes informally called *doing the plumbing*, since the task is analogous to connecting up the major components in a water system (baths, toilets, sinks for example) using a variety of different pipes; and
- developing new components for reuse. This is sometimes necessary if it is not possible to identify a suitable component for a particular task. A common example is when it is necessary to orchestrate several re-used components to complete a particular task.

## Criteria for re-usable components (444)

There are several criteria that govern the selection of components for re-use by an engineer:

- the functional and architectural compatibility of the component and the proposed system. Developers of re-usable components must make design decisions, in particular concerning the components API and architecture, just like those working on end-user projects. These decisions impose constraints in the developers of systems who seek to re-use components. The adapter design pattern (Section 18.2.3) can be used to wrap the functionality of a re-used component to make it compatible with other systems. However, Garlan et al. [1995] have noted that there are many (apparently trivial) situations where the underlying architecture of a component and the proposed system make integration challenging;
- the cost of licencing and/or supporting the re-used software. As discussed in Section 1.4 There are numerous business models for developing software, and consequently a variety of budget models (how finance is made available to run a project). The way that re-used software must be pro-

cured, as much as the overall cost influences the component selection process<sup>5</sup>;

- the maturity of the re-used software. Re-using mature software reduces risks in software development, since in general:

- the architecture and functional API is more *stable*. Rapidly changing software imposes greater costs on system developments which use it, since the proposed system must be continually altered to manage the change. In mature software, the architecture and API change more slowly because the design has been demonstrated to satisfy the requirements specification, meet the needs of users *and* because users would resist dramatic change;
  - the overall quality of the software will be higher, because more defects will have been discovered and removed due to more extensive testing (either by the development team, or by the larger user base); and
  - the developing organisation of re-used will provide better support for the software in the form of documentation, or expert advice. When significant changes are introduced into the software, support is provided to help users migrate to the new version. The cost of providing this support can be spread amongst a larger number of customers.
- the flexibility and configurability has been improved to ease adaptation of the re-usable software to the needs of users. This is related to the maturity of the design, since cleanly separated designs tend to be more flexible and adaptable.

The emphasis on particular criteria will depend on the development project, the type of re-usable software to be selected and the characteristics of the available re-usable software components.

### 17.3 Developing re-usable software

There are several aspects to developing re-usable software:

- identifying the appropriate degree of flexibility and configurability in the component;
- maintain a dependent independent architecture;

Economics of software  
re-use (445)

---

<sup>5</sup>I've talked to several engineers who have reported difficulty in providing 'free' software as part of a larger system to the UK government, because it can't be budgeted for in the civil service's procurement process (I can't verify whether the stories are true).

- budgeting sufficient resources to support high software quality. Higher quality software encourages use by other developers; and
- providing incentives for others to develop re-usable components on your behalf.

A re-usable component needs designed to be sufficiently generic that it imposes minimal constraints on its use or the design of the surrounding system. Note that this is an extension of the separation of concerns design principle discussed in Chapter 19. A trade-off must be made between providing sufficiently generic behaviour and avoiding over complexity in the configuration process. If the software is not sufficiently flexible, it will have a limited user base, and insufficient resources to support future maintenance and improvements. However, if excessive flexibility is designed into the software, it may become too difficult for it to be configured to *any* specific instance. This is sometimes called the *inner-platform effect* anti-pattern.

Conversely, a re-usable software component may also suffer from 'dependent capture'. This occurs when the owner of a system that incorporates a framework or other re-usable software (i.e. is a dependent) is politically powerful within the re-usable software's development team. This can happen, for example, if the system dependent is a significant sponsor of the project team, by providing resources for maintenance. A risk is that the re-usable software will evolve in a way that fits with the architecture of the dominant dependent, and lose flexibility. Sometimes, a branch version of the re-usable software may be created to accommodate the specific requirements of the dependent. This causes additional maintenance costs for the re-usable software development team.

Software re-use requires additional investment in the software quality of the re-used components. Software projects specifically intended to develop re-usable software will prioritise support and documentation for other developers, rather than for end users. However, many re-used components begin their existence integrated into other systems. Consequently, it can be difficult to apply the appropriate quality standards early in a re-usable component's project life-cycle. As a software project matures and is incorporated in other applications, there is a responsibility on the development team to provide more detailed and complete documentation. When a component from this sort of origin is separated into a new project, a period of re-factoring may be necessary, before new features can be added.

Finally, the availability of high quality software, is dependent on the willingness of development teams to build software that will be used by other developers, rather than by end users. Economic incentives need to be created so that re-usable software developers are rewarded for their efforts. Cox [1990] argues that much of the so-called software crisis (see Section 1.1) can be attributed to the lack of incentives for software developers to produce standardised 'parts' (re-usable software components) for incorporation by others into software systems. Contrast software development, in which a development team work on

all the aspects of a project for a long period of time, with manufacturing processes for (for example) cars, tinned food, paperback books or standard size clothing. In manufacturing, standardised parts are assembled into sub-systems and pass down the supply line. These parts are then used as standard parts and assembled into larger sub-systems elsewhere. Each step in the assembly process has no need to know how the parts are made – the details are abstracted away. In software, these incentives need to be created artificially, through licencing arrangements or internal incentives within an organization.

## 17.4 The Java Swing GUI Framework

Swing is the official GUI framework for the Java language. The features of Swing include:

- a lightweight implementation, meaning that Swing is completely implemented in Java executed in the Java Virtual Machine. Consequently Swing doesn't rely on the execution of native code on the host platform, making Swing platform independent;
- membership of the Java Foundation Classes, which also includes the older Abstract Windows Toolkit (AWT);
- a customisable look-and-feel mechanism to allow Swing GUIs to mimic the appearance of host desktop user interface platforms; and
- full integration into the Java Standard Development Kit (SDK). Figure 17.2 summarizes the packages of the Swing framework.

Features of Swing  
(446)

The swing packages containing a rich collection of features for developing Java GUIs. The following sections provide a tour of the key features of Swing, highlighting where the framework can be extended and completed to provide a complete application. You should try the Java code for yourself to explore the use of the Swing toolkit. The complete Java code for the Swing classes is included in the package which accompanies these notes.

javax.swing	javax.swing.plaf.synth
javax.swing.border	javax.swing.table
javax.swing.colorchooser	javax.swing.text
javax.swing.event	javax.swing.text.html
javax.swing.filechooser	javax.swing.text.html.parser
javax.swing.plaf	javax.swing.text.rtf
javax.swing.plaf.basic	javax.swing.tree
javax.swing.plaf.metal	javax.swing.undo
javax.swing.plaf.multi	

Figure 17.2: packages in the Java Swing GUI framework

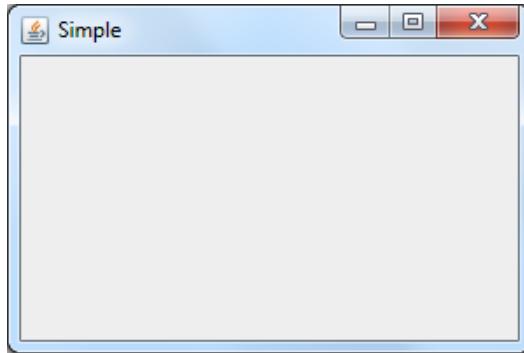


Figure 17.3: a basic swing window

#### 17.4.1 Basics

Basic Window (447)  
Simple (448)

A simple swing window is shown in Figure 17.3. The code for implements the Simple class is shown below.

```
package uk.ac.glasgow.oose2.swing;
import javax.swing.JFrame;

public class Simple extends JFrame {

    private static final long serialVersionUID =
        -1679421450260258244L;

    public Simple() {
        setSize(300, 200);
        setTitle("Simple");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }

    public static void main(String[] args) {
        Simple simple = new Simple();
        simple.setVisible(true);
    }
}
```

Even though the code is small, the application window can do quite a lot. It can be resized, maximized, minimized and closed. All of this complexity is hidden from the application programmer.

The first step is to import the `JFrame` widget class. `JFrame` is a top level window container, which is used for holding other widgets. By having `Simple` extend `JFrame`, we make `Simple` be such a container. The method calls inside the constructor create the initial appearance of the window:

- `setSize()` establishes the initial size of the window;
- `setTitle()` sets the string in the window's title bar; and

- `setDefaultCloseOperation()` causes the window to close if we click on the close button in the upper right hand corner. If we had not made this call, nothing would happen if we clicked on the close button

The code below shows how to use the toolkit to find information about the display and center the GUI window on the screen.

[CenterOnScreen \(449\)](#)

```
public class CenterOnScreen extends JFrame {

    private static final long serialVersionUID =
        -5066399896276680120L;

    public CenterOnScreen() {
        setSize(300, 200);
        setTitle("CenterOnScreen");
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        Toolkit toolkit = getToolkit();

        Dimension screenSize = toolkit.getScreenSize();
        int centerX = screenSize.width/2 - getWidth();
        int centerY = screenSize.height/2 - getHeight();
        setLocation(centerX, centerY);
    }

    public static void main(String[] args) {
        CenterOnScreen cos = new CenterOnScreen();
        cos.setVisible(true);
    }
}
```

To center the window, we need to know the resolution of the monitor; for this, we obtain an instance of the Toolkit class via the static `getToolkit()` method.

We determine the screen dimensions via the static `Toolkit.getScreenSize()` method. To place the window on the screen, we call the `setLocation()` method.

Figure 17.4 illustrates the addition of two buttons to the GUI. The buttons also have behaviours associated with them which are performed when the buttons are pressed. The code below shows how to add buttons to a GUI.

```
 JButton beep = new JButton("Beep");

 ActionListener beepAL = new ActionListener() {
    public void actionPerformed(ActionEvent event) {
        toolkit.beep();
    }
};
```

[Buttons and Events \(450\)](#)

[Creation and events \(451\)](#)

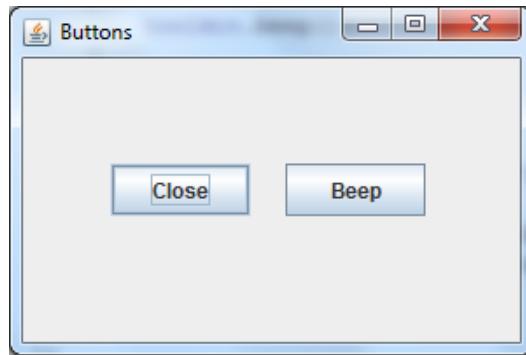


Figure 17.4: a window containing two buttons

```

beep.addActionListener(beepAL);

JButton close = new JButton("Close");

ActionListener closeAL = new ActionListener() {
    public void actionPerformed(ActionEvent event) {
        System.exit(0);
    }
};

```

We create two buttons using the `JButton()` constructor. In each case we construct a new instance, with a `String` label to appear on the button. Then we construct a new instance of the `ActionListener` interface by implementing the over-ridden `actionPerformed()` method *inline*. This means that the implementation of `ActionListener` declared in the method is *anonymous*, i.e. it has not type name.

In the first case, we specify that a beep sound should be emitted when the button is pressed and the `actionPerformed()` method is invoked, and in the second case, the application is closed. Each `ActionListener` is registered with the appropriate button, using the `addActionListener()` method.

Note that:

- an `ActionListener` is an example of an `Observer`; and
- a `Button` is an example of an `Observable`

from the Observer-Observable pattern (see Section 18.4.3).

Having created the buttons, they now need to be added to the `JFrame` container, as shown in the Java code below.

```

 JPanel panel = new JPanel();
 panel.setLayout(null);
 beep.setBounds(150, 60, 80, 30);
 panel.add(beep);

```

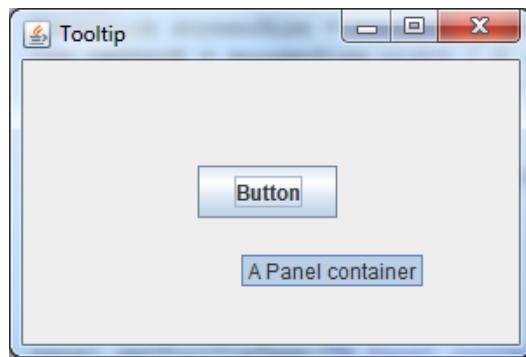


Figure 17.5: attaching a tool tip to a widget

```
close.setBounds(50, 60, 80, 30);
panel.add(close);
```

A new JPanel instance (a lightweight container) is created. The JPanel's existing layout policy for contained widgets is over-ridden by setting the JPanel layout manager to **null**. Next, the two buttons are added to the panel and set their absolute positions by specifying their *bounds* relative to the panel. Finally, the JPanel is added to the JFrame's *content* pane.

Many different swing components can have a *tool tip* set. Tool tips are short pieces of text containing helpful information describing the purpose/effect of a widget, or even part of the GUI. Figure 17.5 illustrates the appearance of tooltips in Swing. The Java code below shows how to implement tool tips for the buttons and panels created above, using the setToolTipText() method.

Tooltips (453)

```
JPanel panel = new JPanel();
getContentPane().add(panel);

panel.setLayout(null);
panel.setToolTipText("A Panel container");

JButton button = new JButton("Button");
button.setBounds(100, 60, 80, 30);
button.setToolTipText("A button component");

panel.add(button);
```

Notice that the Swing toolkit determines which tooltip to display, depending on where the mouse pointer is located on the application. The toolkit selects the tool tip for the widget in the highest visible layer on the GUI container.

### 17.4.2 Menu Bars and Tool Bars

Menu bars are used to collate lists (menus) of functions (menu items) of a GUI application. A sub set of the most frequently used functions are also often replicated in tool bars. Figure 17.6 illustrates an example Swing menu bar with a single menu containing a single menu item. The Java code below shows how to implement a menu bar using Swing.

```
JMenuBar menubar = new JMenuBar();

JMenu file = new JMenu("File");
file.setMnemonic(KeyEvent.VK_F);

JMenuItem fileClose = new JMenuItem("Close");
fileClose.setMnemonic(KeyEvent.VK_C);
fileClose.setToolTipText("Exit application");

fileClose.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
        System.exit(0);
    }
});

file.add(fileClose);
menubar.add(file);
setJMenuBar(menubar);
```

First a new menu bar instance is constructed to store all the menus. Then a Menu instance, labelled "File" is constructed. A keyboard short cut (ALT-f) is added to the menu, so that it can be selected from the menu bar using the mouse or the keyboard. The ALT-f short cut follows a common graphical user interface convention for accessing a file menu.

By convention, a file menu contains menu items for manipulating files, including open, close and save. So next, a new MenuItem instance is constructed

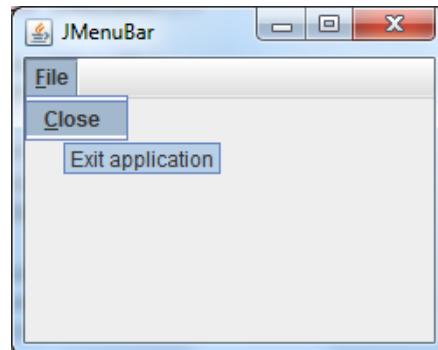


Figure 17.6: a simple menu bar

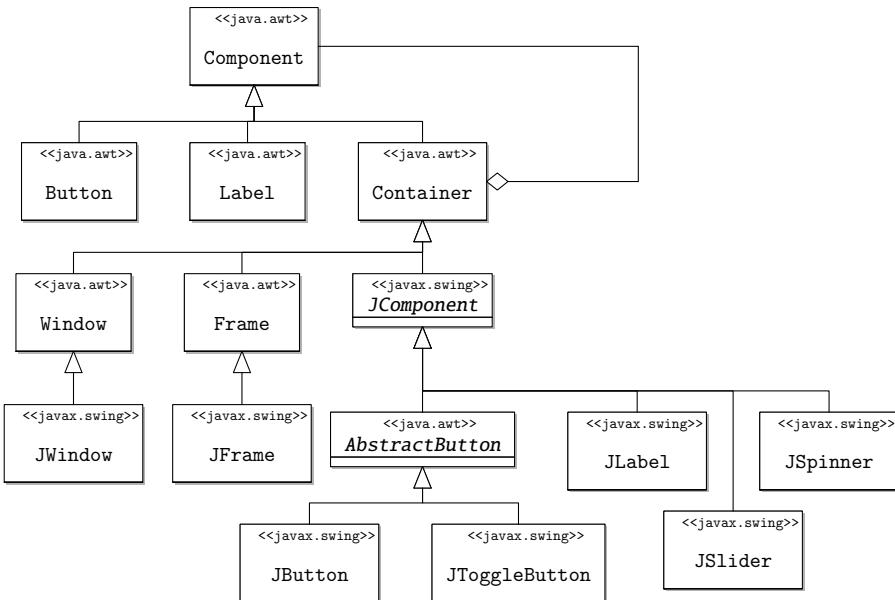


Figure 17.7: the AWT and Swing class hierarchy

with the label "Close". A keyboard short cut is added for this item (ALT-C) and a tool tip ("Exit Application").

To add some functionality to the menu, an `ActionListener` is added to the close menu item that causes the application to exit normally. Notice that the `ActionListener` is another anonymous class.

Finally, the close menu item is added to the file menu, the file menu is added to the menu bar and the menu bar is added to the surrounding `JFrame`.

### 17.4.3 Controlling Container Layout

Figure 17.7 illustrates the class hierarchy of the Java Foundation Class framework, which combines both Swing and Abstract Windows Toolkit (AWT). The top of the hierarchy is the AWT class, `Component`, which defines the basic functionality of any GUI widget. The Abstract Window Toolkit (AWT) distinguishes between widgets and containers. Widgets implement the functionality of a window, such as `Button`, `Label` and `TextField`, whereas containers structure the layout of any contained components. Containers can be nested within one another to build complex layouts. This is an example of the composite (general hierarchy) pattern (see Section 18.2.6).

In Swing (and unlike AWT), all components on a graphical user interface are sub classes of AWT Containers, extending layout control to all visible components. Some containers provide the actual functionality of the user interface, such as buttons (`JButton`), labels (`JLabel`), text fields (`JTextField`), and canvases (`JCanvas`). Other containers group children into suitable layouts.

Controlling Layout  
AWT vs. Swing (457)

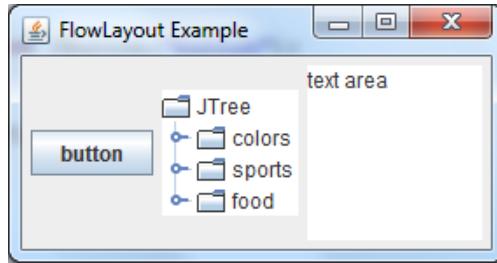


Figure 17.8: example of using the FlowLayout layout manager

Layouts can either be controlled using absolute settings, as in the Buttons example (which is rather tedious), or using a LayoutManager, which lays out the GUI according to a prescribed behaviour.

FlowLayout is the default layout manager. The manager puts components into a row in the order in which they were added. Figure 17.8 illustrates how the manager lays out widgets inside a container. The code which implements the example is shown below.

FlowLayout (458)

```

 JPanel panel = new JPanel();

 JTextArea area = new JTextArea("text area");
 area.setPreferredSize(new Dimension(100, 100));

 JButton button = new JButton("button");
 panel.add(button);

 JTree tree = new JTree();
 panel.add(tree);

 panel.add(area);

 add(panel);

```

To put a component inside of a container, we use the `Container.add()` method. If they do not fit into one row, then they continue on in the next row. Components can be added from left to right and vice versa. The manager supports alignment of components in multiple rows. Implicitly, components are centered and there is a 5 pixel space between components and the edges of the container.

The flow layout manager sets a preferred size for its components; if we did not set the preferred size for the text area component, its size would be the size of its text.

The GridLayout layout manager aligns all the components in a container within a grid. Consequently all the components in the grid are the same size. Figure 17.9 illustrates the effect of the GridLayout layout manager for con-

structuring the keypad for a calculator application. The Java code below shows how to use the `GridLayout` class, as well as how to define different border styles for a Swing component.

```
JPanel panel = new JPanel();
add(panel);

Border border =
BorderFactory.createEmptyBorder(5, 5, 5, 5);
panel.setBorder(border);
```

A new `JPanel` is created and added to the surrounding `JFrame`. An empty border is created using swing's `BorderFactory` class and set as the border for the `JPanel`. This creates a 5pt invisible boundary around the panel. Other borders can also be created using the `BorderFactory`, such as a bevelled or raised borders.

Notice that the `BorderFactory` is the simpler variant of the factory pattern described in Section 18.3.2. The `BorderFactory` contains methods for creating the different border types.

```
int rows = 5, columns = 4;
int hgap = 5, vgap = 5;

GridLayout layout =
new GridLayout(rows, columns, hgap, vgap);

panel.setLayout(layout);
```

A new `GridLayout` manager is created for a container with 5 rows and 4 columns. The constructor is also passed arguments for vertical and horizontal spacing between cells (5 pixels).

```
String[] buttonLabels = {
```

GridLayout and  
borders (460)

Layout (461)

Add the components  
(462)



Figure 17.9: using the grid layout manager

```

    "Cls", "Bck", "M", "Close",
    "7", "8", "9", "/",
    "4", "5", "6", "*",
    "1", "2", "3", "-",
    "0", ".", "=",
    "+"
};

for (String buttonLabel: buttonLabels)
    panel.add(new JButton(buttonLabel));

```

A string array of twenty elements is created to hold all the labels for the GUI. The labels are added to the container in sequence (using a for-each loop) as JButton components. The layout manager fills in each row from left to right and top to bottom as new components are added.

A third layout policy can be used with the BorderLayout, illustrated in Figure 17.10. The BorderLayout manager divides the container into 5 regions, North, West, South, East and Center. The GUI window containing a button along the northern border, a label along the southern border and a toolbar along the western border. Figure 17.10(b) illustrates how a tool bar can be undocked from its parent window. The toolbar can also be re-docked by clicking on its close button.

BorderLayout and tool bars (463)

Each region can have only one component. If it is necessary to put more than one component into a region, a nested JPanel can be used, with the panel having any layout manager. The components in the north, west south and east regions receive their preferred sizes; the component in the center takes up all of the remaining space.

```

setTitle("BorderLayout");

JToolBar horizontal = new JToolBar();
horizontal.setFloatable(false);

JButton bexit = new JButton("exit");
bexit.setBorder(new EmptyBorder(0, 0, 0, 0));
horizontal.add(bexit);

```

We want to add a toolbar to the top of the GUI (just below where the menu bar would be placed). A JToolBar is created with a single JButton added ("exit"). The capability to float is disabled for the toolbar. The tool bar is then added to the north of the containing JFrame.

```

JToolBar vertical = new JToolBar(JToolBar.VERTICAL);
vertical.setFloatable(true);
vertical.setMargin(new Insets(10, 5, 5, 5));

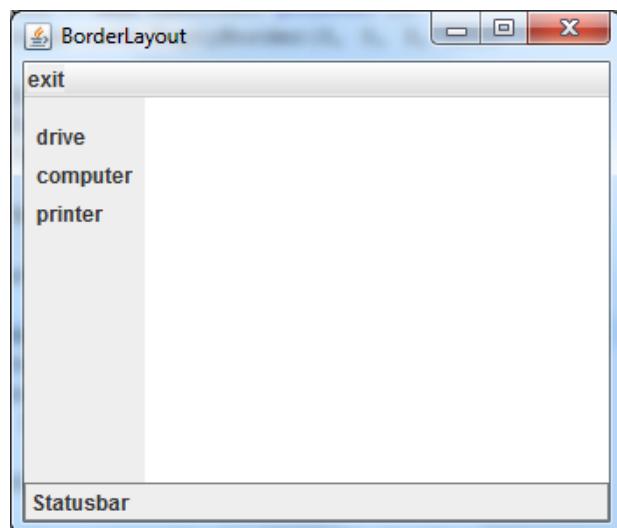
JButton jBdrive = new JButton("drive");
jBdrive.setBorder(new EmptyBorder(3, 0, 3, 0));

JButton jBcomputer = new JButton("computer");

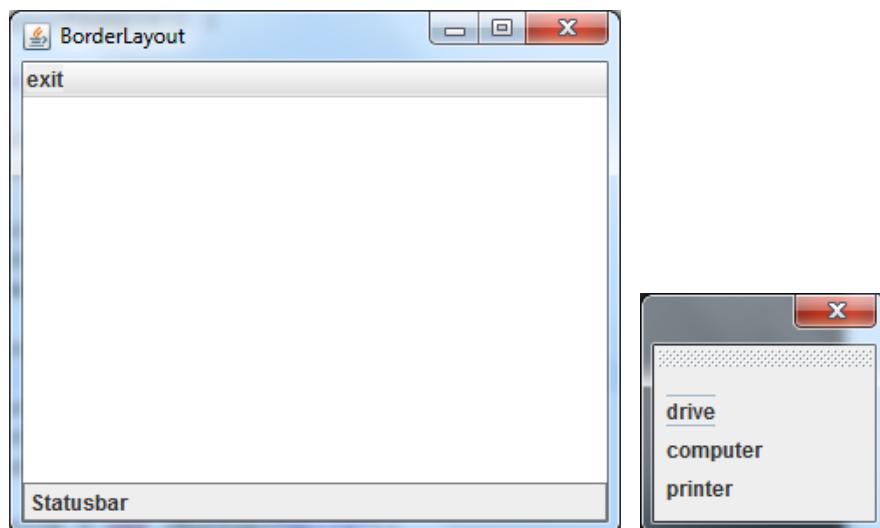
```

Horizontal Toolbar (464)

Vertical Toolbar (465)



(a) docked tool bar



(b) un docked

Figure 17.10: the BorderLayout layout manager with a tool bar

```

jBcomputer.setBorder(new EmptyBorder(3, 0, 3, 0));
JButton jBprinter = new JButton("printer");
jBprinter.setBorder(new EmptyBorder(3, 0, 3, 0));

vertical.add(jBdrive);
vertical.add(jBcomputer);
vertical.add(jBprinter);

add(vertical, BorderLayout.WEST);

```

The toolbar is added to the left of the GUI (just below where the menu bar would be placed). A JToolBar is created with three JButton instances added: ("drive", "computer" and "printer").

The toolbar is set to be floatable, which means it can be removed from the GUI during interaction and it will be automatically placed in its own JWindow (this is actually the default for JToolBar).

Note that the vertical layout is specified for the toolbar in the constructor. The tool bar is then added to the west of the containing JFrame.

## - Text Area and Status Bar (466)

```

add(new JTextArea(), BorderLayout.CENTER);

JLabel statusbar = new JLabel(" Statusbar");
statusbar.setPreferredSize(new Dimension(-1, 22));
statusbar.setBorder(LineBorder.createGrayLineBorder());
add(statusbar, BorderLayout.SOUTH);

```

A new JTextArea is added to the center containing JFrame, so this will take up all the available space, including the empty area to the right of the frame. A JLabel is added to the bottom of the JFrame with the label " Statusbar". The BorderFactory class is used to give the status bar a defined area.

### 17.4.4 Events

## Events in Swing (467)

All GUIs are driven by *events*, usually from the user. Events encapsulate information about the state which has changed as a result of a user action.

Event management follows the Observer-Observable design pattern (see Section 18.4.3). An event source (the *observable*) is a swing component whose state changes, often because of an action by the user. In the class diagram example in Figure 17.11, this is the JButton class, which inherits its observable behaviour from AbstractButton.

An event listener is an *observer* that can be registered to receive notification that some observable object has changed. In the example, the interface to be implemented is ActionListener. Any object whose class implements ActionListener can be registered to receive ActionEvents via the AbstractButton.addActionListener() method. Each AbstractButton instance maintains a collection of ActionListener instances that have been registered with it (see the aggregation association on the diagram). When the button is trig-

gered (via an invocation of `fireActionPerformed()`, all registered `ActionListener` instances are notified of the event via the `actionPerformed()` method.

There are numerous types of events in Swing, for example:

Types of events (468)

- `ActionEvent` indicates actions such as button presses;
- `KeyEvents` indicate that a keyboard key has been pressed;
- `MouseEvents` indicates that the mouse has been moved, or that a mouse button has been clicked;
- `TextEvents` indicate that some text in a component such as a `JTextArea` or `JLabel` has changed; and
- `FocusEvents` indicates that the active component on the GUI has changed.

There are several ways of arranging listeners and actions for swing widgets, some of which we have already seen:

- external `ActionListener` classes;
- inner `ActionListener` classes;
- anonymous inner `ActionListener` classes (as in the Buttons application); and
- derived `ActionListener` classes.

Structuring user interfaces with event listeners (469)

The following section will discuss the approaches to implementing GUI listeners that we haven't seen yet.

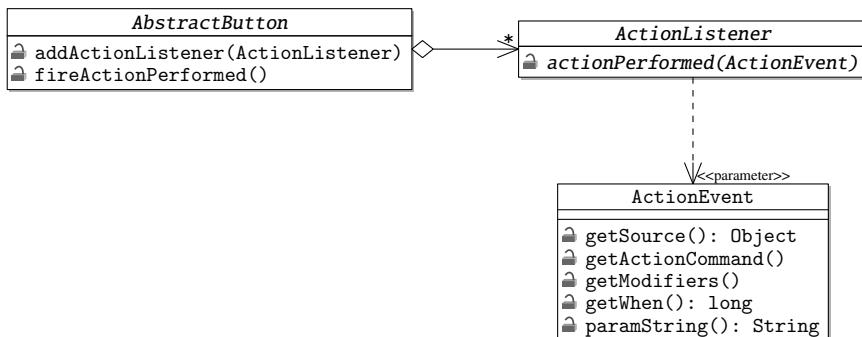


Figure 17.11: example of the swing event model

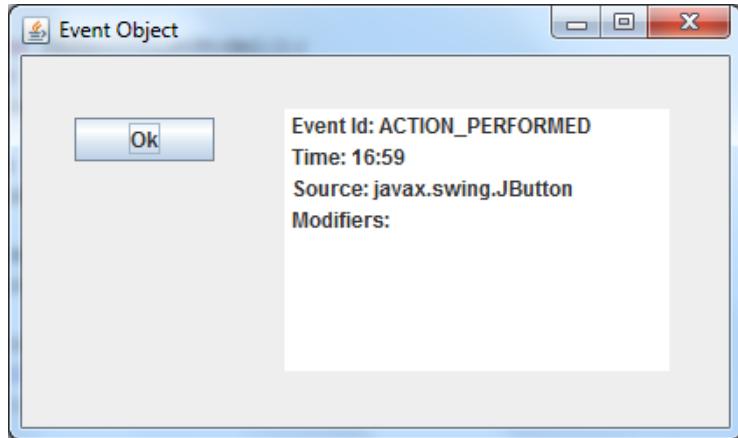


Figure 17.12: the event object information GUI

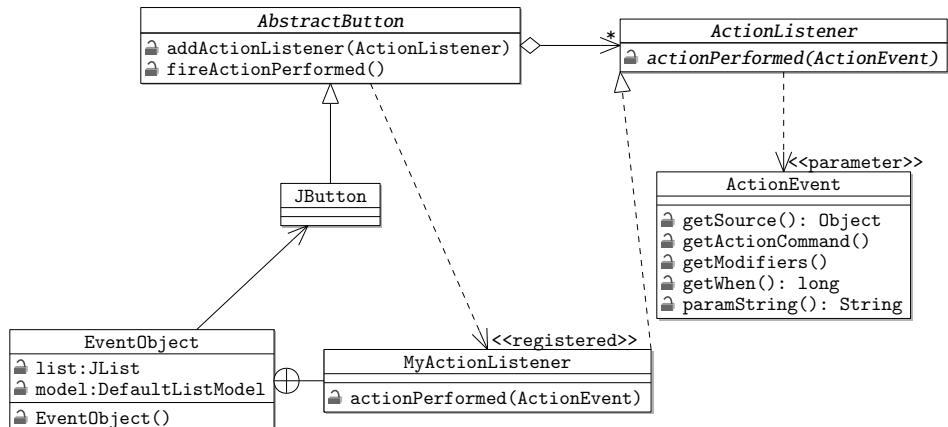


Figure 17.13: action listeners as inner classes

### Action Listeners as Inner Classes

Figure 17.12 illustrates the EventObject GUI. The GUI contains a list and a button. The text area is used to report the information that can be gathered when an interaction with the GUI takes place. Specifically, when the button is pressed, the id of the resulting event, the time it occurred, the source Swing component it came from and any modifiers that were applied (holding down a keyboard key for example) are reported in the text area.

Events and lists (470)

Figure 17.13 shows how EventObject completes the (observer-observable) button action listener pattern in Figure 17.11 using an inner-class. The application consists of a main class (EventObject) with a constructor, EventObject (), for setting up the GUI, and an inner, nested, class MyActionListener, which implements the ActionListener interface. An inner class is denoted on a UML class diagram using the crossed-circle arrow head.

Action listeners as inner classes (471)

The MyActionListener class implements the actionPerformed(ActionEvent) method and is registered to receive ActionEvent events from the JButton, via the registration methods in the AbstractButton super class. Note that JButton is just one type of AbstractButton in the Swing toolkit.

The MyActionListener instance has access to the instance attributes of its parent instance, because it is constructed from an inner class of EventObject. Consequently, the MyActionListener instance can access and modify the features of the model and list attributes.

The Java code below shows the outline of the EventObject class in Figure 17.13 (method bodies are omitted).

```
public class EventObject extends JFrame {  
    private JList list;  
    private DefaultListModel model;  
    public EventObject() {  
    }  
    class MyActionListener implements ActionListener {  
        public void actionPerformed(ActionEvent event) {  
        }  
        private String getFormattedModifiers(int modifiers){  
        }  
        private String getFormattedDate(long when) {  
    }
```

The code below shows the body of the EventObject() constructor. Much of the GUI is built as for previous examples. The JList instance is passed a ListModel for representing the data associated with the list. The model will be used by the MyActionListener instance to add information about events to the JList component on the GUI. Note this illustrates good design practice by separating data (in the DefaultListModel) from the view (in the JList widget). A MyActionListener instance is created and added to the JButton.

EventObject.  
EventObject()  
(473)

```
setTitle("Event Object");  
  
JPanel panel = new JPanel();  
panel.setLayout(null);  
  
model = new DefaultListModel();  
list = new JList(model);  
list.setBounds(150, 30, 220, 150);  
  
JButton ok = new JButton("Ok");  
ok.setBounds(30, 35, 80, 25);  
  
ActionListener al = new MyActionListener();  
ok.addActionListener(al);  
  
panel.add(ok);  
panel.add(list);
```

actionPerformed()  
(474)

```
    add(panel);
```

The code below illustrates the body of the actionPerformed() method of the MyActionListener inner class. When actionPerformed() is invoked (as a result of a button press), the method adds information about the event to the list model, for display by the list in the GUI.

```
    if (!model.isEmpty()) model.clear();

    if (event.getID() == ActionEvent.ACTION_PERFORMED)
        model.addElement(" Event Id: ACTION_PERFORMED");

    String date = getFormattedDate(event.getWhen());

    model.addElement(" Time: " + date);

    String source = event.getSource().getClass().getName()
    ;
    model.addElement(" Source: " + source);

    String modifiers =
        getFormattedModifiers(event.getModifiers());
    model.addElement(modifiers);
```

First any data in the list model is cleared. Then the following information is added to the list model:

- the type of event;
- the date when the press occurred to accessed from the ActionEvent.getWhen() method;
- the source of the ActionEvent is the JButton instance from where the button press originated. The code obtains the name of the source instance's class ("JButton"); and
- the modifiers, which are any keys that were held down when the button was pressed.

The Java code below shows how the modifiers are formatted for presentation in the list on the GUI. The code is contained in the getFormattedModifiers() method, which is called by the actionPerformed() method.

()  
(475)

```
private String getFormattedModifiers(int modifiers){
    StringBuffer buffer = new StringBuffer(" Modifiers: ")

    ;

    if ((modifiers & ActionEvent.ALT_MASK) > 0)
        buffer.append("Alt ");
```

```

if ((modifiers & ActionEvent.SHIFT_MASK) > 0)
    buffer.append("Shift ");

if ((modifiers & ActionEvent.CTRL_MASK) > 0)
    buffer.append("Ctrl ");

return buffer.toString();
}

```

The modifier is a 32 bit integer value, whose individual bits are used to flag whether a particular modifier was used or not. Each bit flag mask is specified by a static constant field in the ActionEvent class, ALT\_MASK:int for the Alt button, SHIFT\_MASK:int for the Shift key and so on. Consequently, each button modifier can be parsed out from the modifier value using the bitwise AND operator, &.

A StringBuffer is used to collate a formatted report on the modifiers that were held down at the time the button was pressed. This buffer is then returned to the calling actionPerformed() method as a String.

The Java code shown below illustrates how the time that the button was pressed is formatted for presentation. The code is from the body of the getFormattedDate() method.

```

Locale locale = Locale.getDefault();
Date date = new Date(when);
return
DateFormat.getTimeInstance(
    DateFormat.SHORT, locale).format(date);

```

getFormattedDate()  
(476)

The date is provided as a long value, obtained by calling the getWhen() method on the ActionEvent instance. The value represents the number of milli-seconds since the Unix Time epoch. More information on date handling can be found in the Java SDK reference for the java.util.Date class. The when parameter is wrapped in a Date() instance.

A default Locale instance is obtained from the Locale class to be used to format the date for the current time zone and location conventions. The DateFormat class is then used to format the date variable and return the formatted String instance to the calling actionPerformed() method. Notice the SHORT flag used to specify the type of time formatter required.

## Derived Widgets

Figure 17.14 illustrates the UsingInterface GUI, which was implemented using derived widgets with self contained listeners. The class has a single button, close placed using the setBounds() method on the GUI.

UsingInterface (477)

Figure 17.15 shows how UsingInterface implements the event-listener (observer-observable) pattern using derived listeners. In this approach, there is still an inner-class MyButton, which has access to the fields of the UsingInterface class.

UsingInterface (478)

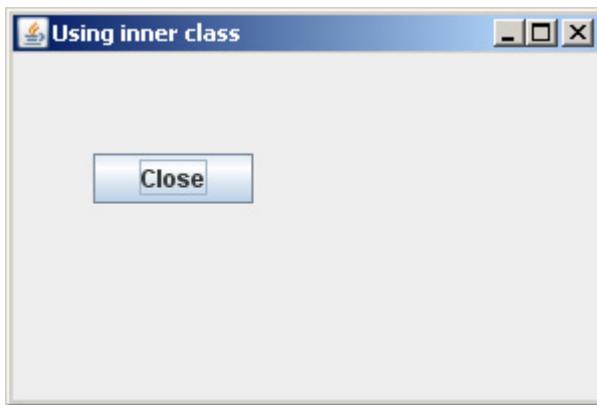


Figure 17.14: using interface

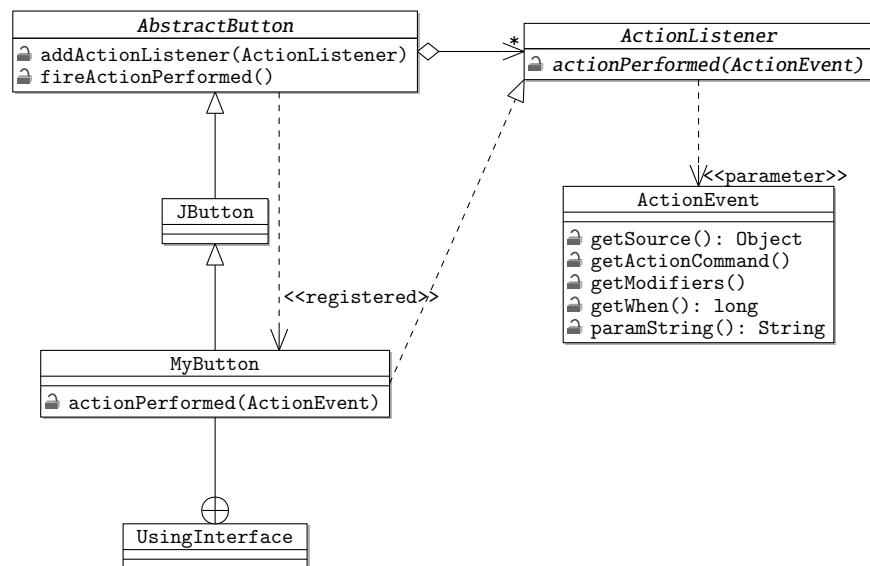


Figure 17.15: derived widget classes with self contained listeners

However, the inner MyButton class extends the JButton class, *and* implements the ActionListener interface. In addition, during construction, each MyButton instance registers itself, with itself (via the AbstractButton class). This means that instances of the MyButton class will receive events about itself.

The Java code below shows how the derived widget approach is implemented for the MyButton class.

MyButton (479)

```
class MyButton extends JButton implements ActionListener
{
    private static final long serialVersionUID =
        1877814626850005219L;

    public MyButton(String text) {
        super.setText(text);
        addActionListener(this);
    }

    public void actionPerformed(ActionEvent e) {
        System.exit(0);
    }
}
```

The MyButton(String) constructor invokes the super constructor of JButton and passes it the label text to be displayed on the button. The constructor also registers the instance as an ActionListener of itself. When the button is pressed the actionPerformed() method is invoked, and the system exits.

## Events from Multiple Sources

The application illustrated in Figure 17.16 shows the de-coupling between GUI widgets and the listeners which handle the events they generate. The application has four buttons, but only one listener. In this approach, the ButtonListener is implemented as an inner class of the MultipleSources GUI, similar to the arrangement shown Figure 17.13.

The Java code shown below shows the outline of theMultipleSources class (method bodies are omitted). The code includes a constructor and an inner ButtonListener class which implements ActionListener.

Events from Multiple Sources (480)

```
public class MultipleSources extends JFrame {
    JLabel statusbar;
    public MultipleSources() {
    }
    class ButtonListener implements ActionListener {
        public void actionPerformed(ActionEvent e){
        }
    }
    public static void main(String[] args) {
    }
}
```

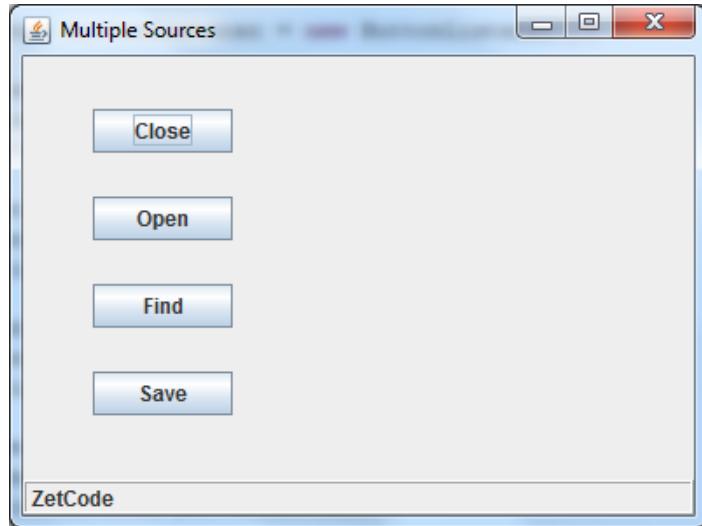


Figure 17.16: capturing events from multiple sources

The Java code below shows the body of the `MultipleSources()` constructor. Much of the code is now familiar from previous examples. This part of the `MultipleSources()` constructors sets up the GUI with a `JLabel` status bar and a `JPanel` to hold the buttons.

`MultipleSources()`

(482)

```
setTitle("Multiple Sources");

JPanel panel = new JPanel();
panel.setLayout(null);
add(panel);

statusbar = new JLabel(" ZetCode ");
add(statusbar, BorderLayout.SOUTH);
Border border = BorderFactory.
createEtchedBorder(EtchedBorder.RAISED);
statusbar.setBorder(border);
```

The next part of the constructor sets up each of the four buttons with a shared `ButtonListener` instance to report button clicks:

`MultipleSources()`

(483)

```
ButtonListener buttonListener = new ButtonListener();

JButton close = new JButton("Close");
close.setBounds(40, 30, 80, 25);
close.addActionListener(buttonListener);

JButton open = new JButton("Open");
open.setBounds(40, 80, 80, 25);
open.addActionListener(buttonListener);
```

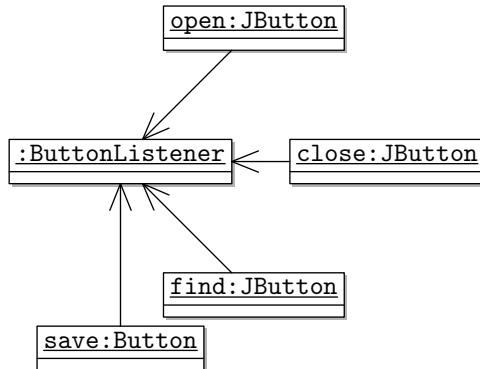


Figure 17.17: the state of the `MultipleSources` application after the constructor has completed

```

 JButton find = new JButton("Find");
 find.setBounds(40, 130, 80, 25);
 find.addActionListener(buttonListener);
 panel.add(close);
 panel.add(open);
 panel.add(find);
 panel.add(save);

```

Figure 17.17 shows the state of the `MultipleSources` program when the constructor has finished. The four `JButtons` share the `ButtonListener` instance, so that the `ButtonListener` is responsible for handling events from all of the buttons.

Finally, the code shown below illustrates the implementation of the `ButtonListener` inner class itself.

```

class ButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent e){
        JButton o = (JButton) e.getSource();
        String label = o.getText();
        statusbar.setText(" " + label + " button clicked");
    }
}

```

Events from multiple sources (484)  
`ButtonListener` (485)

The `actionPerformed()` method uses the `getSource()` method of the event to determine which button was pressed. Notice that the source of the event must be cast to a `JButton`, since there could be many different types of `ActionEvent` sources. Once the button has been obtained, the button label is accessed and used to set the text of the `statusbar` `JLabel` component.

### Choosing an `ActionListener` pattern

Choosing which widget-behaviour pattern depends on three factors:

- readability;
- encapsulation (cohesion); and
- extensibility (coupling).

Using derived widgets with self contained listeners is often convenient, but couples observable behaviour with the action to take. Each new behaviour will require a new derived MyButton like class, increasing the potential for cloning.

Using an external ActionListener class is an appropriate way of decoupling actions from behaviour, but reduces cohesion, since a listener class implementation is often closely associated with whichever class will be affected by the action. An external listener class will not have access to internal attributes of the affected class, meaning they will need to be exposed for access. This potentially increases coupling with other parts of the system, by increasing the scope of the target application's public API.

Using inner class action listeners improves cohesion, since the action listener is kept with the class that it will affect. However, this approach can make the source code harder to read, since one class implementation is nested inside another.

#### 17.4.5 Dialogs

Dialogs (486) A dialog is a window which is used to “talk” to the application. Types of dialog include:

- custom dialogs are extended from JDialog;
- standard (convenience dialogs), e.g. for choosing files are available in the Swing toolkit; and
- *Modal dialogs* block input to other application windows while they are active.

Figure 17.18 illustrates the implementation of an ‘about’ dialog for an application. About dialogs display basic information about an application, including the application name, version number, author and/or software company, as well as any copyright or legal notices. About dialogs are by convention accessed via the ‘Help’ menu’s ‘About’ item in a GUI. The dialog also usually includes an ‘Ok’ or ‘Close’ button for when the user has finished reading the application information.

AboutDialog() (487) The Java code below shows the implementation of the AboutDialog class, which extends the JDialog class. All the implementation is contained in the AboutDialog() constructor.

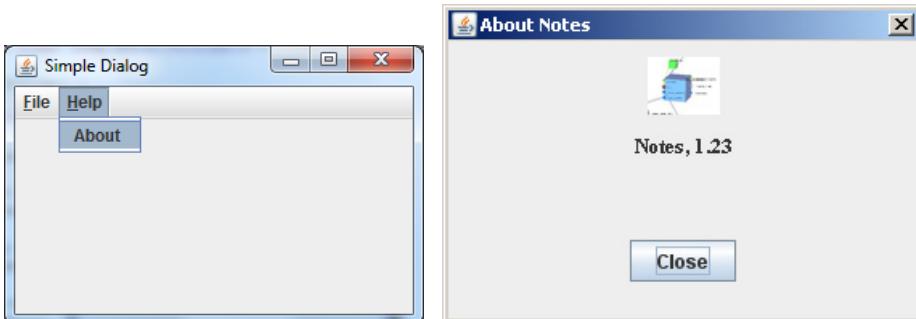


Figure 17.18: a simple dialog

```

setLayout(
    new BoxLayout(getContentPane(), BoxLayout.Y_AXIS));

add(Box.createRigidArea(new Dimension(0, 10)));

ImageIcon icon = new ImageIcon("images/swing/note.jpg")
    ;
JLabel label = new JLabel(icon);
label.setAlignmentX(0.5f);
add(label);

add(Box.createRigidArea(new Dimension(0, 10)));

JLabel name = new JLabel("Notes, 1.23");
name.setFont(new Font("Serif", Font.BOLD, 13));
name.setAlignmentX(0.5f);
add(name);

```

The constructor lays out the dialog area using the `BoxLayout` layout manager, which has similar features to `FlowLayout`. Separators are added to the dialog using the `Box` class.

Images can be added to GUI components using the `ImageIcon` class. The class has a constructor which takes a file path `String` as an argument. The resulting icon instance can then be used as the label argument to a `JLabel` constructor call.

The code also shows the use of alternative fonts for Swing. A new `Font` instance is constructed using the font face "`Serif`", the `BOLD` style (a static `Font` class constant), and size 13 points. The `setFont()` method is then used to set the `JLabel` instance's font.

The next block of code adds a `JButton` in the same way as Buttons. However, instead of setting the `ActionListener` to close the application when an `ActionEvent` is received, the `dispose()` method of the super class is called, which just closes the dialog.

`AboutDialog() (488)`

```
add(Box.createRigidArea(new Dimension(0, 50)));
```

```

        JButton close = new JButton("Close");
        close.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent event) {
                dispose();
            }
        });

        close.setAlignmentX(0.5f);
        add(close);

        setModalityType(ModalityType.APPLICATION_MODAL);
    }
}

```

The final statement in this block sets the modality type for the dialog. The dialog is made APPLICATION\_MODAL, meaning that it blocks input to other parts of this application while the dialog is active, but other applications on the desktop are unaffected.

The dialog is linked to the GUI application by adding a menu item to the menu bar. The Java code below shows the body of the constructor for the SimpleDialog class. The code is substantially the same as the code for adding the file menu shown in Section 17.4.2.

SimpleDialog()  
(489)

```

    setTitle("Simple Dialog");

    JMenuBar menubar = new JMenuBar();

    JMenu file = new JMenu("File");
    file.setMnemonic(KeyEvent.VK_F);

    JMenu help = new JMenu("Help");
    help.setMnemonic(KeyEvent.VK_H);

    menubar.add(file);
    menubar.add(help);
    setJMenuBar(menubar);
}

```

SimpleDialog()  
(490)

This code shows how to cause the AboutDialog to be made visible after clicking on the "About" JMenuItem.

```

    JMenuItem about = new JMenuItem("About");

    about.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent event) {
            AboutDialog ad = new AboutDialog();
            ad.setVisible(true);
        }
    });

    help.add(about);
}

```

## 17.5 The Apache Axis Web Service Framework

This section introduces the concept of middleware frameworks for developing distributed applications, and explains how the service oriented architecture model of middleware is realized in the Apache Axis framework.

There are several uses for middleware frameworks:

- provide a deployment, coordination and composition platform for diverse, reusable components. Engineers can focus on application details, while leaving communication to be handled by the middleware;
- distribution of computation across multiple processors and domains of control, so that components can be exposed for re-use by other organisations as well as utilise computational services provided by others to compose new applications;
- allow for multiple “behind the scenes” implementations of services advertised and invoked via a common middleware; and
- ‘advertise’ and ‘sell’ a computational service to others.

Middleware frameworks (491)

Figure 17.19 illustrates the Service Oriented Architectural pattern. A client wishes to utilise some functionality provided by a remote service. To do this, the client component engages in *discovery*, in which the *registry* component is queried regarding the availability of services matching the needs of the client. Services that are able to offer the functionality will have previously registered their capabilities with the registry.

The client is provided with a list of possible services, and *binds* to that service to interact with it. This arrangement is known as *late-binding*, because the client only links to the services it depends on immediately prior to using them.

Before we proceed further, there is some terminology to be aware of

Service Oriented Architecture Pattern (492)

Some terminology (493)

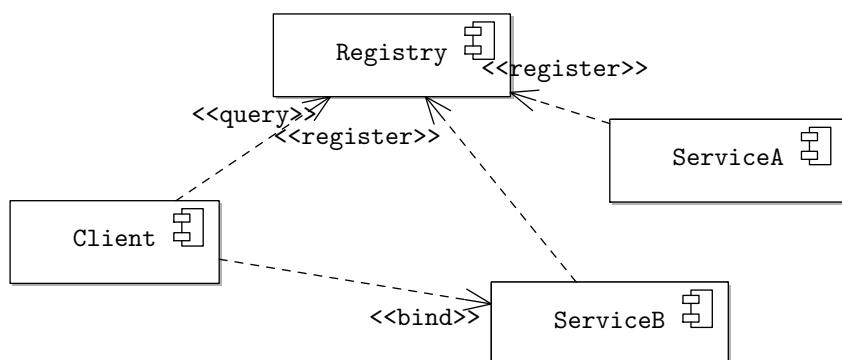


Figure 17.19: the service oriented architecture pattern

- Jakarta Tomcat, is an *application container* used for deploying program components as *web services*
- SOAP, known as the Simple Object Access Protocol, although, this term is now deprecated, probably because SOAP is neither simple, nor for really about accessing objects!. SOAP is:
  - an XML based messaging protocol;
  - used (by us) for making remote procedure calls (RPC) between web clients and services.
- Apache Axis - a web service *framework*, including:
  - a SOAP implementation;
  - APIs for invoking remote web services; and
  - tools for deploying WS applications.

Limitations of Axis/SOAP (494)

There are some limitations of the Axis/SOAP framework:

- existing programs cannot be automatically divided into equivalent web service components
- web service execution is stateless. It is necessary to manually store state information between invocations. The Web services resource framework extends web services to give explicit support to stateful components
- calls to remote web services must be manually coded into client programs;
- axis cannot automatically serialise/deserialise complex values; and
- serialisation and de-serialisation of objects via SOAP is expensive.

Deploying a Web Service (495)

### 17.5.1 Deploying Web Services

Web services are deployed in the Tomcat server's axis web application. Typical locations are:

```
%# on a muck
%>export TOMCAT_HOME=/opt/local/share/java/tomcat5
%# on ubuntu and other *nix
%>export TOMCAT_HOME=/var/lib/tomcat5
%# axis is a tomcat webapp
%>export AXIS_HOME=${TOMCAT_HOME}/webapps/axis
```

These are the default locations set in the ant build.xml script included with the software accompanying these notes. The script above is using bash shell variables to define a location for axis. There are two methods for deploying web services in axis:

- Java Web Services (JWS)
- Web Service Definition Descriptor (WSDD)

## Java Web Services

Deploying using JWS is relatively, straight forward. In summary, the steps are: Deploying a web service using JWS (496)

1. write the service as a Java program (recommend eclipse for this)
2. Make any methods that are to be exposed as web services public
3. Change the file extension from .java to .jws
4. Copy the source file to the axis home directory:

```
% ${TOMCAT_HOME}/webapps/axis
```

5. Test the service using a web browser: `http://<hostname>:8080/axis/<service>.jws?method=<method>&parameter=<parameter>`

The code below shows an example web service.

```
package uk.ac.glasgow.oose2.soa;

public class HelloWorld {
    public String sayHelloTo(String name){
        return "Hello "+name+", how are you doing?";
    }
}
```

Use the command prompt to copy this file to the axis directory:

```
%>sudo cp \
% src/uk/ac/glasgow/oose2/soa/HelloWorld.java \
% ${AXIS_HOME}/HelloWorld.jws
```

Use a text editor to remove the package statement from the .jws file, then test the service in a browser:

```
http://<hostname>:8080/axis/HelloWorld.jws?method=sayHelloTo&parameter=
Tim
```

You should see the following reply message in your browser (the exact format may vary).

```
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <soapenv:Body>
```

Example Service - HelloWorld (497)

SOAP Reply Message from HelloWorld (498)

```

<sayHelloToResponse
    soapenv:encodingStyle=
        "http://schemas.xmlsoap.org/soap/encoding/">

    <sayHelloToReturn xsi:type="xsd:string">
        Hello Tim, how are you doing?
    </sayHelloToReturn>

</sayHelloToResponse>

</soapenv:Body>
</soapenv:Envelope>

```

And that's it! Notice that the String return value is encoded in the reply message.

## Setup Eclipse with Axis (499)

The next step is to create a Java web service client to interact programmatically with the web service. To do this, you need to add axis jar libraries to your project's class path:

Project → Properties → Libraries → Add External Jars

as shown in Figure 17.20.

The next step is to create the Java client itself. The Java code below shows how to implement a client that interacts with the web service we have already

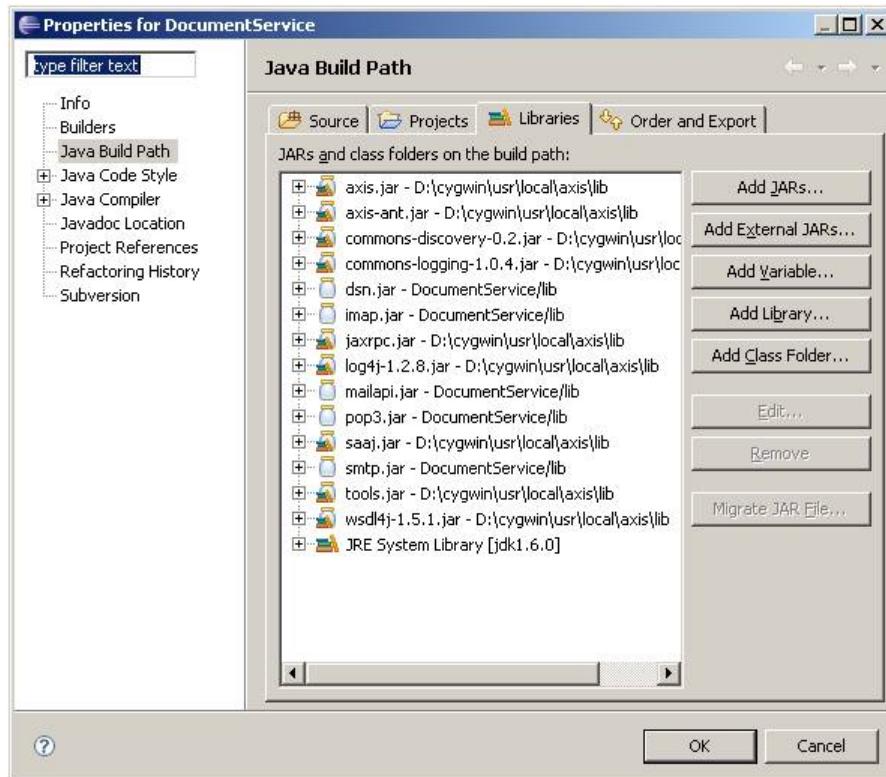


Figure 17.20: setting up Eclipse with Axis

created.

A Java Client (500)

```
public class HelloClient {
    public static void main (String[] args)
        throws ServiceException,
        MalformedURLException,
        RemoteException{

        Service service = new Service();
        Call call = (Call) service.createCall();

        String serviceURL =
            "http://macdonald.dcs.gla.ac.uk:8080/axis/HelloWorld.
            jws";

        call.setTargetEndpointAddress(new URL(serviceURL));
        call.setOperationName(
            new QName("http://DefaultNamespace", "sayHelloTo"));

        String returnValue = (String)
            call.invoke(new Object[] {"Hubert"});

        System.out.println(returnValue);
    }
}
```

The `main()` method signature is altered to indicate that it may throw an exception of a number of different types. Really, we should enclose the offending statements in a **try-catch** block, however this is a convenience for the time being to avoid cluttering the code.

The first two statements create the specification for the remote service and a call to that service, respectively. Then, the end point URL of the call is set to be that of the JWS service that has already been deployed. The next statement sets the name of the web service operation to be invoked by the call. Finally, the call is invoked, and the result of the call is cast to a `String` and printed out. The output on the console is:

```
Hello Hubert, how are you doing ?
```

## Web Service Deployment Descriptor

The Web Service Deployment Descriptor (WSDD) mechanism provides an alternative means of deploying web services that addresses many of the limitations of JWS, in particular:

- no packages allowed;
- one java source file per service; and

Web Service  
Deployment  
Descriptor (501)

- a tight coupling between java source and the service specification.

In the WSDD approach, class files must be pre-compiled, and copied to the axis application directory:

```
%> cd workspace/<PROJECT>/bin
%> sudo cp -r ./ ${AXIS_HOME}/WEB-INF/classes
```

or archived in a .jar file:

```
%>jar -cvf MyService.jar ./
%>sudo cp MyService.jar ${AXIS_HOME}/WEB-INF/lib
```

It can sometimes be useful to restart Tomcat if you update the class files in your web service.

In addition to the implementation, the web service requires a Web Service Deployment Descriptor, which describes how the implementation should be treated by the application container. The code below shows an example WSDD file:

```
<deployment
  xmlns="http://xml.apache.org/axis/wsdd/"
  xmlns:java="http://xml.apache.org/axis/wsdd/providers/java"
>

<service name="HelloWorld" provider="java:RPC">
  <parameter
    name="className"
    value="uk.ac.glasgow.oose2.soa.HelloWorld"/>
  <parameter name="allowedMethods" value="sayHelloTo"/>
</service>

</deployment>
```

## The Web Service Deployment Descriptor (502)

## Deploy with WSDD (503)

The steps for using the WSDD are as follows:

1. Upload the config/HelloWorld.wsdd file to the server
2. Upload the script build.xml ant script
3. From a command line type:

```
%>./deploy.sh HelloWorld.wsdd
```

4. This uses an administration client program to pass the deployment description to the axis AdminService
5. Test the deployment (note the different URL):

```
http://<hostname>:8080/axis/services/<service>?method=
<method>&parameter=<parameter>
```

you can also set this all up as an eclipse launch configuration.

6. Test the client with the modified URL as well.

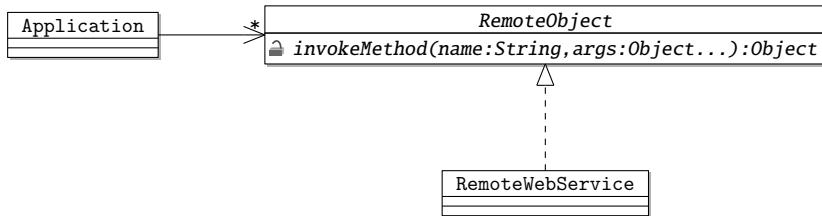


Figure 17.21: a generic remote service accessor

### 17.5.2 Distributed System/Web Service Engineering

Deploying and debugging web services (as distributed applications) can sometimes be quite fiddly. Some tips on managing the development are:

- web services can be clients of other services;
- it is possible to create multiple instances of the same service by changing the names of the end points and adding the new services to the WSDD file (or make separate WSDD files for each service);
- remember that components are stateless between WS invocations (although static attributes values *are* retained);
- use simple types for arguments and return values where possible;
- web service RPC calls do not handle large amounts of data very well. For example, byte arrays are encode as Base64, so the JVM will quickly run out of memory. Use attachments to WS messages to transfer large amounts of data; and
- debugging information `System.err` and `System.out` are written to

Tips for Designing  
Web Service  
Applications (504)

Good software design practice means that it can be useful to separate out web service invocation from processing code. Figure 17.21 illustrates a pattern for doing this. The `Application` class has a number of references to implementations of a `RemoteObject` interface. This interface has a single method that accepts an operation name and arguments for invocation, and returns a general object. The actual implementation of the `RemoteObject` could be either web service client code, or local code that invokes the appropriate Java method via reflection.

The Java code below shows how this is implemented. The `Application` class is a web service client, with a single method, `doSomething()`, for doing some interaction with a remote web service. The `Application()` constructor initializes

A Generic Remote  
Service Accessor (505)

Example -

Application (506)

the reference to the remote service using a RemoteWebService implementation. This is the only part of the code that 'knows' how the remote service end point is implemented. The doSomething() method invokes the service method "sayHelloTo", with argument "tim", but doesn't know how the invocation is handled.

```
public class Application {
    private RemoteService helloWorldService;

    public Application (){
        try {
            helloWorldService = new RemoteWebService(
                "http://macdonald.dcs.gla.ac.uk:8080/axis/services/
                HelloWorld");

        } catch (MalformedURLException e) {}
    }

    public void doSomething(){
        try {
            String result = (String)
                helloWorldService.invokeMethod("sayHelloTo", "tim");
            System.out.println(result);
        } catch (Throwable t) {}
    }

    public static void main(String[] args){
        new Application().doSomething();
    }
}
```

RemoteWebService  
(507)

The code below shows how the RemoteService interface is realized by a concrete web service class. The code is the same as for the remote web service client above.

```
public class RemoteWebService implements RemoteService {

    private URL serviceURL;

    public RemoteWebService(URL serviceURL){
        this.serviceURL = serviceURL;
    }

    public RemoteWebService(String serviceURL)
        throws MalformedURLException{
        this(new URL(serviceURL));
    }

    @Override
    public Object invokeMethod(String name, Object...
```

```

        arguments)
throws ServiceException, RemoteException{

    Service service = new Service();
    Call call = (Call) service.createCall();

    call.setTargetEndpointAddress(serviceURL);
    call.setOperationName(
        new QName("http://DefaultNamespace", name));

    return call.invoke(arguments);
}

```

Alternatively, the invocation could be handled by a local stub, which passes the service call off directly to the Java class that will eventually implement the web service. The code below shows how this is done.

```

public class ReflectionService implements RemoteService{
    private Class<?> clazz;

    public ReflectionService(Class<?> clazz){
        this.clazz = clazz;
    }

    @Override
    public Object invokeMethod(String name, Object... args)
        throws Throwable {
        Method invoc = null;

        Object target = clazz.newInstance();

        for (Method method: clazz.getMethods())
            if (method.getName().equals(name)){
                invoc = method;
                break;
            }
        if (invoc != null && target != null)
            return invoc.invoke(target, args);

        else throw new Exception();
    }
}

```

A local stub for  
RemoteWebService  
(508)

Rather than receiving a URL of a remote service in the constructor, a class object, `clazz` is provided. The `invokeMethod()` operation is implemented by:

- obtaining a new instance of the specified class;
- finding the method with the same name as the invoked service (note that

this implementation might be problematic if more than one method has the same name in a class); and

- invoking the method on the target instance with the specified parameters.

The result is exactly the same as invoking the operation on a web service backed by that Java class.

### 17.5.3 Serializing and De-serializing JavaBeans

The example applications we have seen so far have managed relatively simple data types, in particular string values. Other automatically supported data types include:

- `String`
- `int` / `Integer`
- `Byte`
- `byte[]`

Unfortunately, Axis cannot automatically serialise and de-serialise complex data types to XML for transmission between components in a distributed application. The *JavaBean* design pattern and the `BeanSerializerFactory` class can be used to transfer more complex objects between components. The Java code below shows how this is done, using an application taken from the axis samples library<sup>6</sup>

The class shown below is implemented following the *JavaBean* pattern. All the attributes in the class have corresponding accessor and mutator method pairs which follow the get/set convention. For example the `customerName` attribute can be accessed using the `getCustomerName()` method and altered using the `setCustomerName()` method. In addition, the class has a default nullary constructor and implements the `Serializable` interface.

```
package uk.ac.glasgow.oose2.soa;

public class Order {
    private String customerName;
    private String itemCodes[];

    public void setCustomerName(String customerName) {
        this.customerName = customerName;
    }

    public String getCustomerName() {
        return customerName;
    }
}
```

<sup>6</sup>TOMCAT\_HOME/webapps/axis/samples

```

}

public void setItemCodes(String itemCodes[]) {
    this.itemCodes = itemCodes;
}

public String[] getItemCodes() {
    return itemCodes;
}
}

```

The class shown below is the web service which processes customer orders. The class has a single method, processOrder(), which takes an Order as its only argument.

```

package uk.ac.glasgow.oose2.soa;

public class OrderService {
    public String processOrder(Order order){
        String sep = System.getProperty("line.separator");

        String response = "Hi, " + order.getCustomerName() + "!" +
            sep;

        response+="You seem to have ordered the following:" + sep
            ;

        for (String item: order.getItemCodes()){
            response+= sep+"item: "+item;
        }

        response+=sep + sep +
        "if this has been a real order processing system, " +
        "we'd probably have charged you about now.";

        return response;
    }
}

```

The order service is setup in much the same way as the one seen above using WSDD. However, we also need to tell axis how to deserialize an incoming XML formatted order to a Java instance. This is done in the WSDD file, which is used when the service is deployed. The code below shows how to adapt the WSDD file to register the type mapping from XML to Java for the service.

```

<deployment
    xmlns="http://xml.apache.org/axis/wsdd/"
    xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">

    <service name="OrderProcessor" provider="java:RPC">
        <parameter name="className"
            value="uk.ac.glasgow.oose2.soa.OrderService"/>

```

Example Java Bean Service - OrderService (511)

Mapping Beans in the WSDD File (512)

```

<parameter name="allowedMethods" value="processOrder"/>
<beanMapping
    qname="myNS:Order"
    xmlns:myNS="urn:BeanService"
    languageSpecificType=
        "java:uk.ac.glasgow.oose2.soa.Order"/>
</service>
</deployment>

```

The key extension is the `<beanMapping/>` tag, which states that any XML values of data type `Order` in the name space `myNS` should be mapped to the Java language type `uk.ac.glasgow.oose2.soa.Order`.

The final step is to alter the client so that it knows how to serialize the complex data types to XML using a `BeanSerializerFactory`.

```

Order order = new Order();
order.setCustomerName("Jim");
order.setItemCodes(new String [] {"644"});

Service service = new Service();
Call call = (Call) service.createCall();

QName qn = new QName("urn:BeanService", "Order");
BeanSerializerFactory bsf =
    new BeanSerializerFactory(Order.class, qn);
BeanDeserializerFactory bdsf =
    new BeanDeserializerFactory(Order.class, qn);
call.registerTypeMapping(Order.class, qn, bsf, bdsf);

String serviceURL =
    "http://macdonald:8080/axis/services/OrderProcessor";

call.setTargetEndpointAddress(new URL(serviceURL));
call.setOperationName(
    new QName("http://DefaultNamespace", "processOrder"));

String returnValue = (String)
    call.invoke(new Object [] {order});

```

Much of the code is familiar, including the setting up of a service call and the invocation of that call with appropriate arguments. The additional code creates a `BeanSerializerFactory` for the `Order` class and the xml `Order` type. This factory is responsible for serializing instances of `Order` to XML. The factory is registered with the service call, before the call is invoked.

## Summary

Selecting the right framework will depend on the particular characteristics required for your distributed application.

This chapter introduced the concepts of software frameworks, and used the Java Swing GUI framework as an example of a framework, with hooks, slots and toolkits.

The axis web service container and SOAP protocol implementation is an example of a distributed middleware framework. There are many other such frameworks, such as:

- Java Remote Method Invocation;
- Common Object Request Broker Architecture (CORBA);
- Java NetBeans;
- Google Protocol Buffers;
- Java Messaging Framework; and
- Globus Grid Toolkit.

Other Java supported  
middleware  
frameworks (514)



# Chapter 18

## Using Design Patterns

### Recommended Reading

PLACE HOLDER FOR lethbridge05object

Recommended reading  
(515)

PLACE HOLDER FOR gamma94design

### 18.1 Describing and Choosing Pattern

Many small scale design problems in software engineering re-occur frequently from project to project. Many of these problems concern the suitable arrangement of classes to provide a particular feature, or ensure the system exhibits a required non-functional property. When addressing these problems, software designers are concerned with ensuring good quality design that eases maintenance and future evolution of the software design.

Since the problems re-occur from project to project, it perhaps isn't too surprising that the same solutions are often applied as well. When a solution to a re-occurring design problem becomes well established, i.e. it is an accepted solution amongst professional software engineers for that problem, it can be documented as a *design pattern*.

You should think of software design patterns as templates that can be readily adapted to solve a general design problem in a specific context. This is rather like design patterns in tailoring. When a tailor makes a new suit, the tailor doesn't normally choose a design themselves. Instead, the customer is asked to select a style from a catalogue. A template for the parts of the suit is then applied to the material chosen by the customer, and the exact sizes adjusted to ensure a good fit.

Just as for tailoring, there are also catalogues of software design patterns. Each pattern in a catalogue is described in a consistent manner. This includes the:

Design patterns –  
think *tailoring* (516)

Describing design  
patterns (517)

- context and problem, giving the general circumstances in which it is appropriate to use the pattern;
- forces or constraints, describing the design principles that must be considered when addressing the problem;
- solution including a class diagram showing the *participant* classes in the pattern and how they *collaborate*;
- a rationale, explaining why the proposed pattern is a good solution to the general problem and how the forces are reconciled. In addition, it may be necessary to explain any trade-offs or limitations for the system as a result of applying the pattern; and
- any related patterns and/or anti-patterns.

A pattern description should also reference external documentation, if it has been proposed by someone else. All the patterns in this chapter come from the text books referenced in the recommended reading (although in some cases I've adapted the class diagrams that describe them).

*Anti-patterns* are common re-currences of poor solutions to problems addressed by real design patterns. An anti-pattern can often appear to be a reasonable solution a design problem, or superficially as an extension to an existing design pattern. However, anti-patterns often take legitimate design principles to extremes, and as a consequence end up causing more difficulty than the problem they were supposed to address.

Some examples of anti-patterns that will be referred to later on, when discussing real patterns are:

- the inner platform effect;
- excessive sub-typing based on variable value partitioning;
- database as messaging platform;
- abstraction inversion;
- magic push button;
- object orgy; and
- the God object.

The following sections present a catalogue of design patterns, according to three categories of pattern: structural, creational and behavioural. For each pattern, the general problem and design considerations are described, with an example of the problem. Then the solution is presented and applied to the example problem. Where appropriate, an anti-pattern is also described.

## 18.2 Structural Patterns

Structural patterns describe ways of arranging relationships between classes. We have already had a brief look at the first two structural design patterns in Chapter 13. Structural patterns can often be applied during the refinement stages of analysis because one of their applications is the appropriate representation of aspects of the problem domain in a class diagram.

### 18.2.1 Abstraction – Occurrence

We have already seen an example of the abstraction-occurrence design pattern in Chapter 13. Figure 18.1 illustrates an example problem where the abstraction-occurrence pattern applies. A system is being developed to manage the delivery of courses in a university. The figure illustrates two instances of the Lecture class. Notice the use of stereotyped dependencies to indicate that two instances are of type Lecture. The class has attributes for the location of the lecture, the day, start and end time, course and lecturer.

Example (519)

The lecture course is scheduled in a timetable so that there are a number of lectures each week, occurring on the same day, at the same time and in the same location. This information must be replicated for each instance of Lecture that fits into a particular period in the timetable has the same information. Even worse, if that information changes (the lectures are re-scheduled to take place between 12 and 1pm, for example) then the information must be changed in each instance of Lecture that is affected.

The general problem addressed is as follows.

Problem (520)

- Groups of instances of a particular class share the same values for some

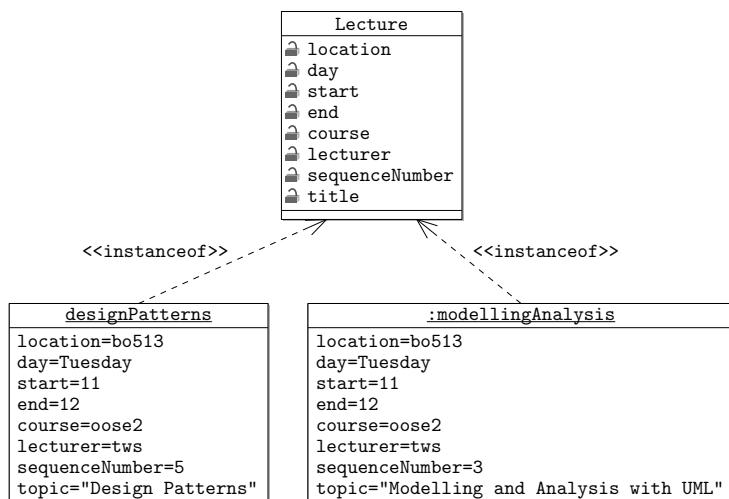


Figure 18.1: repeated data values in instance attributes

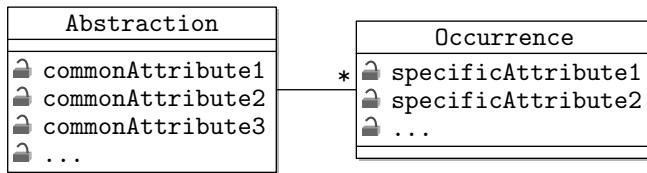


Figure 18.2: the abstraction occurrence design pattern

attributes;

- the data stored as attribute values is replicated across many instances;
- when an attribute value is changed, the change must be replicated across many different instances; and
- generalizing the attributes into a super class won't help, and violates the *is-a* rule.

Figure 18.2 illustrates the class diagram template for the abstraction-occurrence pattern that addresses the problem. All attributes that are replicated between instances are collected into an *abstraction* class. All attributes that can be different from one instance to the next are left in the *occurrence* class. A one-many relationship is drawn between the abstraction and its occurrence, indicating that many occurrences will share the common data stored in the abstraction.

Now, if some of the data in the abstraction changes, this data will only have to be altered in one instance. Figure 18.3 illustrates the application of the abstraction-occurrence pattern to the lectures problem.

The class diagram illustrated in Figure shows the re-organisation of the application's structure. The attributes that are shared between lectures in the same timetable period (*location*, *day*, *start*, *end* and *course*) are grouped together into a new abstraction class called *TimetableLecture*. The attributes that describe the particular lecture that will be delivered in a specific period/week combination are placed left in the *Lecture* occurrence class.

Figure 18.3(b) illustrates how the instances from Figure 18.1 are altered by the application of the pattern. A new instance of *TimetableLecture* is shared between the two instances of *Lecture*, storing the common attribute values.

The abstraction-occurrence pattern can be extended in more than one direction if necessary. Figure 18.4 illustrates an extension of the pattern for the lecture example. The figure shows that details about the course that a lecture is associated with have been added to the class diagram. Notice that a *Course* is modelled as an aggregate of *Lecture*. These two classes represent an abstraction for a lecture course that runs in successive years. The occurrences are the:

- *CalendarCourse*, for which there will be an instance for each of the academic years in which the course runs; and

Solution: abstraction – occurrence (521)

Solution: abstraction – occurrence (522)

Extended solution (523)

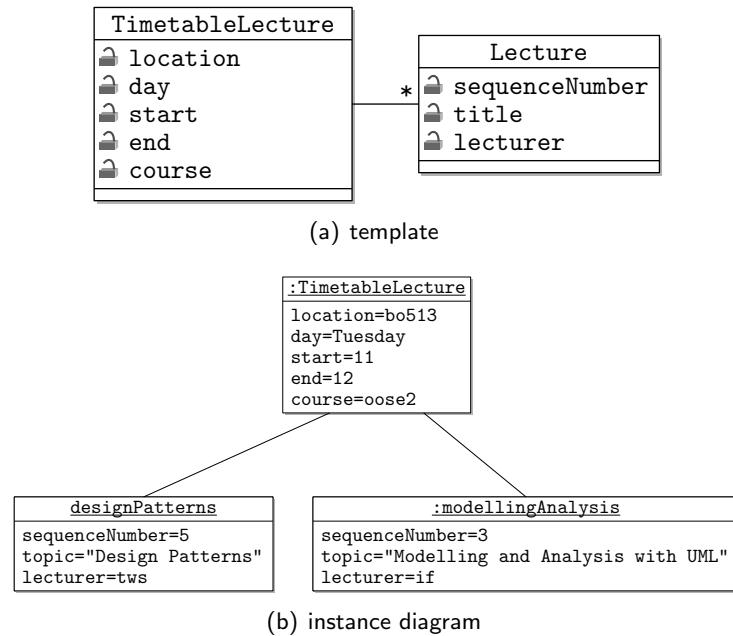


Figure 18.3: applying abstraction occurrence to lectures

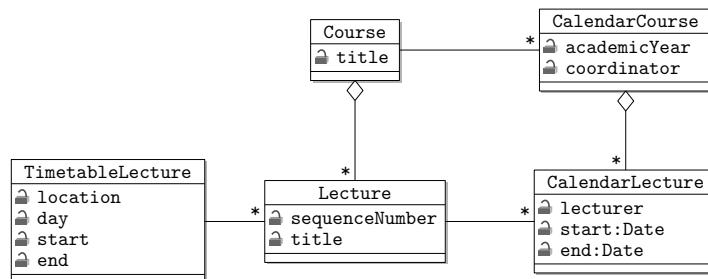


Figure 18.4: extending the abstraction occurrence pattern

- `CalendarLecture`, for which there will be a separate instance for each lecture that takes place for a course in an academic year.

Notice also that the `CalendarCourse` is an aggregate of `CalendarLecture`, just as for the `Course` and `Lecture` classes.

In summary, the figure now illustrates three applications of the abstraction-occurrence pattern: `TimetableLecture` to `Lecture`; `Lecture` to `CalendarLecture` and `Course` to `CalendarCourse`.

### 18.2.2 Player – Role

Example (524)

As for, abstraction-occurrence, the player-role pattern was first introduced during the discussion of class diagram refinement in Section 13.3.4. Another example is shown in Figure 18.5.

The figure illustrates a model developed to manage mortgages on properties for a financial institution such as a bank or building society. The model is an attempt to capture two ways of classifying mortgages in a single inheritance hierarchy:

- whether the mortgage is fixed rate or variable rate of interest. Fixed rate mortgages mean that a customer pays a fee to secure a guaranteed interest rate for a set period (say three, five or seven years) before transitioning to a variable rate. A variable rate mortgage follows the interest rate set by the country's central bank; and
- whether the mortgage requires repayment of the full value loaned, or just the interest calculated on the loan. Interest only mortgages have lower monthly repayments, but the customer never ends up owning the property.

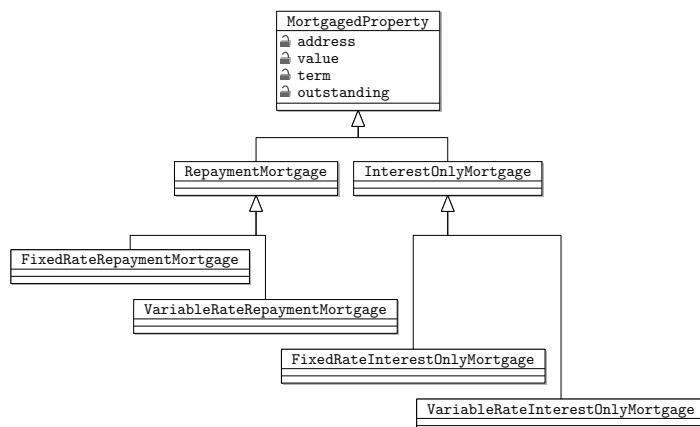


Figure 18.5: a complex inheritance hierarchy

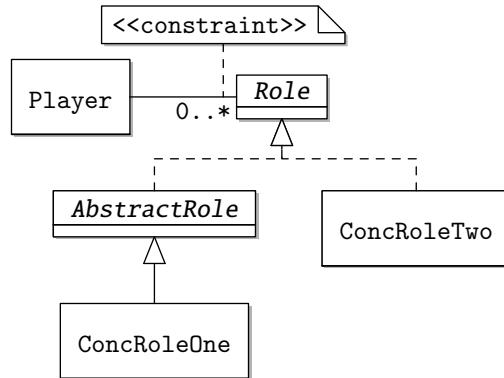


Figure 18.6: the player-role design pattern

The model is problematic because the two different types of interest calculated must be replicated across two concrete classes each. Re-ordering the specialization won't help because we would need to replicate the functionality for payment type across two sub-classes each.

The general problem addressed by player-role is as follows.

Problem (525)

- A class could legitimately be modelled as a sub-class of two or more super-classes; and/or
- some of the instances of the sub-class may not exhibit the behaviour of the super-classes all of the time.

One solution to the problem is to apply multiple inheritance. However, this is generally best avoided, since it adds complexity to the model. The player-role pattern provides a better solution, by separating some of the functionality of a *player* class into a *role*. Roles can then be assigned to *player* classes as necessary during the life-cycle of an application. Figure 18.6 illustrates the template for the player-role pattern.

The figure shows that the *player* class is associated with a number of *roles*. The relationship between a *player* and a *role* (one-one or one-many) is application specific, and there may be further, more complex constraints that must be applied in particular contexts, as shown by the constraint note (a *player* cannot play two particular *roles* at the same time, for example).

The *role* itself is normally an interface or abstract class that will be realized or completed by a number of concrete classes. In some cases, it may be useful to specify an abstract *role* that realizes some of the general behaviour of the interface on behalf of a collection of concrete *roles*.

Figure 18.7 illustrates the application of the player role pattern to the mortgage model. A new class *Property* is introduced to store information about properties on which the financial institution has provided a mortgage. The *Mortgage* class becomes an abstract *role* for the *Property* *player* class, with two

Solution: player – role  
(526)

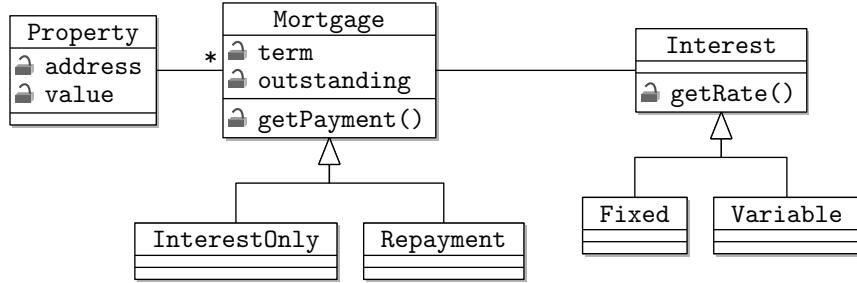


Figure 18.7: applying the player-role pattern to mortgages

concrete classes implementing the behaviour of the two different types of payment mechanism. The interest rate calculated for the mortgage is re-modelled as an abstract role of the mortgage class, with two different concrete classes for the two ways of setting interest rates.

Solution: player – role  
(527)

Notice that there are consequently two different applications of player-role in this model: **Property** to **Mortgage** and **Mortgage** to **InterestRate**. However the multiplicity of the two player-roles relationships is different. A property can have more than one mortgage placed on it. However, each mortgage can only have one way of determining interest at a time.

### 18.2.3 Adapter

Re-use of pre-existing functionality (in the form of frameworks, libraries and components) is strongly encouraged in object oriented software engineering. Reuse ensures that effort is not wasted on re-implementing common aspects of application functionality (such as security functions, logging and so on) whilst also helping to improve the the maturity and hence quality of the reused software. However, re-use requires that the pre-existing component be incorporated into the new application. This can be disruptive if substantial modelling and implementation has already taken place.

Example (528)

Figure 18.8 illustrates an example of this problem. The abstract **Match** class includes a method for determining the winner of the match. The sportster framework is being instantiated for a cricket league that uses the Duckworth-Lewis method to determine the winner of the cricket match (the method is a

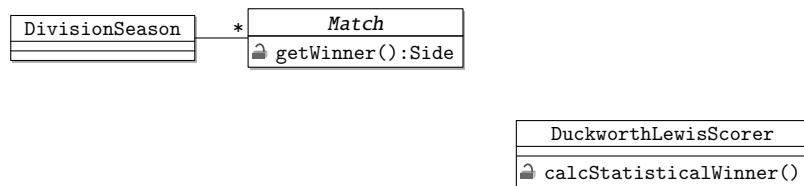


Figure 18.8: a software re-use problem

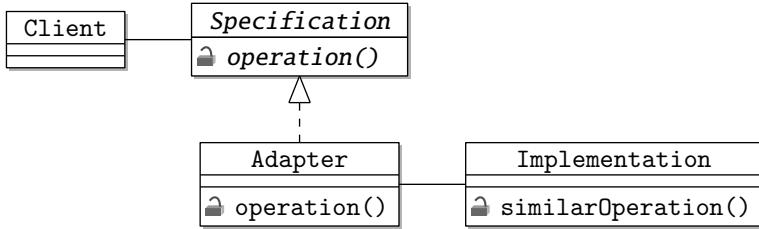


Figure 18.9: the adapter design pattern

complex formula that can be applied to a cricket match score if there is no clear winner). Consequently, it is necessary to implement a sub-class of the Match class that implements the Duckworth-Lewis method.

It is discovered that a pre-existing class DuckworthLewisScorer already provides this functionality, so ideally, it would be better to integrate this class into the framework, rather than attempt a complete re-implementation. However, the class is not a sub class of Match and cannot be altered to fit with the sportster application.

The general problem addressed by the adapter design pattern is as follows. Problem (529)

- A client class requires access to some functionality via a particular collection of operations (e.g. in an abstract class or interface);
- re-use of another classes that provides the functionality would reduce repetition and development costs; but
- the functionality is provided not via the prescribed operations.

Figure 18.9 illustrates the adapter design pattern template. The client class interacts with the specification of a service, given by an interface. The adapter realizes the interface by wrapping the functionality of the Implementation with the correct operation name. In this arrangement, the client class has no knowledge of the implementation details of the Specification interface.

Figure 18.10 illustrates the application of the adapter pattern to the cricket match problem. The adapter is the CricketMatch class, which realizes the Match interface by wrapping the functionality of the DuckworthLewisScorer class.

Solution: adapter pattern (530)

Solution: applying the adapter pattern (531)

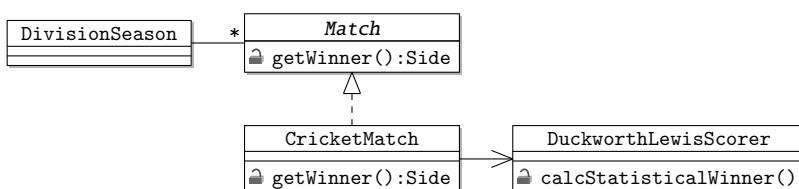


Figure 18.10: applying the adapter pattern to support re-use

The adapter pattern solves a different problem than delegation. In delegation, some of the internal functionality of a class is provided by the delegator. The delegator may provide other functionality as well. Adapters are used when a client needs to access services through an already specified interface. The adapter provides access to the functionality for the service via the interface and should do nothing else.

### 18.2.4 Proxy

Sometimes it is not appropriate for one class to interact directly with another. Figure 18.11 illustrates an example of this problem for a centralized authentication service. The service is accessed by a number of clients to authenticate users from different applications and platforms. There might be a web browser client and a mobile phone application client, for example. A decision is made for auditing purposes to introduce a logging mechanism of all attempts by clients to authenticate users.

Example (532)

A logging framework is identified as being suitable for tracking activity (represented by the `Logger` class). However, it is not desirable to alter functionality of the `AuthServer` service, by implementing the logging mechanism directly, because not all projects using the `AuthServer` will want the logging feature.

Problem (533)

The general problem addressed by the proxy pattern is as follows.

- There is a need to mediate access to the instances of a class from a client, because (for example):
  - the objects may be not be stored directly within the same environment as the client;
  - the objects are *heavyweight*, imposing considerable run time cost to load into the local environment; or
  - we wish to perform extra actions when particular operations are invoked, which should be transparent to the client;

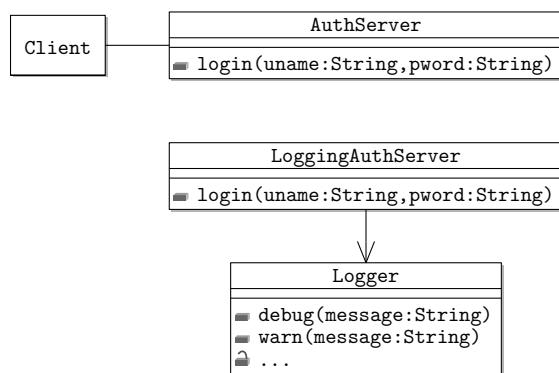


Figure 18.11: transparently inserting method call logging

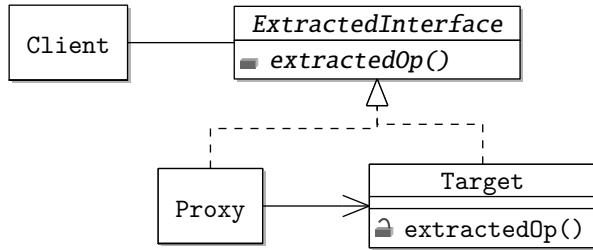


Figure 18.12: the proxy design pattern

- we wish to monitor all interactions with an object for debugging or auditing purposes;
- the exact implementation of the instances should be hidden from the client; and
- the implementation of the target class should not be altered.

The template for the proxy pattern which addresses this problem is shown in Figure 18.12. An extracted interface from the target service is provided to the client, rather than the target directly. Both the proxy and the target class realize this extracted interface and either can be provided to the client. When the proxy is provided it mediates direct access to the target service. Notice that the association between Proxy and Target classes is directed, the proxy knows about the target service, but not vice-versa.

Figure 18.13 illustrates the application of the proxy pattern to the authentication server problem. The `AuthServer` is now an extracted interface from an implementation, `AuthServerImpl`. The proxy is the `LoggingAuthServer` that will log any attempted authentications using the `Logger` before passing on the logged messages to `AuthServerImpl`.

Solution: proxy pattern (534)

Solution: applying the proxy pattern (535)

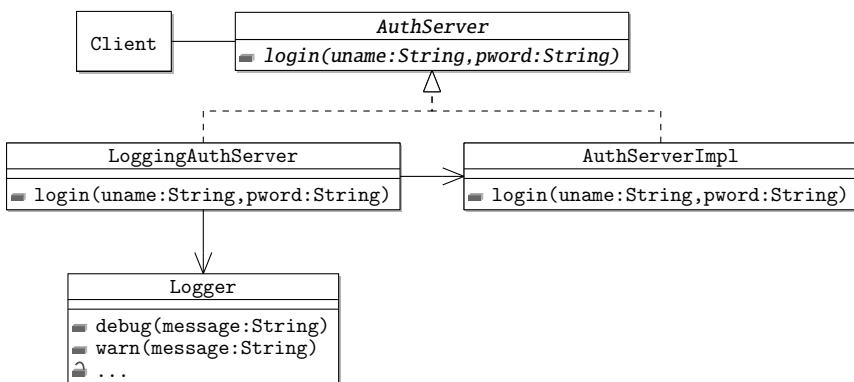


Figure 18.13: applying the proxy pattern

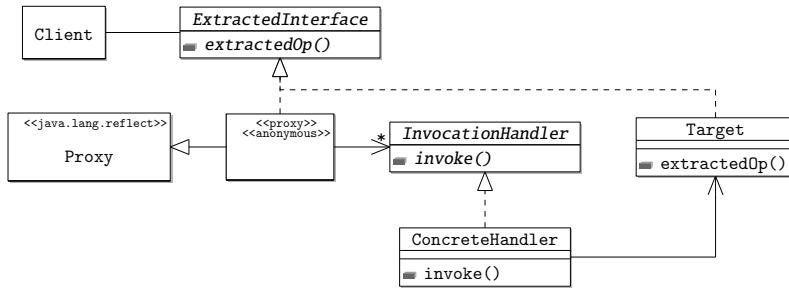


Figure 18.14: the Java reflected proxy mechanism

Note that the use of the proxy pattern typically requires the target instance to provide a particular interface - it can't just be accessed via its concrete class. The provided interface must often be explicitly extracted from the target class during the re-organisation of the design.

At first look, the proxy and adapter patterns seem to be rather similar. However, the clue to the different uses is the extra realization relationship between the target functional class and the extracted interface in the proxy pattern. In addition, there is an explicit association from the proxy to the target. Adapters are used to incorporate provided functionality into an existing system without disrupting the host system architecture. The proxy pattern is used to transparently *insert* extra functionality *between* two classes in a system.

The Java SDK contains facilities for automatically generating *anonymous* proxies via *reflection*. Reflection means the ability to programmatically inspect the structure of an object oriented program (and make changes to it) from within the same program during runtime. The reflection features of the Java SDK are collected into the `java.lang.reflect` package.

Figure 18.14 illustrates the general arrangement for constructing an anonymous proxy. As for the design pattern, the client interacts with an instance of an extracted interface, which is realized by the Target application. In addition, the `Proxy` class can be used to generate an anonymous proxy instance for any interface, using the static `getNewProxyInstance()` method. Notice that the proxy class has no name and is denoted as `<<anonymous>>` in the class diagram.

The proxy instance will pass any method calls to the specified interface to an instance of the `InvocationHandler` interface, which specifies a single `invoke()` method. Any `ConcreteHandler` class must realize the `invoke()` method. This implementation will contain the proxy actions to be taken, which may include passing the method call on to the 'real' target instance.

The code below illustrates how to generate a proxy for the authentication service problem. The first step is to specify the extracted interface, `AuthServer`:

```

public interface AuthServer {
    /**
     * @return true if the user exists and the password

```

## Proxies in Java (536)

### Generating proxies in Java: using reflection (537)

```

 * matches that of the specified user, else false.
 */
public boolean login(String uname, String pword);

```

and a corresponding target implementation:

```

public class AuthServerImpl implements AuthServer {
    private Map<String, String> users =
        new HashMap<String, String>();

    @Override
    public boolean login(String uname, String pword) {
        String actualPassword = users.get(uname);
        return (actualPassword == null) ? false :
            actualPassword.equals(pword);
    }

    @Override
    public void addUser(String uname, String pword) {
        users.put(uname, pword);
    }
}

```

Next, we must provide an implementation of the InvocationHandler to log calls to the AuthServer.login() method. The LoggingHandler class utilises the log4j logging framework to record login attempts, before passing them on to a target object:

Simple  
implementation (538)

A logging proxy (539)

```

public class LoggingHandler implements InvocationHandler {
    ...

    private static Logger logger = Logger.getLogger(
        LoggingHandler.class);
    {
        PropertyConfigurator.configure("config/l4j.properties")
        ;
    }

    private Object target;

    public LoggingHandler(Object target) {
        this.target = target;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object
        [] arguments)
        throws Throwable {

        String methodSig = method.getDeclaringClass().
            getSimpleName() + ".";

```

```

        + method.getName();

    // log the call
    logger.info("\nInvocation of method [" + methodSig + "]"
        on instance ["
        + target.hashCode() + "]. " + "\nConcrete type ["
        + target.getClass() + "] " + "\nArguments ["
        + Arrays.toString(arguments) + "].");

    // pass on the call to a the real target
    return method.invoke(target, arguments);
}
}

```

The parameters of the invoke method are:

- proxy the anonymous proxy instance on which the method has been invoked;
- method an object which provides details of the method which was invoked on the proxy (parameters, return type, annotations, owning class and so on); and
- arguments an array of arguments used to invoke the method on the proxy.

The first statement of the invoke() method constructs a string representation of the method signature that was invoked. The next statement logs the method that was called and the arguments used. The final line of code uses reflection to invoke the called method on the 'actual' target, using the same arguments as were used on the proxy.

Putting it all together  
(540)

The code below shows how the proxy arrangement is actually assembled.

```

public static void main (String[] args){

    //Get the application class loader.
    ClassLoader loader = Main.class.getClassLoader();

    //Create a target instance to be proxied
    AuthServer authImpl = new AuthServerImpl();
    authImpl.addUser("admin", "password");

    //Wrap the target instance with an invocation handler
    InvocationHandler invoc = new LoggingHandler(authImpl);

    //Specify the interface to be proxied.
    Class<?>[] proxiedInterfaces = new Class<?>[]{ 
        AuthServer.class};

    //Get a new proxy for the instance of AuthServer.
}

```

```

AuthServer authProxy =
    (AuthServer)Proxy.newProxyInstance(loader,
        proxiedInterfaces, invoc);

// Invoke login
authProxy.login("admin", "password");

```

The first statement obtains the default system *class loader*, an object responsible for loading other class definitions into memory. The next three statements set up the target *AuthServer*, adding a sample user, and construct a new *LoggingHandler* to wrap the target.

The next statement defines the interfaces that the anonymous proxy will realize in an array of class objects. Class objects are the reflective representation of classes in Java, providing information about a class' name, members, super class and so on. Class objects are used to represent both classes and interfaces.

Finally, the anonymous proxy is instantiated using the *newProxyInstance* () method, which requires a class loader, an array of interfaces to be realized and an *InvocationHandler*. The final statements tests the assembled proxy, producing the output shown below.

```

INFO 2011-02-16 19:01:37,242 main uk.ac.glasgow.oose2.patterns.
    proxy.LoggingHandler :
Invocation of method [AuthServer.login]
Instance [437220289]
Concrete type [class uk.ac.glasgow.oose2.patterns.proxy.
    AuthServerImpl]
Arguments [[admin, password]].

```

Sample output (541)

## 18.2.5 Façade

The general problem addressed by the Façade pattern is as follows.

Problem (542)

- A number of client classes manage similar interactions with a number of service classes;
- the service classes may be from several different packages;
- many of the actions taken require multiple repeated steps;
- it is desirable to ensure that the services are accessed in a consistent way, e.g. for logging purposes; and
- the current implementation means that there is extensive *coupling* between clients and service classes.

Figure 18.15 illustrates an example of the problem. A framework is being developed to provide a drive-by-wire control of a vehicle. The classes that have been developed to control the different *effectors* in the vehicle have been organised into a *package* called *car*.

Example (543)

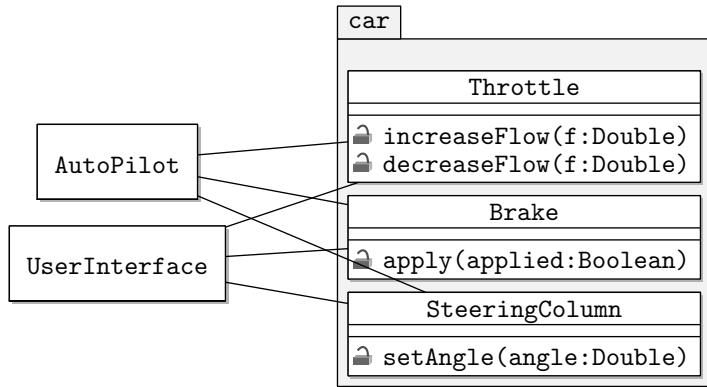


Figure 18.15: high coupling in a vehicle control system

Different client classes can use the package's class to control the car (AutoPilot and UserInterface classes are shown), but have to manage a complex sequence of actions to effect change in the car's state safely. For example, to slow the car down it is necessary to reduce fuel flow to the throttle and then apply the break. The code to effect these combined changes will have to be replicated in the different client classes, increasing the amount of cloning.

Figure 18.16 illustrates the façade pattern which addresses this problem. A façade groups together complex sequences of more primitive actions within a package and presents them as single operations to a client. In the diagram, a Façade interface is provided to the client classes. The implementation of the façade operations is provided in the FaçadeImpl class, which orchestrates the interaction across multiple classes in different packages.

In this arrangement, the interactions with the supplied packages are dra-

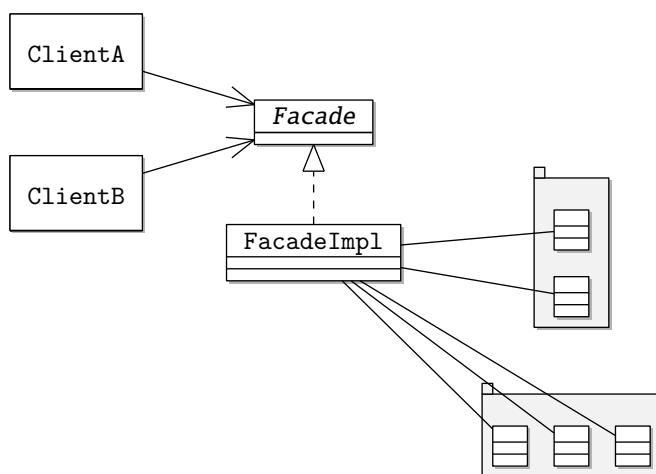


Figure 18.16: the facade design pattern

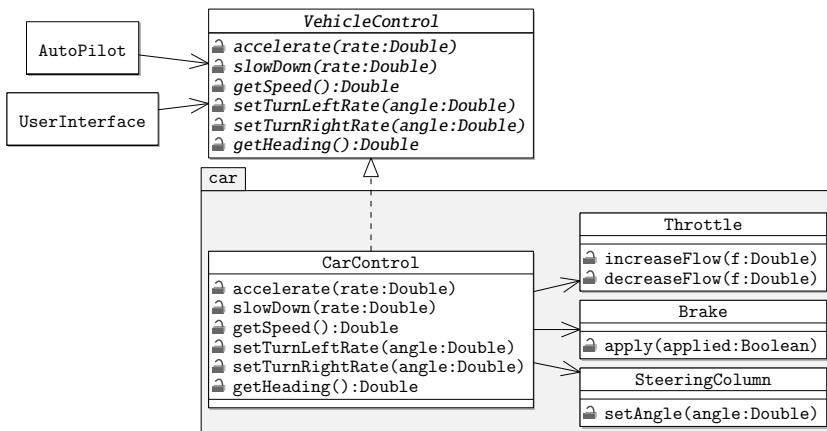


Figure 18.17: applying the facade pattern

matically simplified for the client classes. In addition, the `FacadeImpl` implementation details can be altered without having to make changes to the clients (provided the exported interface does not change).

Figure 18.17 illustrates the application of the façade template to the car control system. A new `VehicleControl` interface has been provided to the clients as the specification of the façade, which simplifies the control of the vehicle by declaring atomic operations that can be invoked to effect change in the car. A `CarControl` class realizes the interface for the `car` package, orchestrating interactions between the `Throttle`, `Brake` and `SteerColumn` effector control classes.

As noted above, the separation of the specification and implementation of a façade allows the underlying implementation of the façade to be altered without affecting the clients. In the example, an alternative package of effectors for a different vehicle type or model could be developed (say lorry, or even better speedboat!). The client classes could be used with these packages provided they realize the `VehicleControl` interface.

A common place to use a façade is between layers in a *layered architecture* (See Section 19.2.5). In this case, façades are used to specify the operations that a higher level layer can take with a layer at a lower level. Using a façade makes inter-changing different layer implementations easier.

Notice that the façade is a superficial layer on top of existing functionality: all the existing functionality is retained by the underlying classes. The façade is only responsible for coordinating between other classes to simplify access, it does not combine all the functionality within itself. This approach is called creating a *god object* and is considered an anti-pattern. The pattern arises because all the functionality of the underlying classes is combined into the single god class to simplify access.

Solution: applying the façade pattern (545)

Anti-pattern: the god object (546)

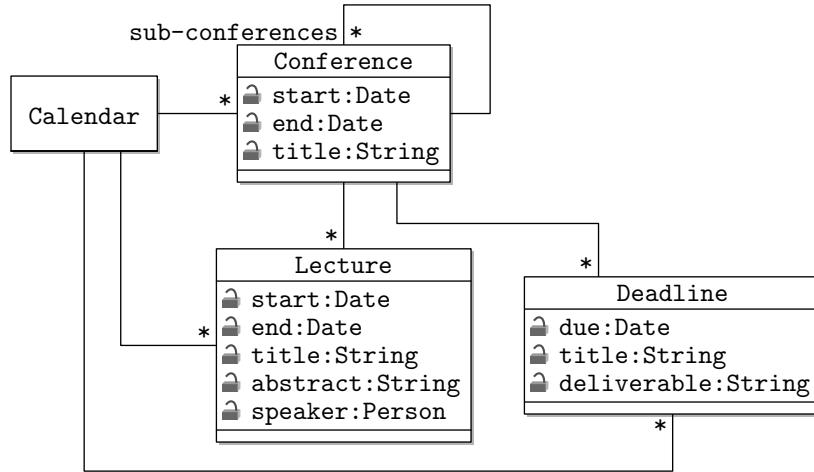


Figure 18.18: a highly coupled design with potential for generalization

### 18.2.6 Composite Pattern

Problem (547)

The general problem addressed by the composite pattern is as follows.

- The first iteration of a modelling process has produced a lot of classes with similar attributes.
- The classes represent different parts of an object hierarchy:
  - some of the objects are composites of the same class;
  - some of the objects are elements; and
  - some are both.
- It is desirable to present a common representation of the shared attributes, whilst enforcing the hierarchy rules.

Figure 18.18 illustrates an example of the problem. The figure shows a model for a timetabling system used to manage academic conferences. In an academic conference, researchers submit papers describing their work, which are then peer-reviewed by members of the conference committee. Papers accepted by the conference appear in the published proceedings and are presented by the authors in a lecture.

Example (548)

Key events in a conference are deadlines (for paper submissions, workshop proposals and so on, notifications of accepted papers), and the lectures that are delivered during the conference. Conferences, Lectures and Deadlines are all types of event with times and titles. The diagram shows that a Calendar has many Conference, Lecture and Deadline events, but must handle them all separately. In addition, a Conference has many Lecture and Deadline events.

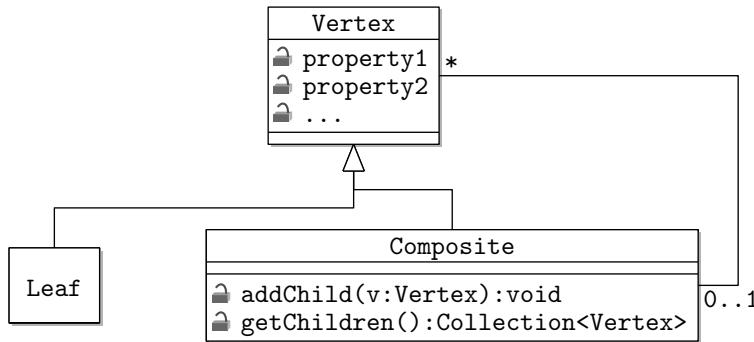


Figure 18.19: the composite (or general hierarchy) pattern

The result is a highly coupled model, with attributes with the same meaning cloned across different classes.

Figure 18.19 illustrates the template for the *composite pattern* (called the *general hierarchy* by Lethbridge and Laganière [2005]). The common properties of the model classes are collated into a single **Vertex** class. This class is then sub-classed by a **Leaf** and **Composite** type. The **Composite** type can contain many **Vertex** types, while the **Leaf** type cannot. However, both of the sub-classes inherit the shared properties from the **Vertex** class.

Figure 18.20 illustrates the application of the composite pattern template to the event management problem. The **Vertex** class in this case is an abstract **Event** class, which collates the common attributes of the different event classes: `start`, `end` and `title`.

Solution: composite pattern (549)

Solution: composite pattern (550)

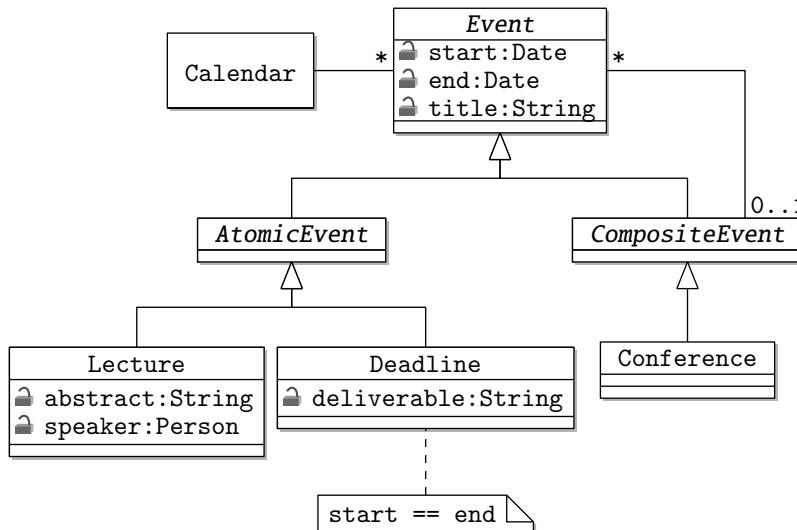


Figure 18.20: applying the composite pattern to the calendar problem

Two further abstract sub-classes are also defined, which represent the Leaf and Composite vertex sub-types in the template: `AtomicEvent` and `CompositeEvent`, respectively. This is an extension to the composite design pattern, which allows more than one type of atomic and composite event to be modelled. In the diagram, the two atomic event types are `Lecture` and `Deadline` (which store the event type specific attributes), and there is a single composite event type, `Conference`. The `Calendar` has been simplified to maintain a collection `Event` types.

Notice also that `Deadline` is constrained to have the same start and end date. Compromises of this sort are sometimes necessary in order to obtain the benefit of applying a design pattern. A good validation check for this arrangement is whether the semantics of the inherited (but additionally constrained) properties are still valid. A reasonable way of modelling an instantaneous event is having the start and end dates the same.

Visitor – visited (551)

The composite pattern is often used with the *visitor-visited* behavioural design pattern.

### 18.2.7 Read-only Interface

Problem (552)

The general problem addressed by the Read-only interface pattern is as follows.

- The objects of a class store potentially sensitive data;
- the data should be accessible for reading, however,
- the data should only be editable by clients with appropriate privileges.

Figure 18.21 illustrates an example of this problem in a user management framework. Two classes require access to the `User` class. The `Authenticator` class needs to make authentication checks when the system users log on. The `UserConfiguration` class needs to be able to alter the state of the `User` instance (presumably by authenticating). It is undesirable for the `Authenticator` class to have access to the `setPassword()` mutator method, since it is only responsible for checking whether the supplied password is correct.

Figure 18.22 illustrates the template for the read-only interface pattern which addresses this problem. The operations on the implementation which only access (but don't alter) the state of an instance are extracted to a `ReadOnly` interface, which is supplied to the unprivileged client classes. Operations that can alter the state of the implementation are grouped into the `Mutable` interface, which extends the `ReadOnly` interface.

The diagram illustrates an extension to the pattern proposed by Lethbridge and Laganière [2005], by making the `MutableInterface` features of the implementation an interface as well.

Figure 18.23 illustrates the application of the read-only pattern to the user management framework. All the methods of the `User` class have been extracted

Example (553)

Solution: read-only interface (554)

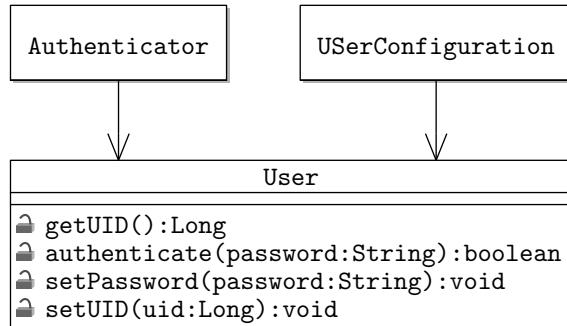


Figure 18.21: a user management class

as interface operations. The `getUID()` and `authenticate()` accessor operations are grouped together in the read-only `AuthUser` interface, while the `setPassword()` and `setUID()` operations are grouped together in the `ConfigUser` mutable interface. The `Authenticator` class only has access to the accessor operations of the read-only `AuthUser` interface, while the privileged `UserConfiguration` class has access to the additional mutator operations in the `ConfigUser` interface.

Solution: read-only interface (555)

## 18.3 Creational Patterns

Creational patterns address problems concerning the appropriate construction and access to objects in an object oriented system.

### 18.3.1 Singleton

The general problem addressed by the singleton pattern is as follows.

Problem (556)

- An entity is best represented as the only instance of a class (the set of instances of the class is a *singleton*);

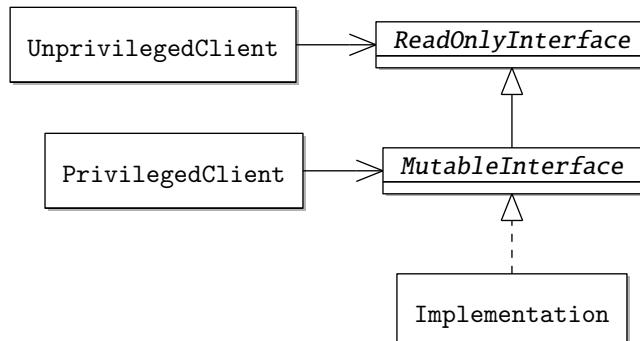


Figure 18.22: the read-only interface design pattern

- we need to define a constructor so that the singleton member of the class can be initialised;
- and we need to provide access to the singleton member;
- however, we need to restrict access to the constructor, so that clients cannot create further instances of the class.

Figure 18.24 illustrates two examples of problems addressed by the singleton pattern. The University class is part of a lecture course management system. The system is used within a single university, whose attributes (principal, address and even name) can change over time.

The Report and Glossary classes are part of a document management system used in a technical organisation. The organisation wishes to maintain a consistent glossary of reports across all reports (so that the organisation establishes and maintains a common language concerning its work). However, the terms stored in the glossary will be added to over time (as new reports are written) and the definition of some terms may also evolve.

#### Example (557)

Notice that an instance doesn't have to be a singleton in the real world, as for the examples shown in Figure 18.24. There is more than one university, for example. Using a singleton states that for the purposes of the *problem domain*, which describes a part of the real world, there can only be one instance of the singleton represented in the system.

Figure 18.25 illustrates the template for the singleton design pattern which addresses this problem. A singleton class is defined with:

- a single private constructor, which is thus only accessible from within the class itself;
- a private static class attribute, `singletone`, of its own type which is used to store the singleton instance; and

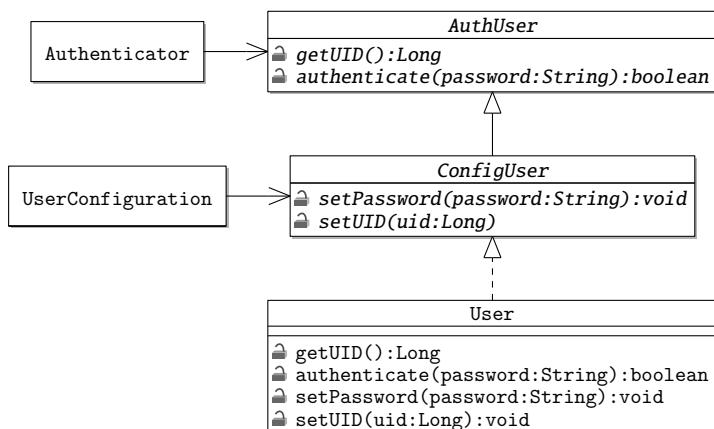


Figure 18.23: separating privileges on the user class

- a public static method, `getSingleton()`, which provides access to the singleton attribute.

The static attributes of a class are denoted in a class diagram by underlining their signatures. The Java code to implement the singleton template is shown below.

```
public class Singleton {
    private static Singleton singleton;

    private Singleton() {
        // perform any initialization
    }

    public static Singleton getSingleton() {
        //handle for first call
        if (singleton == null)
            singleton = new Singleton();
        return singleton;
    }
}
```

Solution: singleton  
(558)

Notice that the `Singleton` instance is only created if the `getSingleton()` method is called. This is called *lazy instantiation* and is appropriate if the singleton instance may not be accessed in a particular invocation of the program, and particularly so if instantiating the singleton is computationally expensive. The alternative *hasty instantiation*, by initialising the `singleton` attribute when the `Singleton` class is loaded is useful if it is expected that the singleton will certainly be accessed during the invocation of the program.

Figure 18.26 illustrates the application of the singleton pattern to the glossary and university problems. The term `singleton` is replaced with the class

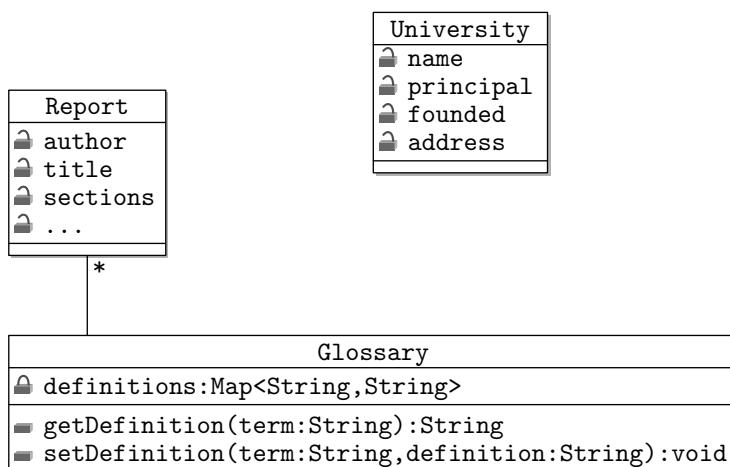


Figure 18.24: two example singleton problems

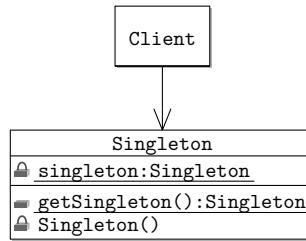


Figure 18.25: the singleton design pattern

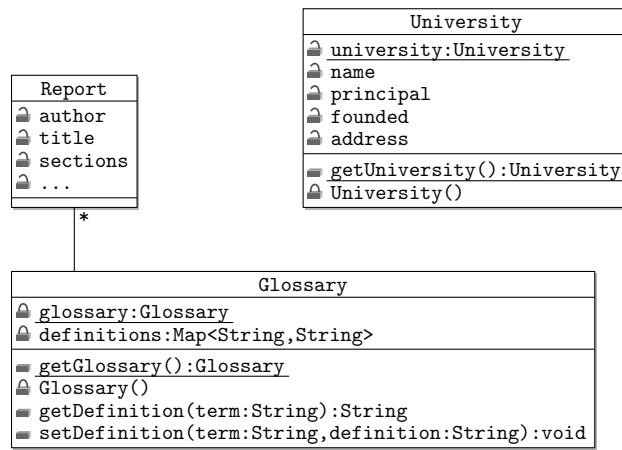


Figure 18.26: enforcing singleton behaviour on the university and glossary

names for the private constructor, private static attribute and public static method. Notice that in both cases, instance operations and methods can still be defined as appropriate for the class. A university still has a name, principal, date of founding and address; and a glossary maintains a map from terms to descriptions.

A disadvantage of the singleton pattern is that the constructor cannot be parameterised outside of the class. If the singleton needs to be configured, this must either be hard coded into the `getSingleton()` operation, or else performed internally, by reading a configuration file, for example.

Note that the `enum` type (see Section 2.4.2) is a generalization of the singleton design pattern - a fixed number of pre-defined instances can be created for a particular enumeration type.

### 18.3.2 Factory

Solution: applying the singleton pattern (559)

The general problem addressed by the factory design pattern is as follows.

- A client needs to construct instances that will provide some functionality as part of the client's work;

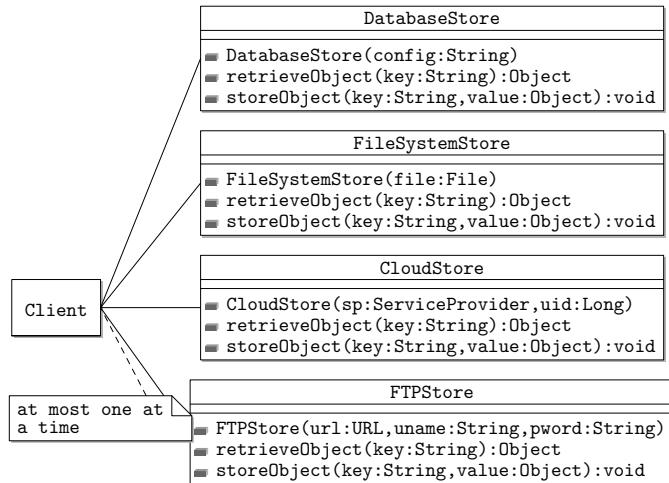


Figure 18.27: problem

- the clients will use all the instances in the same way;
- however, the behaviour of the instances will change depending on the context in which the application is used.

Figure 18.27 illustrates an example of the problem addressed by the factory design pattern. A client application needs to be able to be able to store *persistent* instances between invocations of an application. The application is used on different platforms, for which a variety of different storage platforms can be appropriate (in a database, on the file system, with a cloud storage provider, or on an FTP server). Ideally, we would like to be able to alter the storage mechanism used by the client between platforms without altering the client itself.

Example (561)

Figure 18.28 illustrates the template for the factory pattern which addresses this problem. The client is provided with the an implementation of the Factory interface, (in the diagram an instance of either ConcreteFactoryA, or ConcreteFactoryB). The interface specifies an operation, `createProduct()` for generating instances of classes which implement the Product interface (either ProductA! or ProductB).

Solution: factory pattern (562)

The factory Product specifies the generic required functionality needed by the Client, with the implementation details provided by the classes which realize the Product interace. The Client can use the provided Factory to generate instances which provide the necessary functionality, without knowing the precise implementation details. This avoids hard-coding the use of a particular constructor into the Client class.

Figure 18.29 illustrates the application of the factory pattern for two of the storage mechanisms (databases and the FTP server).

Solution: factory pattern (563)

Notice that, unlike the singleton pattern:

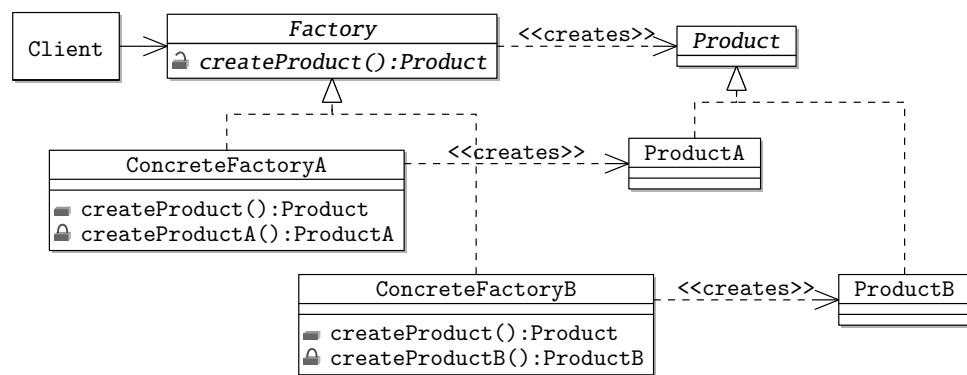


Figure 18.28: the factory design pattern

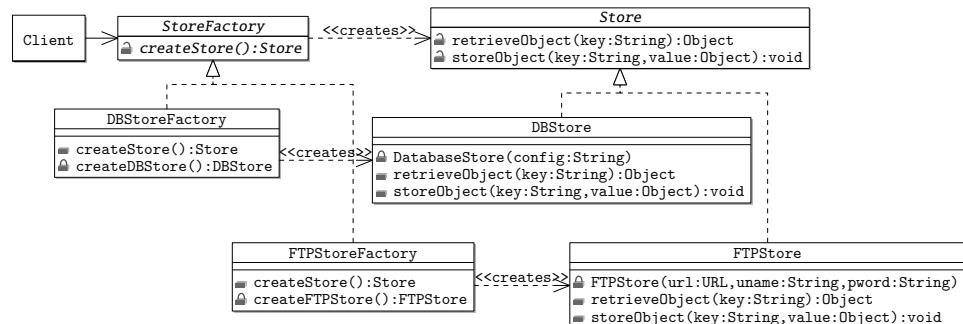


Figure 18.29: applying the factory pattern

- the creational method in the factory is not static, instead an instance of the factory is created to provide instances of the product; and
- each instance produced by the factory should be ‘fresh’, i.e. it should be a new object not previously provided to other clients.

There is a primitive variant of the factory pattern which utilises a static method in the factory class for generating instances of the product class. In this case, it is not necessary to instantiate the factory class before using it to create instances of the product. However, this does mean that the client class is coupled to the product produced by the factory, even if it doesn’t know the product’s implementation details. This approach is effectively a hybrid of the singleton and factory patterns.

## 18.4 Behavioural Patterns

Behavioural patterns describe solutions to problems regarding the interactions between instances in an object oriented program.

### 18.4.1 Iterable – Iterator

The general problem addressed by the iterator-iterable pattern is as follows.

Problem (564)

- Many different concrete and abstract data structures for aggregates; and
- need to loop over an aggregate of elements in a general way so that underlying implementation can change independently.

Figure 18.30 illustrates the template for the iterator-iterable pattern which addresses this problem. We have already seen how to use iterators in Java, when we reviewed the Collections framework (see Section 18.4.1). An `Iterable` interface is exported by any class whose instances contain elements which can be iterated over. These classes provide an `Iterator` via an implementation of the `iterator()` operation. Iterators define methods for iterating over the elements of an iterable instance.

Solution: Iterator – Iterable (565)

We already saw an application of the iterable-iterator pattern in the implementation of the `BagOfSweets` class in Section . Figure 18.31 shows the class diagram for another example:the application of the iterable-iterator pattern to the `Hashtable` class. The `Hashtable` class provides (and also defines as an inner class) the `HashIterator` class, which realizes the `Iterator` interface.

There is a variant of the iterator pattern, in which the iterator defines a different set of operations (where `T` is a generic type (see Tutorial 2.10):

Solution: applying Iterator – Iterable (566)

- `isDone() :boolean`, equivalent to negating the `hasNext() :boolean` operation defined in the Java iterator;

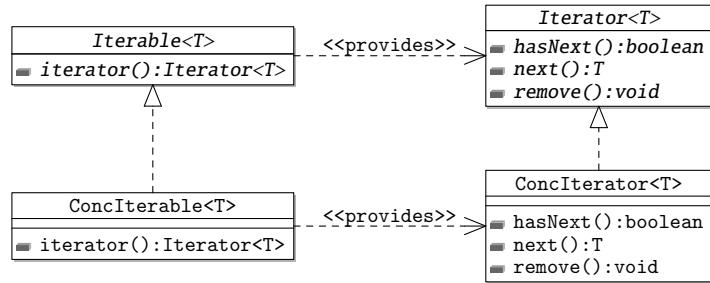


Figure 18.30: the iterable-iterator design pattern

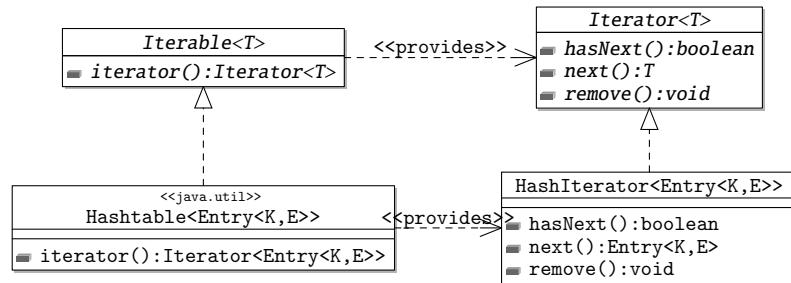


Figure 18.31: the Hashtable-HashIterator relationship

- `next():void` and `currentItem():T`, collectively equivalent to `next():T`; and
- `first():void`, which resets the iterator, and has no equivalent in the Java iterator.

The variant is known as the ‘Gang of Four’ (GoF) iterator, after the number of authors of the book it comes from, Gamma et al. [1994]. Conveniently, the iterator also has four methods, aiding memory!

### 18.4.2 Delegation

We’ve already come across how to denote this pattern in Chapter 13. The general problem addressed by the delegation design pattern is as follows.

- The functionality required by one class is already provided by another;
- re-using the provided functionality would reduce development costs, complexity, and replicated development;
- however, the classes cannot be related by an inheritance relationship because this would violate the is-a rule.

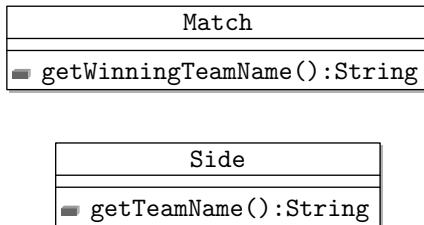


Figure 18.32: re-using functionality from another class

Figure 18.32 illustrates another example that can be addressed through delegation. The Match class defines a method called `getWinningTeamName()`. The responsibility for storing team names is already specified in the Side. `getTeamName()` method.

Figure 18.33 illustrates the general template for the delegation pattern. The delegating method invokes the delegated method and returns the result from the invocation. The note contains the code for the delegating method.

Figure 18.34 illustrates the application of the delegation pattern to the Match / Side problem. The Match class maintains a reference to the winningSide Side instance, which is the delegated responsibility for storing the name of the winning team.

An object oriented program must sometimes chain together sequences of delegations to realize some behaviour. In this situation, a delegate becomes a delegator in the chain of method calls. In each case, the delegator does not know how the delegate method is implemented. Figure 18.35 illustrates the chaining together of delegations for the Match class problem. The Match `.getWinningTeamName()` method invokes the Side.`getTeamName()` method, as before. In turn, the Side.`getTeamName()` method invokes the `getName()` method of its team attribute and this then invokes the `getName()` method of its club attribute.

Notice that delegation is often complementary to abstraction-occurrence. The occurrence uses the data held in common by the abstraction to fulfill it's own responsibilities.

The *object orgy* is an anti-pattern that resembles the delegation technique.

Example (568)

Solution: Delegation (569)

Solution: applying delegation (570)

Chaining delegation (571)

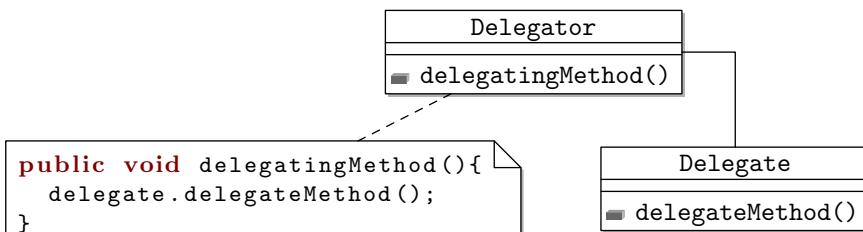


Figure 18.33: the delegation design pattern

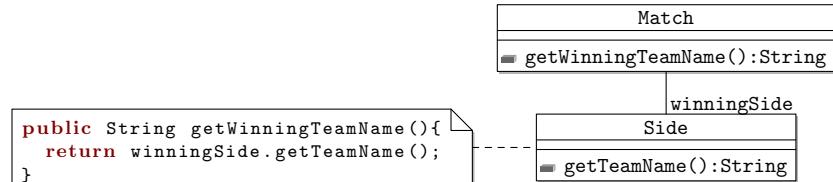


Figure 18.34: applying the delegation pattern to the winner problem

Anti-pattern: object orgy (572)

Problem (573)

However, rather than delegating explicitly from one method to another, the delegator directly accesses the fields of the delegate to obtain the data value required. This approach increases the coupling between classes, since the delegator must now know how the data is stored in the delegate. Worse, if the delegation is a chain, as described above, the coupling occurs across several classes that would otherwise be partitioned into separate parts of the program.

### 18.4.3 Observer – Observable

The general problem addressed by the observer-observable pattern is as follows.

- A class needs to be notified about changes in another class or classes;
- the changes occur on an infrequent or irregular (unpredictable) basis; and
- the detection of the condition when a notification should be sent should be separated from the response.

Figure 18.36 illustrates an example of this problem. An application is under development to manage alerts from different types of natural disaster. The EvacuationAlarm class has a method for sounding an alarm so that a local

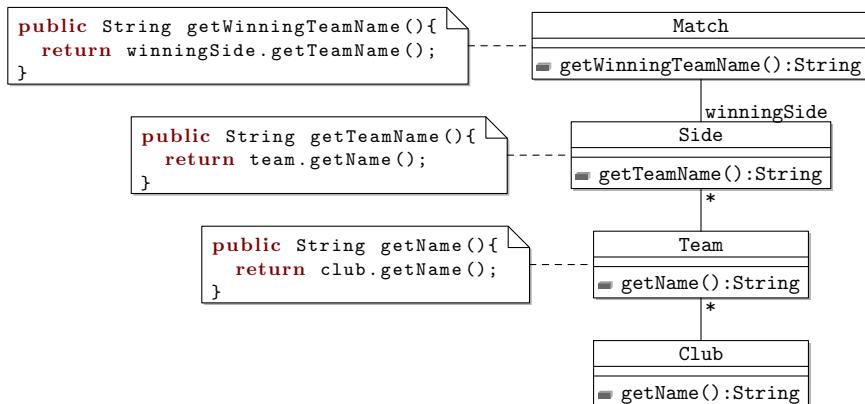


Figure 18.35: chaining delegation across several classes

WaterLevelSensor	EvacuationAlarm	Seismometer
■ getLevel():Double	■ soundAlarm():void	■ getEnergy():Double

Figure 18.36: the evacuation alert system problem

population knows it should leave a particular area because of the danger of a natural disaster.

The WaterLevelSensor is a *wrapper* class for a water depth sensor that can be placed on a river bank to monitor water levels. The `getLevel()` method will report the current water level whenever it is called. Similarly, the Seismometer class is the programmatic interface to a seismometer instrument that can be placed in or on the ground by a geologist. It has a method, `getEnergy()`, for reporting the current energy level of vibrations passing through the ground.

Example (574)

The problem faced by the developers is to identify the situations when water or energy levels in the respective sensors exceeds a safety margin and thus sound the alarm. The occurrence of an either situation is unpredictable (otherwise the alarm wouldn't be needed), necessitating the constant monitoring of water and energy levels. It is possible that further sensor types could also be added (for the presence of smoke, or wind speeds, for example), so if the implementation is all placed in a single monitoring class, it will have to be altered to manage each type of sensor (and threshold).

Figure 18.37 illustrates the template for the *observer-observable* pattern which addresses this problem. An observable is a class that represents some property that other classes need to be informed about. Instances of classes that realize the `Observer` interface can be registered with the `Observable` class via the `addObserver()` operation.

Solution:

Observer-Observable  
(575)

When the `Observable` class detects some phenomena that should be reported to the registered `Observer`, the `notifyObservers()` method is invoked. This method invokes the `notify()` operation of each of the registered `Observer` instances.

Notice that the `notifyObservers()` method is private, while the `notify()` operation is public. This is because the `notifyObservers()` method is a internal mechanism for reporting the phenomena, whereas, the `notify()` operation is the public part of the `Observer` interface - the `Observer` could receive notifica-

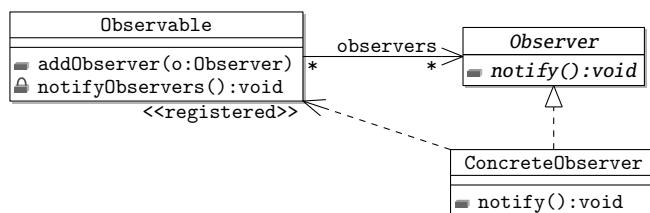


Figure 18.37: the observer-observable design pattern

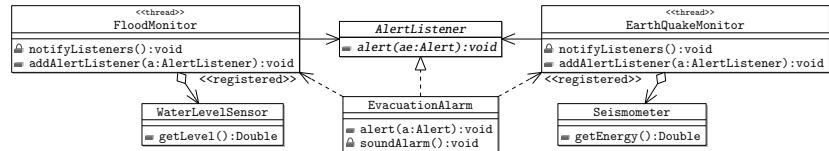


Figure 18.38: applying the observer-observable pattern to the alert system

Solution: applying  
observer-observable  
(576)

tions from many different sources.

Figure 18.38 illustrates the application of the observer-observable pattern to the evacuation alarm problem. The `EvacuationAlarm` class realizes a new observer interface, `AlertListener`, so that it can receive notifications.

An `EvacuationAlarm` instance is registered with two observable classes, `FloodMonitor` and `EarthQuakeMonitor`. The instances of both of these classes are *threads* (see Section 23.6), which continually *poll* a sensor instance, `WaterLevelSensor` and `Seismometer` respectively. Polling is the alternative approach for obtaining information from observing. Rather than waiting to receive a notification that something has happened, a polling thread continually and regularly samples the state of a phenomenon. Polling is appropriate in the opposite circumstances to observing: when the phenomena will change regularly and frequently (probably changing faster than the sensor can be polled).

When either observable thread detects a situation when an alert should be issued, all the instances registered as `AlertListener` instances are informed via the `notifyListeners()` method. Each instances receives a notification via the `alert()` method. Notice that this method is parameterised with the detail of the alert, encapsulated in an `Alert` class.

#### 18.4.4 Other Design Patterns

Other patterns worth investigating (577)

There are many other design patterns that we haven't mentioned yet. For example:

- Inversion of control
- Command
- Builder
- Template method

You should consider investigating these patterns as part of your reading time.

## Summary

Design patterns are accepted good solutions to re-occurring software design problems. The key to applying design patterns is a careful identification of the

problem context in order to select the appropriate pattern or combination of patterns to be applied. Be wary of developing anti-patterns within your software design, through the inappropriate or excessive application of otherwise good design principles.

## **18.5 Workshop: Inner Classes**

## 18.6 Exercises

1. Apply the abstraction – occurrence pattern to the following situations:
  - (a) A train journey class. Trains run on a regular daily timetable and have a driver, conductor, trolley server and pre-booked seats for each journey.
  - (b) A restaurant reservation class. The reservation includes the name and address of the restaurant, the number of diners, the date and time that the meal will begin and the table that will be used.
  - (c) A class for the programmes in a television schedule (such as you would find in a newspaper). Programmes have a date and time and title. A programme can be a episode from a series or one-off special.
  - (d) A class for recording the borrowing of books from a library. A library may hold several copies of the same book. The title and author of the book, the name and address of the borrower and the date of the loan must be recorded.

In each case, sketch the class diagram for the application of the pattern and the Java code.

2. Consider the following design problems
  - (a) A student should be notified whenever the deadline for a coursework item is due on a course they are registered for. Students should also be reminded of examinations the day before they are scheduled.
  - (b) A file system is organised into a hierarchy of directories. Both files and directories have a name, owner and set of permissions. Directories may contain other files.
  - (c) A company allocates responsibilities to employees, such as 'Finance Director' and Chief Executive Officer. Each formal responsibility is associated with a bonus (for performing that role). A single employee may have many responsibilities. A responsibility may also be shared between employees.
  - (d) A user interface is being developed for a GPS handset. The user interface is intended to work generically with a number of different handset models.

The user interface can be used to set way points along a route in advance of a journey. To do this, the user interface must get the current route, then determine the location of the new way point, then finally save the journey back to the device. The GPS must also play an alert when each way point along a current journey is reached.

In each case:

- state the design pattern(s) you would apply to solve problem;
  - give a justification for using the pattern(s), based on the problem context and the constraints that you identify; and
  - sketch a class diagram illustrating how the pattern(s) can be applied in this case.
3. Consider the domain model shown in Figure 18.39 developed as a solution to the problem described in Figure 13.26  
What design patterns could you apply to the model to improve the design? Explain how you would apply the patterns and what the resulting improvement in the design would be.
4. Develop a Java `InvocationHandler` which times the invocation of each call to an interface's operations and logs the result. Hint: review the Javadoc documentation of the `java.util.Date` class for a way of tracking timing information.
5. You are developing a class which extends the `AbstractCollection` class to store generic objects in a multi-set, called `Bag`. You discover a class which implements the mathematical properties of a multi-set, called `GoFBag`. Unfortunately, the `GoFBag` class has followed the Gang of Four iterator pattern, rather than the Java iterator. The `GoFBag` iterator class is called `GoFBagIterator`.
- (a) Sketch a class diagram showing the relationship between `GoFBag` and `GoFBagIterator`.
  - (b) What design pattern would you apply to incorporate the functionality of the `GoFBag` class into your application, without changing the specification of the `Bag` class?
  - (c) Sketch the code for `BagIterator` implementation of the `Iterator` interface, following the design you have adopted for incorporating the `GoFBag` functionality in your class.
6. The observer-observable pattern can be extended by using an abstract `Observer` class.

The abstract class still acts as a thread, and provides the functionality for registering and notifying observers of a particular general type of phenomenon. However, the `run()` method of the thread is left abstract.

Concrete classes then extend the abstract class by implementing the `run()` method to provide the functionality for monitoring the specific observable phenomenon and issuing notifications.

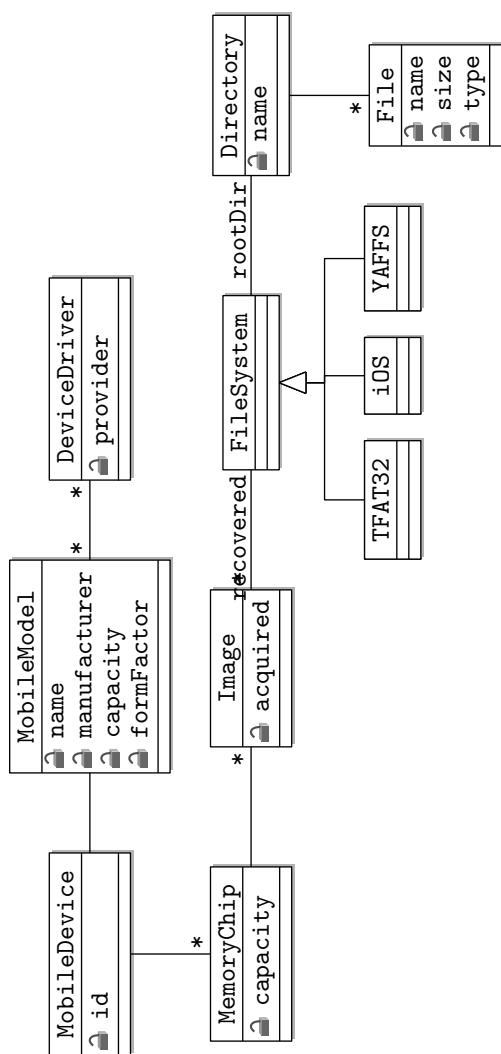


Figure 18.39: a domain model for the mobile forensics application

- (a) How would the class diagram for the general observer-observable template be altered to reflect this variant?
- (b) How could the evacuation system problem be adapted to use this variant of the observer-observable pattern?

Are there any disadvantages to this variant of the pattern?

7. A software development team is working on the configuration mechanism for their Java application. The team has decided that configuration will be managed in a class called Configuration. The operations already specified in the Configuration class are as follows:

```
public class Configuration {
    public void setOption(String option, Object value){
        //TODO
    }

    public ObjectgetOption(String option){
        //TODO
        return null;
    }
}
```

Consider the following three design problems:

- (a) The team discovers that classes which implement the `java.util.Map` interface provide much of the required functionality for storing and retrieving configuration options. The team want to integrate this functionality, but without changing the existing specification.
- (b) The team wishes to centralise all the configurable features for the application in a single instance of Configuration. The Configuration instance should be accessible from any other part of the program, but other parts of the program should not be allowed to create their own Configuration instances.
- (c) Different components in the application need to know when a option is changed in the Configuration instance. Options may be changed unpredictably and infrequently.

For each problem, name a design pattern as a suitable solution. In each case, describe how the design pattern is instantiated for the specific scenario (you may use pseudo code or a UML diagram, if appropriate) and briefly explain why you selected the pattern.

## Chapter 19

# Design Principles and Practice

### Recommended Reading

PLACE HOLDER FOR lethbridge05object

Recommended reading

### 19.1 Introduction

A *software system* is the delineation of a collection of software artifacts from an environment, with a name and an ascribed set of responsibilities. Figure 19.1 illustrates the relationship between a software system and its environment. A system is separated from its environment a *system boundary*. Artifacts within the boundary are within the remit of a system designer to specify and implement.

A system and its environment

A software system will export interfaces that may be used by other systems, but may also depend on interfaces provided by other systems. System artifacts close to or on the system boundary are called *boundary objects* and include user interfaces and interfaces to other systems. Artifacts outside the boundary which directly interact with a system are its *dependencies*. A software system will usually depend on the operation of a *hardware system*.

Notice that the separation of a system from its environment depends on the perspective of the system designer. For example, one designer may consider the

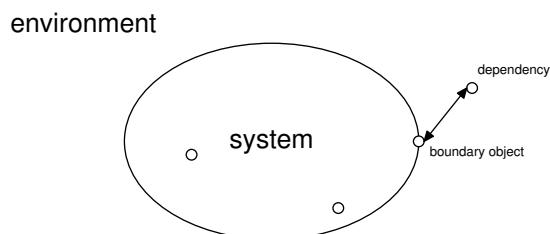


Figure 19.1: the relationship between a system and its environment

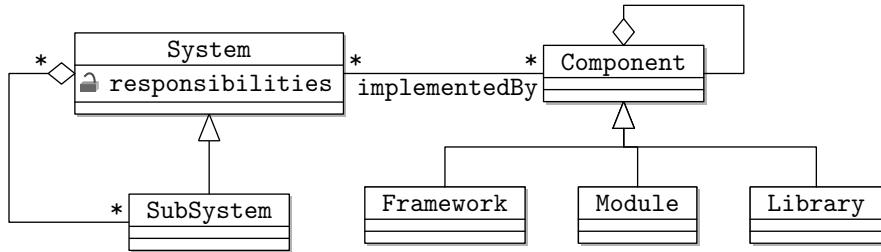


Figure 19.2: software system design terminology (re-drawn and adapted from Lethbridge and Laganière [2005, Figure 9.2, pp312])

development of a web browser application to be a system, which depends on an underlying operating system and collection of hardware drivers. Another designer may consider all these artifacts to comprise a single system. As discussed in Chapter 13, systems are mental constructs that are captured and analysed using models. The appropriate delineation of a system is dependent on the task at hand.

Figure 19.2, re-drawn and adapted from Lethbridge and Laganière [2005, Figure 9.2, pp312]), illustrates the terminology associated with software systems using a class diagram, in particular:

- 
- *Sub-systems* are systems in their own right that are composed into larger systems. Notice that this is a recursive definition. A sub system may be broken down into further sub-systems.
- *Components* are the concrete run time entities that realize the behaviour of systems. Many components are orchestrated to realize a system, and a component may be used in more than one system. Two types of component are *modules* which are programming language specific components, and *frameworks* which are used in multiple systems to provide re-usable functionality.

A software designer works by addressing unresolved design problems. A design problem describes the context in which a customer requirement must be implemented, and the general design principles and considerations that are applicable. In fact, we have already seen numerous examples of design problems when we reviewed design patterns in Chapter 18. Design patterns provide us with solutions to commonly recurring design problems in different contexts.

Unfortunately, there aren't design patterns to address every design problem, so sometimes we need to fall back on general principles of good design to guide us. A designer must take a design decision based on the:

- required functionality according to the agreed software requirements specification for the system;

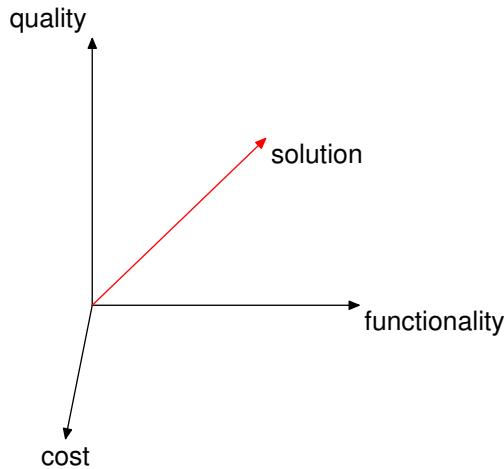


Figure 19.3: the cost-functionality-quality compromise in design

- general principles of good design which influence the overall quality of the system, improving the non-functional properties of the system and easing long term maintainability; and
- the budget available agreed in the project contract.

Figure 19.3 illustrates the trade off between these issues that must be identified by a designer. The three axis of the graph scope the *solution space* for a design problem. If the available budget for the project is increased, then the additional resources can be invested in improving the quality of the system, or adding features. Conversely, if a project is over-running, a decision must be made as to whether features should be dropped from the system, or the desired level of quality be reduced.

Pursuing high quality software design introduces extra costs to a project, because getting design decisions right first time is difficult, and more experienced software engineers are more expensive to employ (and even they won't necessarily get it right). However, there are long term benefits to a software project of high quality software:

- increased reliability
- faster delivery
- reduced maintenance costs
- improved changeability

As noted above, getting the design of a software system right first time is difficult. Consequently, software design is an iterative, problem solving task as illustrated in Figure 19.4. The steps of the software design process are:

Satisficing in design

General benefits of high quality design

The design process

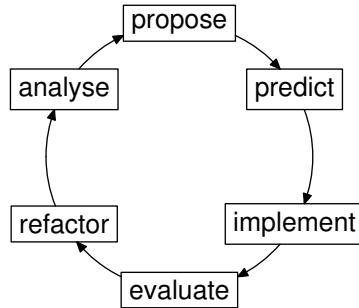


Figure 19.4: the design process

- *analysis* of the problem and its context drawn from the requirements specification, principles of design and available budget;
- *propose* a solution in the form of a design description, such as a class diagram, with accompanying rationale;
- *predict* the consequences (benefits and trade-offs) of applying the design solution, including for the non-functional properties of the system;
- *implement* the solution;
- *evaluate* the solution through testing against the original requirements; and
- perform *refactoring* on the design to address any identified deficiencies.

Over-arching the iterative approach to problem solving is the general strategy to be adopted for choosing which design problems to be solved first. There are two broad approaches to addressing design problems, as illustrated in Figure 19.5. The two sub-figures show the development of a software project over time (from top to bottom) when following:

- *top down refinement*, which proceeds by first developing the overall architecture for the system to meet the high level specification. Then, the specification for the main sub-systems is developed and implemented. Then the components of the sub-system are successively specified and implemented until a complete implementation is achieved; conversely
- *bottom up composition*, which addresses low level design problems first, by identifying available pre-existing frameworks and libraries that are suitable for re-use. These components are then successively orchestrated into larger sub-systems, until a complete system is realized.

The two strategies can be complementary and a mixed approach combining both strategies is often adopted for many software projects. Top down refinement is useful for relating specifications to design and implementation (ensuring

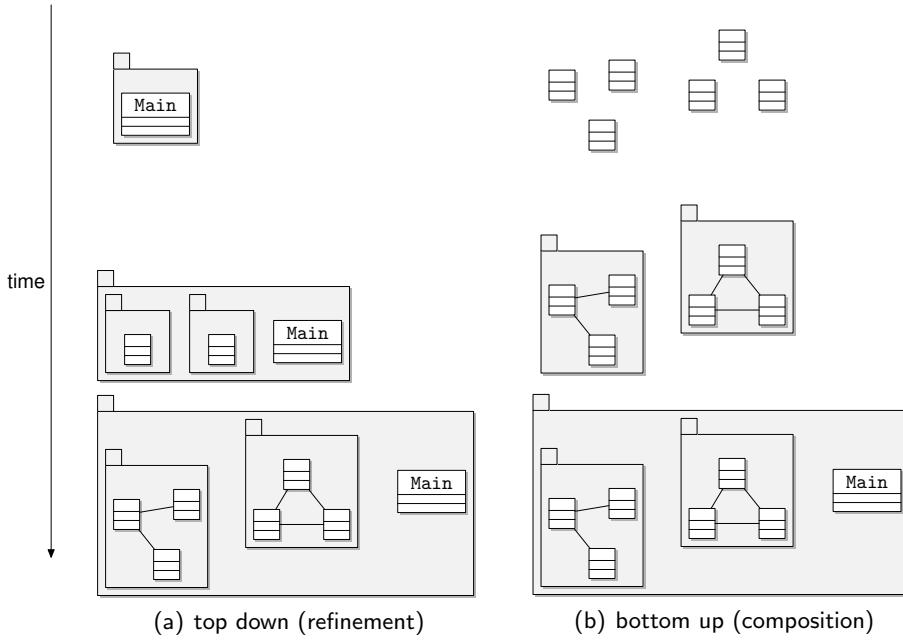


Figure 19.5: design strategies

the system meets its agreed specification). Top-down refinement is a popular strategy for formal software engineering methods. However, bottom-up composition is often better for identifying opportunities for software reuse before too much of the design is developed.

Perhaps unsurprisingly, the design strategies mirror the equivalent strategies for testing. In general the test plan and harness should be refined as a design is developed, regardless of what strategy is followed.

## 19.2 Design Principles

In this section, we will look at the design principles which underpin good design decisions, including the development and application of design patterns. In summary, these principles are:

Design principles

- maintain a separation of concerns;
- maintain sufficient abstraction;
- reuse in design and design for re-use;
- design for testing;
- **increase cohesion;** and

- **reduce coupling.**

Of these, principles, perhaps the most important are the two complementary principles: increase the cohesion and reduce the coupling amongst modules in a software system. If you understand and can apply these two principles, much of the other guidance discussed here will flow naturally. These two principles are explored in detail in Section 19.2.5 and 19.2.6.

### 19.2.1 Maintain a Separation of Concerns

A key skill for a software engineer is to partition a software solution to address different concerns. This is sometimes called a divide and conquer strategy. A large problem is gradually partitioned into smaller, separate problems, until each problem can be addressed in its own right. The benefits of maintaining a separation of concerns are:

- allows a designer to concentrate on small problems in isolation;
- makes the identification of re-usable components which match the smaller problem easier;
- allows specialist developers to work on specific aspects of a problem;
- improves flexibility, since alternative solutions to small problems can be tested in isolation;
- minimises the cost of correcting a bad design decision, since the decision should affect a specific problem only; and
- reduces cost for future maintenance efforts, since changes can be made to one part of the design without affecting another.

Figure 19.6 illustrates, using a package diagram, a software application that has been developed with this principle in mind. The diagram is of a small application for translating Subversion<sup>1</sup> logs into different formats. Subversion is an open source revision control system that can be used to track changes in many types of document. The logs are records of comments left by different users when they commit changes to a document to the system. You can find out more about version control management and Subversion in Chapter 8.

Sometimes it is useful to incorporate log information directly into another document, if you are working on a multi-author draft of a report and need to track what changes have been made during a review of the draft. The application shown in Figure 19.6 can be used to translate the logs from the Subversion native formats (text or XML) to the document format, e.g. L<sup>A</sup>T<sub>E</sub>X, RTF, Word or ODF.

---

<sup>1</sup><http://subversion.tigris.org>

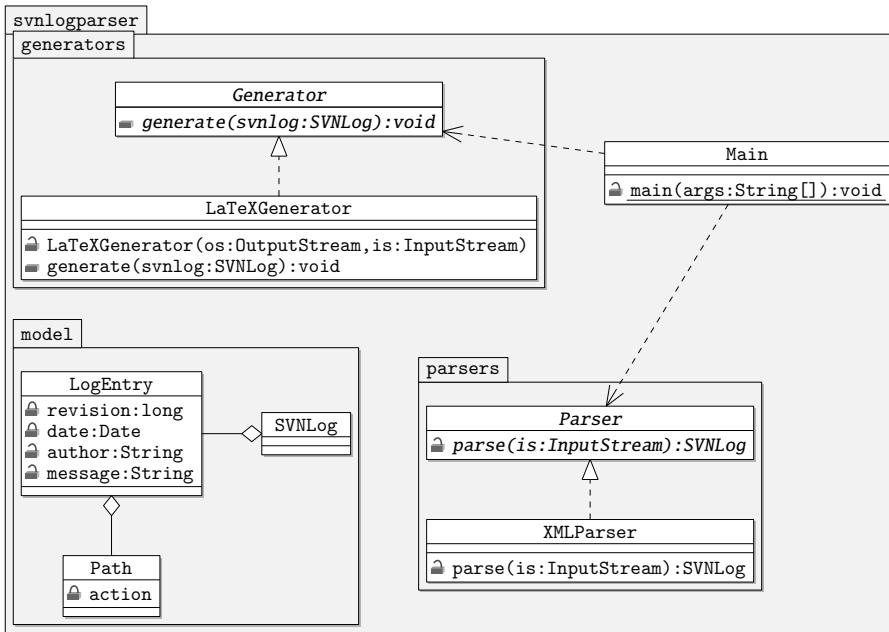


Figure 19.6: separation of concerns in an SVN log parsing framework

The application maintains a separation of concerns because different parts of the translation task are managed in separate sub-packages. The:

- **model** package stores classes which provide a programmatic model of a Subversion log, including a `SVNLog`, `LogEntry`, and `Path` class. This package provides the internal representation of Subversion logs, which can be manipulated independent of input and output formats;
- **parsers** package contains the classes used for reading input logs in different formats and translating them into the internal programmatic representation. The package contains an interface which specifies the API for a Subversion log `Parser`. The package also contains an implementation of the interface for the `XML` log format. The `Parser` implementations have no knowledge of the further use of `SVNLog` instances once they are generated;
- **generators** package contains the classes which produce output based on a programmatic model of a log. The package contains a `Generator` interface with a single method for accepting an `SVNLog` instance for output. A class is also provided which generates `LATEX` format output of log files. Complementary to the `Parser` interface, any implementation of `Generator` has no knowledge of the source of a Subversion log.

The `Main` class in the `svnlogparser` provides the front end of the application

and is the only part of the program which links an implementation of Parser with an implementation of Generator interface.

A risk of a divide and conquer approach to software development is that the emergent properties of a composed system are ignored; or that *cross-cutting concerns* such as security, auditing and input validation are addressed in different ways in different structural modules. The *aspect oriented* approach to software development is intended to address this concern by compartmentalising cross-cutting aspects so that they can be applied wherever they are needed throughout an object oriented design. AspectJ<sup>2</sup> is an implementation of aspect oriented development for Java.

### 19.2.2 Maintain Sufficient Abstraction

Maintain sufficient abstraction

Abstractions allow you to understand the essence of a subsystem without having to know unnecessary details. These more concrete details can be dealt with as a separate design problem later in the project development.

Avoid committing to concrete implementations too early, particularly if:

- parts of the concrete implementation may change; or
- several different implementations may be required.

A good example of this is the Collection framework. All the generic functionality of a collection is placed in the AbstractCollection class. Only the details of storage and retrieval (via iterator and add operations) is left abstract - for concrete classes to implement.

Another example of the use of abstract types is the Input/Output stream framework in Java. All I/O in Java is managed via streams. Input is pulled from an InputStream, while output data is pushed to a OutputStream. The particular implementation backing these streams (a network connection, a file or the console, for example) is often unimportant to the application managing them. The code below illustrates a class that writes input from an InputStream to an OutputStream.

Example – IO stream polymorphism

```
public class EchoArbitraryIO extends Thread{  
  
    private InputStream is;  
    private OutputStream os;  
  
    public EchoArbitraryIO(InputStream is, OutputStream os){  
        this.is = is;  
        this.os = os;  
    }  
  
    public void run(){  
        while(true){  
    }
```

---

<sup>2</sup><http://www.eclipse.org/aspectj>

```

try {
    int i = is.read();
    os.write(i);
    os.flush();

} catch (IOException e) {e.printStackTrace();}
}

public static void main(String[] args) throws
IOException{
new EchoArbitraryIO(System.in, System.out).start();
new EchoArbitraryIO(System.in, System.err).start();

FileOutputStream fos =
new FileOutputStream(new File("polyio.out"));
new EchoArbitraryIO(System.in, fos).start();
}

```

The EchoArbitraryIO can be passed any implementation of an `InputStream` and `OutputStream` and will work with them as normal. In the example, the first EchoArbitraryIO stream reads data from stdin and writes to stdout. The second instance is the same except it writes to stderr. The final instance writes to a file via an instance of `FileOutputStream`.

The code is also an example of a *race condition*. There is no way of guaranteeing which instance of EchoArbitraryIO will read which byte of input from stdin.

Figure 19.7 illustrates a final example of the benefits of retaining abstraction, taken from the evacuation alert system from Section 18.4.3. The figure illustrates the design of an abstract `AbstractAlertListener` class, which implements the `AlertListener` interface.

The `AbstractAlertListener` class is responsible for converting the information about an alert into a generic message to be broadcast. The constructor for the `AbstractAlertListener` is parameterised with the evacuation instructions (where to go, what to take) that should be sent out to every receiver of the message. Several different `AbstractAlertListener` instances with different instructions (say for different geographic locations with different assembly points) could be registered. The message is passed to an abstract method `sendMessage()` for forwarding via implementation specific communication mechanisms.

The three implementations of `AbstractAlertListener` provide communication mechanisms for Simple Message Service (SMS), TextToVoice (over telephone) and issuing messages on a micro blogging site. In each case, the communication mechanism is passed a message title, content and date that the message was issued.

The code for the body of the `AbstractAlertListener` class is shown below:

```
public abstract class AbstractAlertIssuer implements
```

Example – send alert

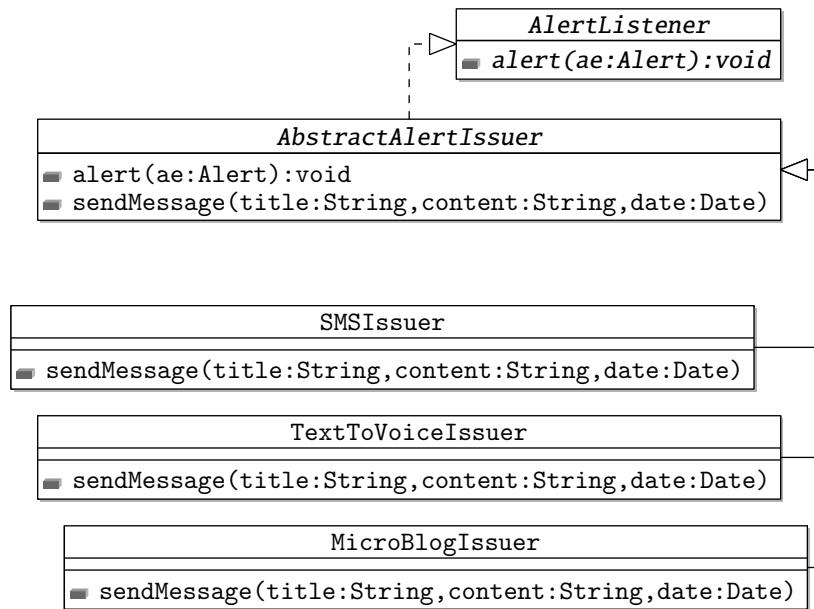


Figure 19.7: extending the evacuation alert system with different communication mechanisms

```

AlertListener {

    private String evacuationInstructions;

    public AbstractAlertIssuer(String evacuationInstructions)
    ) {
        this.evacuationInstructions = evacuationInstructions;
    }

    @Override
    public void alert(Alert a) {
        String title = "[WARNING] a " + a.getType()
            + " alert has been issued.";

        Date date = new Date();

        String message = "A " + a.getType()
            + " alert has been issued. Please read the"
            + " following instructions concerning evacuation"
            + "\n." + evacuationInstructions;

        sendMessage(title, message, date);
    }

    public abstract void sendMessage(String title, String
  
```

```
    message , Date date);  
}
```

### 19.2.3 Reuse and Reusability

Good practices for encouraging the reuse of software and the development of re-usable software are to:

- generalize your design as much as possible within the context of the problem domain. Make *hooks* and *slots* available for others to complete a component. You may find that as you develop your system, certain parts can be extracted as separate projects; and
- reuse existing designs, frameworks and libraries. It is perfectly acceptable to be lazy as a software engineer. There is no point in wasting your project budget on re-inventing solutions which have already been built better by others.

Reuse in design and design for re-use

Beware of anti-patterns when re-using code. In particular, you should avoid *cloning* code to re-use it. In addition, if you don't make any design decisions in your system, you risk end up causing the *inner-platform effect*.

We will look at frameworks in more detail in Chapter 17.

### 19.2.4 Design for Testing

As noted above, design strategies and their application in practice, are rather similar to the testing strategies described in Section 21.5.

Design for testing

A good design should:

- expose sufficient information to support testing; and
- provide exactly the same access to test harnesses as to clients.

If the cause of a failure cannot be isolated in a design, then the design should be further decomposed in the design so that sub-components can be isolated and tested separately. Figure 19.8 illustrates the re-organisation of a system in order to expose faulty functionality for testing.

Example –

re-organising a design for testing

Figure 19.8(a) illustrates an application for simulating the behaviour of a vehicle, connected to its `VehicleSimTest` test harness. The simulator works as a thread which continually calculates the velocity of a vehicle based on the current acceleration and turn angle. This velocity is then applied added to the current location to get the new location. A client class can obtain the vehicle position at any point in time by calling the `getLocation()` operation. A defect has been detected in the simulator concerning the location when the vehicle is decelerating. The defect is believed to be in the `calculateVelocity()` method. Unfortunately, the method is private so cannot be tested in isolation.

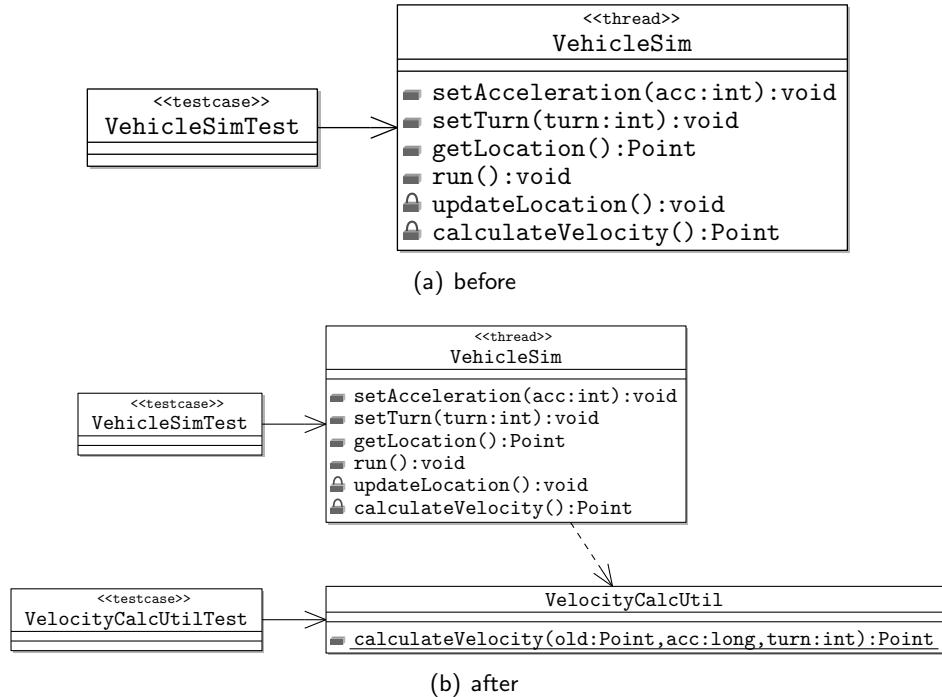


Figure 19.8: re-organising a design to support testing

Figure 19.8(b) illustrates a re-organisation of the design to support testing of the `calculateVelocity()` method. The method has been extracted and placed in a separate utility class, `VelocityCalcUtil`. The method has been made static and parameterised with the arguments necessary to calculate velocity. The new class can now be tested using with a separate test case, `VelocityCalcUtilTest`.

A test first approach to design and implementation can help to enforce the design for testing principle. A test first approach forces the designer to specify the public interface of a module before tests and a subsequent implementation can be developed. Unfortunately, a test first approach can be difficult to achieve in practice, because the designer may not know what the interface of a component should exactly be (remember, design is an iterative process).

In general, you should avoid developing test cases for reused components, which is really the job of that development project. However, test cases may need to be developed to determine that a failure is originating in a reused component and to document this in a defect report for the other project.

### 19.2.5 Increasing Cohesion

In general, cohesion refers to the extent to which related parts of a software system are kept together and separated from other parts of the system. During software development your goal should be to achieve a highly cohesive system.

<b>type</b>	<b>rubric</b>
functional	keep related, stateless functions together and everything else out
layer	write to and read from the level below only
channel	access data from a single source
behavioural	group sequentially and temporally related functions
utility	group functions used by several components

Table 19.1: types of cohesion

Highly cohesive systems are generally easier to maintain and evolve because when a new feature needs to be added, or a defect removed, all the relevant functionality should be collected in one place. There are different approaches to achieving cohesion, summarised in Table 19.1.

Types of cohesion

### Functional Cohesion

Functional cohesion means keeping related functions together *and* keeping everything else out. The design principle requires both clauses to work! Lethbridge and Laganière [2005] make an additional requirement that a function should have no side effects for the state of the system, i.e the function should be stateless.

Functional cohesion

Recall Figure 19.6 on page 393, which illustrates an example of functional cohesion, in which the application is also stateless. The Subversion log parsing application as a system does not maintain any state between each successive invocation. The application just reads in a Subversion log and produces an output in the required format. Notice that functional cohesion can be achieved by maintaining a separation of concerns during the design process.

### Layer Cohesion

Layer cohesion can be achieved by organising the functions of a software system into layers. In a layered design, lower level layers provide services to higher level layers via a well defined interface. In general, successive intermediate layers provide abstraction over the functionality implemented in lower level layers. The functions that can be accessed within each layer are collated into one or more interface, which is a façade for the layer (See Section 18.2.5). Each layer façade is realized by one or more concrete classes, which utilise the functionality of the lower level layers as necessary.

Layer cohesion

The advantage of layer cohesion is that the implementation of one layer is completely separated from its use by a higher layer. If the functionality of one layer needs to be altered, this can be done without changing how it is used by the layers above, provided the operations specified in the interface are not changed. If necessary, the implementation of a layer can be changed completely

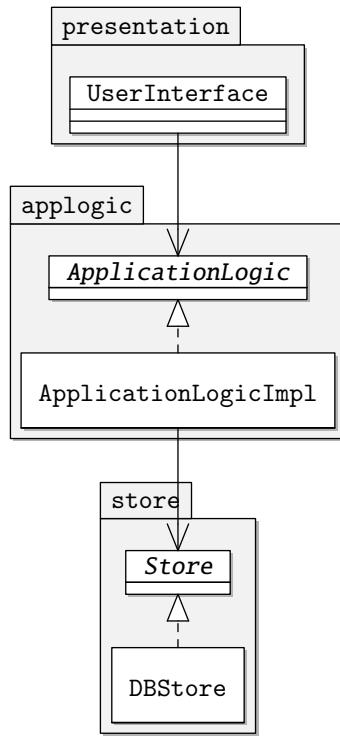


Figure 19.9: generic layered information system architecture

without affecting its use. In addition, new features can be added to the lower level layers before they are utilised in higher level layers, easing system evolution.

Figure 19.9 illustrates a layered architecture for an information system. Information systems are a general class of application designed for storing and manipulating business processes and data.<sup>3</sup> A typical information system consists of three layers:

- a data store layer, where information is persisted, typically in a database management system (DBMS);
- a logic layer which defines the rules by which the information in the store can be manipulated and the summary reports that can be requested by a user; and
- a presentation layer, which controls how data is shown to a user, in a Graphical User Interface, for example.

In the Figure, each layer of the architecture is organised into a package. A layer can interact with the layer below by invoking methods in the public

---

<sup>3</sup>There are numerous generic frameworks for developing information systems, as well as a number of specialised frameworks for developing information systems in specific problem contexts, such as stock control, university course and student management and so on.

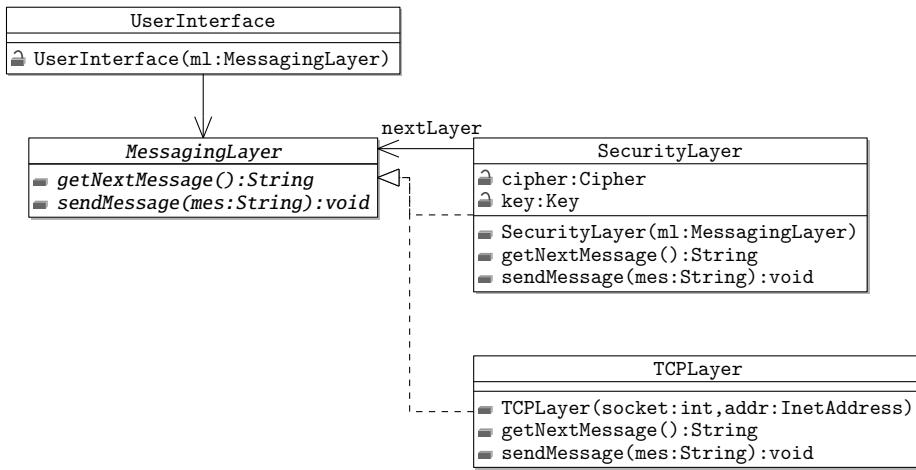


Figure 19.10: a layered design for a messaging system

interface. For example, the `applogic` package can access functionality in the `store` package by invoking operations in the `Store` interface.

Notice that the particular implementation details of a layer can be changed without affecting other layers. For example, the `Store` interface could be implemented using a `FileSystemStore`, rather than `DBStore` if desired, but this wouldn't affect the `ApplicationLogicImpl` class in the `applogic` package immediately above it.

A layered design is commonly adopted for numerous other classes software artifacts, including:

- information systems;
- operating systems;
- network protocol stacks; and
- web server platforms.

Figure 19.10 illustrates the layered design of a instant messaging application, using a class diagram. In this design, the use of layering is slightly different. A `UserInterface` instance interacts with a `MessagingLayer` instance, which can either by a `TCPLayer` for communicating messages over a network connection, or an intermediate `SecurityLayer`, which encrypts messages before forwarding them to a lower level `MessagingLayer`.

Figure 19.11 illustrates the layering of the instant messaging application for a particular configuration of the system using an instance diagram. The diagram shows a top level `UserInterface` instance, which interacts with an intermediate `SecurityLayer` instance. Finally, the `SecurityLayer` instance passes messages on to the `TCPLayer` instance for communication.

Example – instant messenger application

Example – as an instance diagram

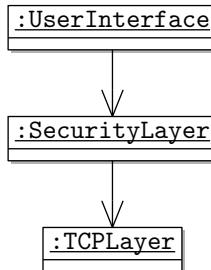


Figure 19.11: instance diagram of a layered architecture for a messaging system

Notice that in this arrangement additional layers could be added if necessary. For example, if we wish to track records of conversations between uses of the instant messaging application, we could provide another implementation of `MessagingLayer` called `LoggingLayer`, which could record messages in a file or database. An instance of this layer could be inserted into the instance diagram in Figure 19.11 between the `UserInterface` and `SecurityLayer` without affecting the existing behaviour.

Layering is recognised as an *architectural design pattern*. Architectural patterns are similar to design patterns, except that they address design problems at the level of sub-systems and components, rather than individual classes. We will return to architectural patterns in more detail in the Professional Software Development (3) course.

### Channel Cohesion

data cohesion  
(communication)

Channel cohesion is achieved by grouping together classes and methods for accessing particular types of data. The advantage of data cohesion is that if the storage mechanism for data changes, all the affected modules will be in the same place.

Figure 19.12 illustrates an example of data cohesion. The figure focuses on the `model` package from the Subversion log parsing application. The package collates all the information about a Subversion log into a single package, and can represent the data model for the application in a single class diagram.

### Behavioural Cohesion

Behavioural cohesion

Behavioural cohesion is often used during the initialization of objects, when several distinct, but related activities need to be undertaken together. Behavioural cohesion is achieved by keeping together functions that are used in the same order (sequential) or at the same time (temporal).

Sequential cohesion refers to methods that are invoked one after the other. Temporal cohesion means methods that are invoked at the same time, but not necessarily in a particular order.

The code below illustrate the initialization steps for the Swing widget illustrated in Figure 19.13.

```
public class CaptureConfigPanel extends JPanel {  
  
    /***/  
    private static final long serialVersionUID =  
        5758818488258704449L;  
  
    private static final String PATH_CHOOSE = "...";  
  
    private static final String CAPTURE_IMAGE_PATH = "Image  
        capture path (dir):";  
  
    private static String DEFAULT_GRABBER_IMAGE_PATH = "imgs  
        /";  
  
    private static final Dimension DIMENSION = new  
        Dimension(400,35);  
  
    private JLabel pathL;  
    private JTextField pathF;  
    private JButton chooseB;  
  
    public CaptureConfigPanel(){  
        super();  
        initialisePanel();  
        initialiseListeners();  
        initialiseConfiguration();  
    }  
}
```

Example – CaptureConfigPanel

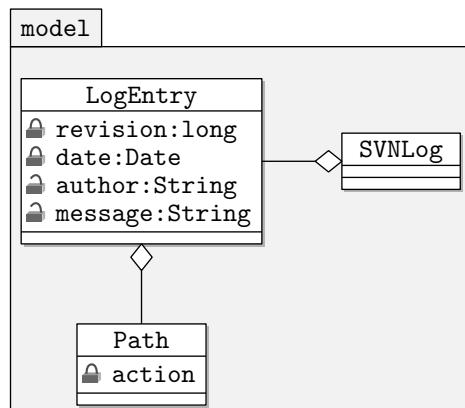


Figure 19.12: data cohesion



Figure 19.13: the CaptureConfigPanel widget

```
private void initialisePanel() {
    setPreferredSize(DIMENSION);

    pathL = new JLabel(CAPTURE_IMAGE_PATH);
    pathF = new JTextField();
    chooseB = new JButton(PATH_CHOOSE);

    SpringLayout sl = new SpringLayout();
    setLayout(sl);

    sl.putConstraint(SpringLayout.NORTH, chooseB, 0,
                    SpringLayout.NORTH, this);
    sl.putConstraint(SpringLayout.EAST, chooseB, 0,
                    SpringLayout.EAST, this);

    sl.putConstraint(SpringLayout.NORTH, pathL, 0,
                    SpringLayout.NORTH, chooseB);
    sl.putConstraint(SpringLayout.NORTH, pathF, 0,
                    SpringLayout.NORTH, chooseB);

    sl.putConstraint(SpringLayout.SOUTH, pathL, 0,
                    SpringLayout.SOUTH, chooseB);
    sl.putConstraint(SpringLayout.SOUTH, pathF, 0,
                    SpringLayout.SOUTH, chooseB);

    sl.putConstraint(SpringLayout.WEST, pathL, 0,
                    SpringLayout.WEST, this);

    sl.putConstraint(SpringLayout.WEST, pathF, 0,
                    SpringLayout.EAST, pathL);
    sl.putConstraint(SpringLayout.EAST, pathF, 0,
                    SpringLayout.WEST, chooseB);

    add(pathL);
    add(pathF);
    add(chooseB);
}

private void initialiseListeners() {
    chooseB.addActionListener(new SelectImagePathListener());
}
```

```

}

private void initialiseConfiguration(){
    pathF.setText(DEFAULT_GRABBER_IMAGE_PATH);
}

private void setCaptureImagePathFromChooser(){
    JFileChooser jfc = new JFileChooser();

    jfc.removeChoosableFileFilter(jfc.
        getAcceptAllFileFilter());

    jfc.setCurrentDirectory(new File(pathF.getText()));

    jfc.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY)
        ;

    jfc.showOpenDialog(null);
}

File f = jfc.getSelectedFile();

if (f != null&&f.isDirectory()) pathF.setText(f.getPath())
    ;
}

class SelectImagePathListener implements ActionListener
{
    @Override
    public void actionPerformed(ActionEvent arg0) {
        setCaptureImagePathFromChooser();
    }
}
//...
}

```

## Utility Cohesion

Utility cohesion is the weakest form of cohesion, and is applied when other potential options for improving cohesion have already been adopted. Some modules provide service or utility features to numerous other modules in a system. Cohesion of these modules can be improved by

grouping modules together that are re-used in several different places and don't have a 'natural' home'.

Utility functions are typically declared static in Java classes, since they shouldn't require access to the internal state of the objects that access them.

Notice that utility functions can still benefit from functional cohesion by grouping together related utility functions into classes. Utility cohesion is also

Utility cohesion

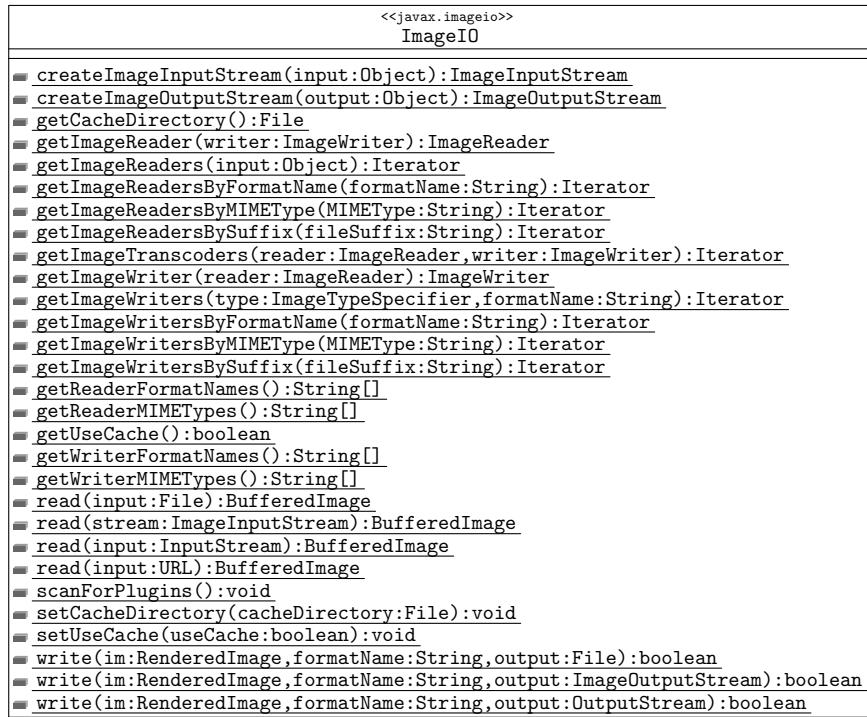


Figure 19.14: the ImageIO utility class - large, but cohesive

useful when a private utility function of a class needs to be tested in isolation. By placing the function in a separate class, make it static and public, we can adhere to the principle of designing the system so that it can be tested, without violating the object oriented principle of encapsulation.

Figure 19.14 illustrates the class diagram for the ImageIO class from the Java SDK. The class collates a large number of static methods for manipulating image sources from within a Java application. The class represents utility cohesion because the methods are used by many other Java applications for convenient image processing.

### 19.2.6 Reduce Coupling

The structure of a software system is typically denoted by the key sub-systems and relationships between them. Some relationships between structural components are necessary in order for the system to do something useful. If the components were completely autonomous, then there wouldn't be much point composing them into a system!

However, each relationship introduced between two components creates a new *dependency* between them, such that a change in one component may require a change the another. The more relationships that are added to the system, the more complex it becomes and the more difficult it becomes to either

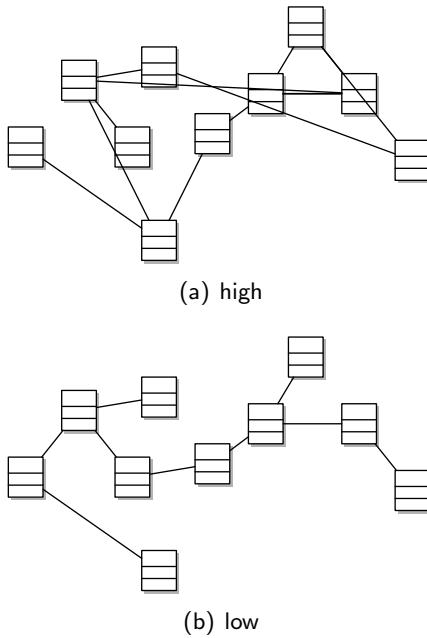


Figure 19.15: reducing coupling in a software design

change the implementation of a component, or replace it completely, without introducing unexpected changes elsewhere in the system.

*Reducing coupling* means that dependencies between components are minimised and strictly specified. Figure 19.15 illustrates a the same system with high and low coupling. In Figure 19.15(a), the high coupling between classes makes system comprehension and evolution difficult, since any change may directly affect many other classes. In Figure 19.15(b), the relationships between components is clearer, and it is easier to establish where cohesion can be improved by grouping together components with greater inter-dependencies. So, reducing coupling is complementary to increasing cohesion. Doing one is quite likely to do the other as well.

There are several different types of coupling that can be identified in a software system, as summarised in Table 19.2.

Reducing coupling

Types of coupling

### Incomplete Delegation

Lethbridge and Laganière [2005] refer to this as content coupling, although I think it is easier to remember as an incomplete application of the delegation design pattern.

The problem occurs when one object accesses (or even worse) modifies the contents of another object without the explicit involvement of the target object's class. Figure 19.16 illustrates an example of the problem. The ScoreBoard class de-references from the Match, Side, Team and Club classes to access the name

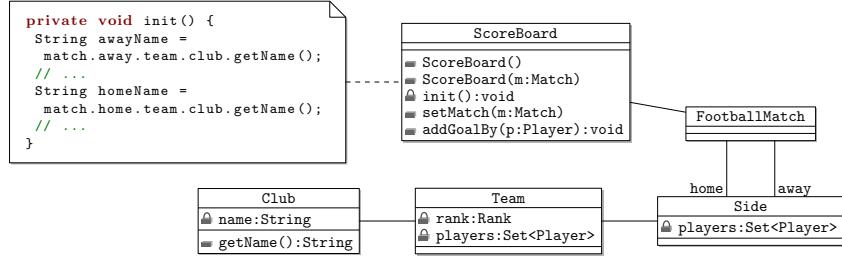


Figure 19.16: incomplete delegation in an abstraction-inheritance pattern

Incomplete delegation  
(content coupling)  
Better

of a Side. This couples the implementation of the Team and Club classes to the ScoreBoard class. If Team or Club change, the ScoreBoard class may also have to be changed.

The remedy is to complete the delegation design pattern, as shown in Figure 19.17. The ScoreBoard class must now explicitly access the FootballMatch class's references to Side using the getAwaySide() and getHomeSide() methods.

Of course, some content coupling is necessary, since not all classes can delegate their functionality to others, otherwise, every class would be trying to

type	rubric
content	the delegation from one module to another is not masked
common data	data used by two modules is stored in a common share
control	the behaviour of a function is controlled by flags
stamp, argument and type	modules become coupled to the operations used
routine	several operations must be invoked together
external/import	the module depends directly on external functionality

Table 19.2: the different type of coupling

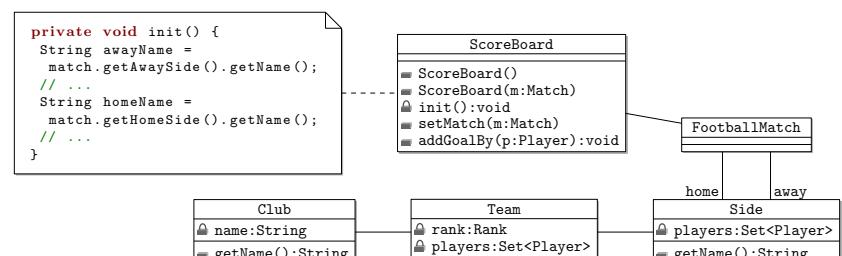


Figure 19.17: decoupling Club from ScoreBoard

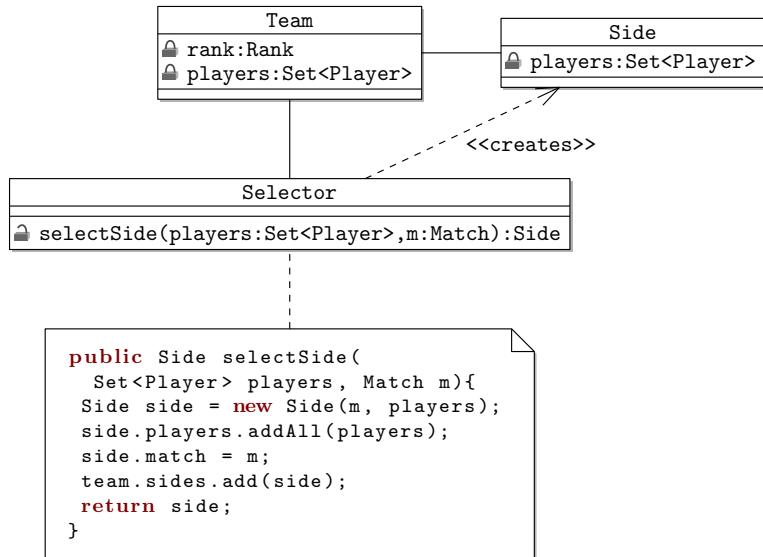


Figure 19.18: content coupling when selecting a side

do everything. Content coupling should be reduced where full delegation can sensibly be applied, but particularly where data is being modified by the client.

Figure 19.18 illustrates another example of incomplete delegation in the sportster system, in which an object of one class directly alters the state of another. The class diagram is taken from the part of the system used to select team members for a side in a match.

The arrangement is problematic because the Selector class adds the Side instance to the set of sides, `sides:Side` in the Team instance. However, the `Selector.select()` method does not validate the set of players allocated to each side. A player may be already committed to another match that day for example. This *business logic* is naturally the responsibility of the Team class.

Figure 19.19 illustrates the re-organisation of the side selection feature to complete the delegation pattern. The `selectSide()` method has been moved into the Team class. The method contains the validation checks that must be performed before the proposed set of Player instances are allocated to a Side for a particular match. If the set of players is inappropriate, the allocation can be partially or wholly rejected by the Team class.

Another example

Better

## Variable Sharing

Variable sharing, or common coupling, occurs when:

- global attributes are shared between components; and
- the value of an attribute in one object is accessed and then retained by another object (caching).

Variable sharing  
(common coupling)

A consequence of variable sharing is that all the client modules will need to be altered if the definition of the global variable changes.

Figure 19.20 illustrates an example of variable sharing in the sportster application. The public, static League.NAME attribute is a global variable shared by the Division, DivisionSeason and Season classes. As a consequence, all of these classes are coupled to the League.NAME attribute, and therefore to each other. If the definition of the attribute changes, the use of the attribute in the other classes will also have to change.

An alternative solution here would be to apply the Singleton design pattern (see Section 18.3.1) to the League class, and encapsulate the NAME attribute as an instance attribute (renamed name, following convention), with the appropriate accessor and mutator methods.

Some globally accessible attributes and operations are necessary, the root logger for an application using log4j, for example, or the `main()` method entry point for Java applications. However, wherever in many cases it is possible to en-

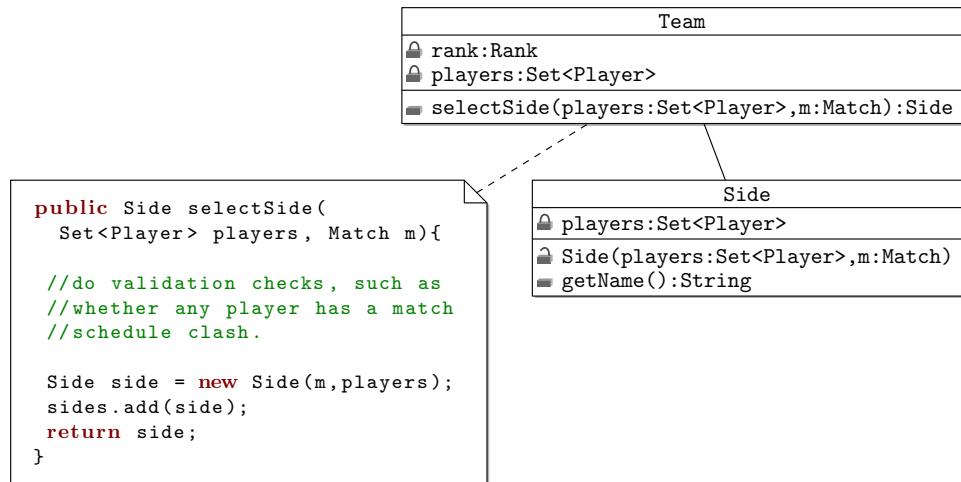


Figure 19.19: placing responsibility for side selection in the Team class

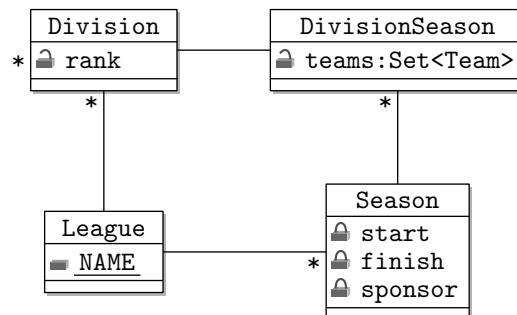


Figure 19.20: common coupling in the sportster application

encapsulate the globally accessible values inside singleton (or multi-ton) instances of their class.

*Caching* is a weak form of common coupling that explicitly acknowledges the need for variable coupling between two objects. In caching, one object (the store) maintains the 'true' value for an attribute. Another object that needs the value of the attribute records a copy of the attribute in a local store (cache). The cache may be periodically refreshed when the calling object suspects that the data value may have changed (sometimes this is done at the instigation of the user).

Caching should be used when:

Caching

- the cost (e.g. computational or network delay) of repeatedly accessing a data value outweighs the risk that the value has changed;
- a calculation must be made using a single value for an attribute, i.e. the value of the attribute cannot be allowed to change between accesses; or
- the use of a notification framework to report that an attribute value has changed is not appropriate.

Note that the decision to cache a data value should be made explicit in the design.

Figure 19.21 illustrates an example of a situation where caching is appropriate for the mobile forensics application, introduced in Figure 13.26 on page 250. The forensics application contains a wrapper class `DeviceDriver`, which is the point of interaction between the forensics application and a mobile device. Different sub-classes of `DeviceDriver` provide the hardware specific implementation for recovering memory images from different models of mobile device. All of these sub-classes will implement the `reAcquire()` abstract method.

Caching – example

Memory images are represented by the `Image` class, which contains the raw data of the memory image as an array of bytes, and a date stamp for when the memory image was acquired. Once acquired, the memory image can be used in numerous forensics applications. For example, the `Image` instances can be passed to a:

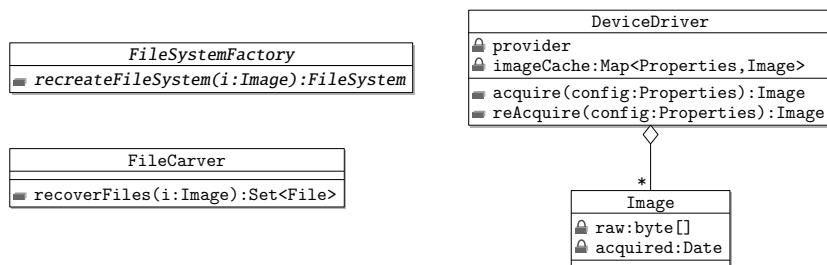


Figure 19.21: caching in the mobile forensics application

- `FileSystemFactory` factory, which can recover the file systems present in the memory image; or
- `FileCarver` file carving application, which can be used to recover file data from the image based on known file signatures. This can be useful when the file system has been corrupted.

Memory acquisition is time consuming, so acquired images are cached by the device driver. A map, `imageCache` is maintained of acquisition configurations to memory images. When the `acquire` method is called, if the image mapped by a particular configuration exists in the map, it is returned. Otherwise, the device driver performs a full acquisition and the result placed in the cache for future use. If the `reAcquire()` method is used, the image in the cache (if any) is ignored and the full acquisition is performed.

## Control Coupling

Control coupling occurs when the effect of a method call is controlled by its parameters. A good indication of control coupling is the use of *flag* parameters in method calls, which signal particular types of behaviour. Control coupling is bad for a design, because it couples the processing of parameters by a method to the way the method is called - the caller and called methods become coupled together. Consequently, if the behaviour of the called method changes (by adding new behaviour and flags, say), then both the called method and also all the client calling methods will have to be changed.

The Java code below shows part of the implementation of a `BiPlaneBomber` class.

Control coupling –  
`BiPlaneBomber`

```
public class BiPlaneBomber {

    private Point2D.Double loc = new Point2D.Double();

    public void move(Direction direction, float dist){
        switch (direction){
            case UP: loc.setLocation(loc.x, loc.y+dist); break;
            case DOWN: loc.setLocation(loc.x, loc.y-dist); break;
            case SLOW: loc.setLocation(loc.x+dist, loc.y); break;
            case ACCELERATE: loc.setLocation(loc.x-dist, loc.y);
                break;
            case BOMB: dropBomb(); break;
            case GUN: fireGun(); break;
        }
    }

    private void fireGun() {/*...*/}
    private void dropBomb() {/*...*/}
}
```

# TODO - screen shot.

Figure 19.22: the bi-plane bomber video game – very retro

The class is part of the implementation of a 2D platform game, of the sort popular in the 1980s. The idea is to fly a WWI era bi-plane across a 2D scrolling landscape on the computer screen (from left to right), shooting down enemy bi-planes as they move from right to left, and bombing enemy gun emplacements and other fortifications. Figure 19.22 gives you the idea.

Biplane bomber

The control of the bi-plane is managed by the `move()` method. When the player issues a move (typically using a key press) the `move()` method is invoked with one of the six possible `Direction` flags. The design is coupled because the behaviour of the method is dictated by its `direction` flag parameter. If a new command is added, both the user interface and the `BiPlaneBomber` class will have to be changed.

In addition, the method has been extended at some point to accept commands for firing the bi-plane's machine gun, and dropping bombs. This is poor design because:

- the `Direction` enumeration now has a mis-leading name; and
- because in some circumstances, the second `dist` parameter is not used. This is an example of argument coupling (see Section 19.2.6).

An alternative approach in this case is to use a *command lookup table*. Figure 19.23 illustrates the design of the lookup table for the bi-plane control commands. The `BiPlaneBomber` class maintains a Map of `Command` enumeration commands over an interface called `BomberOp`. Notice that `Direction` has been renamed `Command` to give a more appropriate label for its role.

Better – using a look

A `BomberOp` has a single operation, `invoke()`, which takes the target `BiPlaneBomber` instance as an argument. The interface is realized by three classes, `Move`, `DropBomb` and `Fire`. `BomberOp` operations can be registered with the `BiPlaneBomber` instance using the `registerOp()` method.

table

The code below shows how the lookup table is used within the `issueCommand()` method.

```
public enum Command {UP, DOWN, SLOW, ACCELERATE, BOMB, GUN}

public class BiPlaneBomber {

    private Map<Command, BomberOp> operations;
```

```

public BiPlaneBomber(){
    operations = new HashMap<Command , BomberOp>();
}

public void registerOp(Command c , BomberOp op){
    operations.put(c,op);
}

public void issueCommand(Command c)
    throws BomberOperationException {
    BomberOp bo = operations.get(c);
    if (bo != null) bo.invoke(this);
}
}

```

When the `issueCommand()` method is invoked on the `BiPlaneBomber()`, the `c:Command` parameter is used to get the operation registered for that command from the map. If the operation has been registered, it is invoked for the `BiPlaneBomber` instance.

Notice that there is only a single `Move` class for moving the bi-plane. However, several `Move` instances can be registered with the `BiPlaneBomber` instance with different initialisation arguments that configure the operation's behaviour. This means that the operation's state is separated from the state of the bi-plane. We could also use the read-only interface design pattern (Section 18.2.7) to limit visibility of the bi-plane's state to instances of `BiPlaneOp`.

The new design means that the state of the `BiPlaneBomber` has been separated from the operations that control its behaviour. New commands can be created and registered with the `BiPlaneBomber` without altering the class. Other than easing evolution of the game, the lookup table provides extra flexibility for

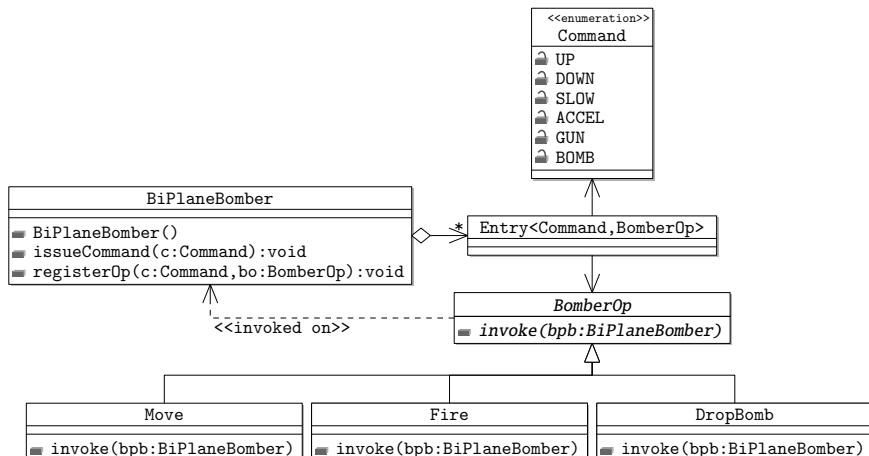


Figure 19.23: using a lookup table to de-couple commands, operations and state

the game's designers. Different instances of BiPlaneBomber can be allocated different combinations of commands.

The use of an interface as a single operation specification is the closest we can get in Java to *function pointers*, or *anonymous functions* which are available in other languages such as C++ or PHP.

## Stamp and Argument Coupling

A not too shocking statement is that it is often necessary to specify parameters for methods, otherwise they wouldn't be able to do very much! However, the number and type of parameters can lead to greater coupling than necessary between a method and the modules that invoke it. In particular:

- *argument coupling* occurs when an operation has a large number of optional primitive arguments; and
- *stamp coupling* occurs when a data type is used as a parameter to an operation.

There can be a conflict between reducing stamp and argument coupling. The Java code below illustrates an example of argument coupling:

```
public void sendMailing(
    boolean isOrganisation,
    String salutation,
    String companyName,
    String firstLine,
    String secondLine,
    String postalTown,
    String postCode){

    if (isOrganisation)
        salutation = "Dear CEO of "+companyName;

    if (secondLine == null){
        String address =
            firstLine+"\n"+postalTown+"\n"+postCode;
    } else {
        String address =
            firstLine+"\n"+secondLine+"\n"+postalTown+"\n"+
            postCode;
    }
    //...
}
```

The code is taken from an application for sending prize draw mailings by post. The `sendMailing()` method is coupled because the method signature contains a large number of primitive parameters. Not all of the parameters are

Stamp and argument coupling

Example: argument coupling

Example: Stamp coupling

used in each invocation of the method. The `companyName` parameter is only used if the `isOrganisation` parameter is `true`, and the `salutation` parameter is only used if the parameter is false. If the `sendMailing()` method signature changes (e.g. because another parameter is added), then all the users of the method will also have to change.

The code below is taken from the same application and illustrates an example of stamp coupling.

```
public class PrizeDrawMailer {

    private String signature = "Mr. Chan Ser (Chief Executive, Super Prize Draws)";

    public void sendMailing(Person p){
        String message =
            p.getTitle() + " " + p.getForenames().get(0) + " " + p.
                getSurname() +
            p.getAddressFirstLine() + "\n" +
            p.getSecondAddressLine() + "\n" +
            p.getPostalTown() + "\n" +
            p.getPostCode() + "\n\n" +
            DateFormat.getInstance().format(new Date()) + "\n\n" +
            "Dear " + p.getForenames().get(0) + ",\nCongratulations " +
            ", you've have been entered into a prize draw for " +
            calculatePotentialPrize(p, null) +
            ". There is a GUARANTEED prize for EVERY entry. " +
            "To claim your prize please phone (09077 432123).*" +
            "\n\nBest wishes and good luck from the team at " +
            "super prize draws!\n\n\t" + signature + "\n\n\n*Calls" +
            " cost Â£5 per minute. \nAll prizes must be
            collected " +
            "in person from the Antarctic. \nPrizes are subject "
            +
            "to availability. \nNo refund for prizes will be
            provided.";

        printMailing(message);
    }

    public void sendMailing(Organisation o){
        //...repeat implementation for organisations.
    }
}
```

The `sendMailing()` method is coupled because of the `p:Person` parameter in the method signature. The method does need to access some of the features of a `Person` instance in order to print the prize draw mailing. However, there are numerous other attributes of the `Person` class that are not needed by the `sendMailing()` method. The method is unnecessarily coupled to all these extra

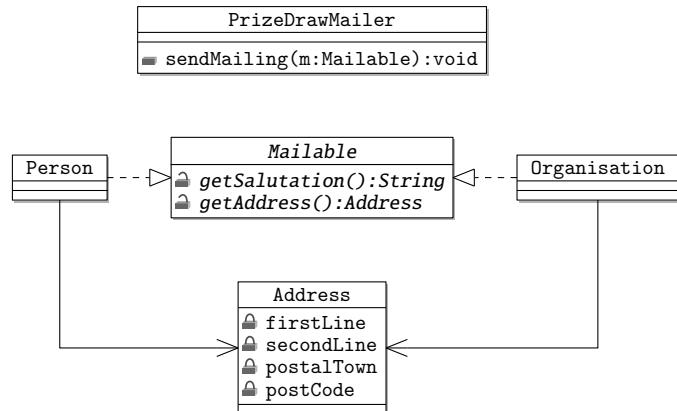


Figure 19.24: using a *Mailable* interface to abstract mailing information

features.

Figure 19.24 illustrates the use of an *extracted* interface in order to reduce both stamp and argument coupling in the *sendMailing()* method. The *Mailable* interface contains only the operations needed by the *PrizeDrawMailer* to generate prize draw mailings. The *Person* class, and a new *Organisation* class both realize this interface. In addition, the address properties of the *Mailable* class are collected together in a new *Address* class. Consequently, the *PrizeDrawMailer* is now only coupled to the *Mailable* interface.

Better – using a  
Mailable interface

### Routine coupling

Most method bodies contain more than one statement in sequence in order to implement the methods behaviour. If the methods must be called in a particular sequence, the methods are coupled together. A developer must remember to get the ordering of calls (and allocation of arguments) correct each time the sequence is required. Routine coupling is particular apparent where the same sequence of methods appears more than once in an application (this is often a result of cloning).

The Java code shown below illustrates an example of routine coupling in an eBook reader application.

Example: Routine  
coupling

```

public class EBook {

    public ArrayList<Chapter> chapters;
    // ...

    public void printBook(Printer printer) {
        Graphics2D g = (Graphics2D) printer.getGraphics();
        ArrayList<Page> pages = getAllPages();
        for (Page page : pages) {
            drawBorder(page, g);
        }
    }
}

```

```

        writeText(page, g);
        printer.sendPage();
    }
}

public void renderPage(Graphics g, int i) {
    Page page = getAllPages().get(i);
    drawBorder(page, g);
    writeText(page, g);
    writePageNumber(i, g);
}
// ...

```

The `drawBorder()` and `writeText()` methods are used in the same sequence in the `renderPage()` and `printBook()` methods. The coupling can be reduced by encapsulating these two calls in a new method which is called atomically.

## Import and external dependency coupling

Almost all classes depend on functionality implemented elsewhere. This is necessary in order to separate concerns within an application. However, this still results in two, usually weak forms of coupling:

- import coupling occurs when one module imports the functionality provided by another; and
- external coupling occurs when an application depends on the configuration of the environment.

Import coupling can be minimised by pruning unused classes from the list of imports (in a Java application). External dependencies should be isolated into well defined modules. A layered design is a good way of achieving this, by wrapping the interactions with external hardware in the lower levels of the application's layers.

## 19.3 Documenting and Evaluating Designs

As noted in Section 19.1, software design is an iterative process because it can be hard to get the arrangement of modules right first time. The evaluation phase of the design process is used to determine how well a design meets its functional and quality requirements.

## Measuring designs

Software can be measured:

- statically (LoC, dependencies, complexity); and
- dynamically (profiling, performance).

There are numerous tools available for measuring software quality, for example:

<http://clarkware.com/software/JDepend.html>

[www.eclipse.org/tptp/](http://www.eclipse.org/tptp/)

During the early iterations of the software design process it is often appropriate to adopt an informal approach to documentation based on white-board sketches and hand-written notes. This is acceptable, because the design is likely to change, so producing extensive documentation that quickly becomes obsolete is inefficient.

However, as a software application's design becomes more stable, the quality and coverage and detail of the accompanying documentation should also increase. Software design documentation should describe the:

Documenting designs

- problem addressed, describing the feature to be provided;
- related documentation, including a reference to the application's requirements specification;
- considerations and priorities, describing the philosophy adopted in addressing the problem, (e.g. what metrics were considered high priority for a good solution?);
- description, of the adopted solution;
- dependencies, on external libraries, frameworks, components or hardware;
- rationale, justifying the design solution; and
- consequences, including any disadvantages for the application of the adopted solution.

## Summary

Software design is an iterative process, in which a trade-off is made between features, quality and budget in order to select an acceptable design solution to a problem. The quality of software designs can be improved by following established design principles. Avoid over documentation of software design until the design has stabilised for an application.

## 19.4 Exercises

1. Consider the source code for a class shown below:

```
class Piece {}

class Player {}

class Rook extends Piece {}

class Pawn extends Piece {}

public class ChessGame extends JFrame{

    public ArrayList<Piece> pieces = new ArrayList<
        Piece>();

    public ArrayList<JPanel> squares = new ArrayList<
        JPanel>();

    /** moves a piece from the source to the target
     * square, if the move is valid */
    public void move (
        int x1,
        int y1,
        int x2,
        int y2, Piece move,
        Piece captive, Player player) throws Exception{

        //check if move is valid...
        if (move instanceof Pawn){
            if (!(y2 - y1 == 1 || y1 - y2 == 1)) throw new
                Exception();
            } // ...
        }

    public void init (){
        //setup the GUI
        //TODO
        //setup the game logic
    }
}
```

Identify examples of coupling in the code. For each example, describe:

- (a) the form of coupling exhibited;
- (b) why it is detrimental to the system design; and
- (c) how you would change the source code to correct the problem.

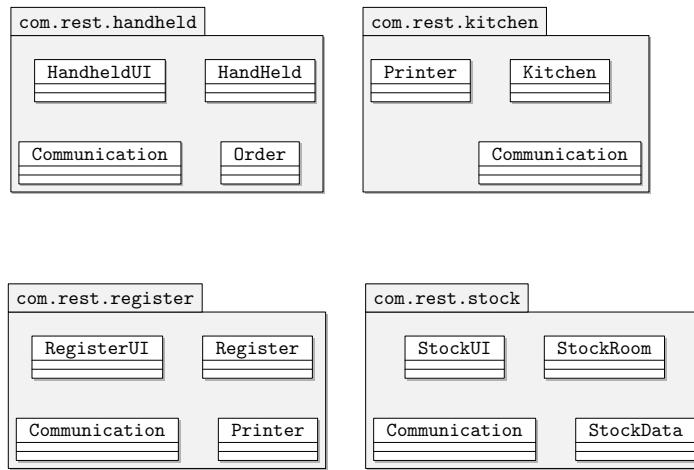


Figure 19.25: food order system

2. Review the design of the Glasgow Calculator Framework
  - (a) Identify examples of good practice in the design and explain how they contribute to high cohesion and low coupling.
  - (b) How could the design be further improved? What would be the consequence of the changes you propose?
3. During a corporate take-over, your company acquires a software system for managing food orders in restaurants. Waiting staff place orders on hand-held devices. The orders are then transmitted wirelessly to a sub-system in the kitchen, which sends the orders to a printer. When a meal is finished, the hand held application can send the bill to an electronic cash register, where a paper copy for the customer is printed. Another sub-system tracks stock levels in the restaurant and send alerts to the hand held device if particular food items have run out. The system is structured into packages as shown in Figure.

 A work colleague proposes to re-organise the system in order to improve the cohesion of the system. She proposes to:
  - (a) move all user interface classes into a new `rest.ui` package;
  - (b) create a new `rest.comms` package for storing common communication functionality;
  - (c) create a new `rest.util` package, containing a `Print` class to handle printing functions; and
  - (d) create a new `model` package for storing classes that describe data, including the `Order` class.

Give your response to each of these suggestions and explain your reasoning.

4. A country's tax office is implementing a new system for calculating income tax payments from individuals.

The calculation of income tax for a year is made by (1) deducting an allowance from an income (2) calculating a proportion of the income as tax, based on the current rate of taxation and (3) finally reducing the payable tax by a social allowance. The current possible allowances are for energy bills or child care costs. At most one form of credit is available for each tax payer.

Consider the Java source code for part of the tax management system shown below.

```
public class Employee {  
    private double income;  
    private String pAYENumber;  
    private Address workAddress;  
    public double getIncome(){return income;}  
    public String getPAYENumber(){return pAYENumber  
        ;}  
    public Address getWorkAddress(){return  
        workAddress;}  
}  
  
public class TaxCalculator {  
  
    public double calculateTax(  
        Employee payer,  
        float rate,  
        double allowance,  
        double energyCredit,  
        double childCredit  
    ) throws Exception {  
        if ((energyCredit > 0.0) && (childCredit >  
            0.0))  
            throw new Exception(  
                "Only one credit is permitted");  
  
        double income = payer.getIncome();  
        double taxableIncome = income-allowance;  
        double tax = taxableIncome*rate;  
        tax = tax - energyCredit - childCredit;  
  
        return tax;  
    }  
}
```

Name and describe two forms of coupling in the current design of the system. In each case briefly propose an alteration to the system design to reduce coupling. You *do not* need to provide code for your solution.



## **Chapter 20**

# **Software Architecture**

### **Recommended Reading**

### **Summary**

## **20.1 Exercises**

# Chapter 21

## Software Testing

### Recommended Reading

PLACE HOLDER FOR lethbridge05object

### 21.1 Aims of Testing and Basic Concepts

Software testing has three key aims:

Aims of testing

- defect detection is concerned with demonstrating the presence of defects in a software system. Software tests are developed that exercise systems in unusual ways. A successful defect test causes the system to fail. Defect testing is performed throughout development;
- failure documentation is concerned with reporting the occurrence of a failure in the form of a test case that exhibits the failure; and
- acceptance demonstration is concerned with demonstrating that a system satisfies its requirements specification. A successful acceptance test means that a system can be handed over to a customer or users. Acceptance tests may be developed by the customer (or another contractor) to demonstrate that the system is satisfactory.

Note that neither a successful acceptance test or an unsuccessful defect test guarantees that a system is defect free. In either case, defects unexercised during testing may remain latent in the software system. Instead, testing is useful for gauging the quality of the software under test. The more testing that is performed, the more defects should be detected (and hopefully removed during debugging). Unfortunately, testing cannot be exhaustive because:

- the budget for testing is finite; and

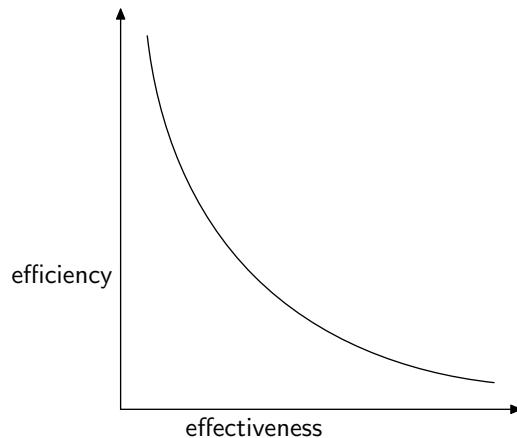


Figure 21.1: effectiveness vs. efficiency in testing

- the more defects that are detected and removed, the more expensive it becomes to detect and remove future (and presumably harder to find) defects.

Figure 21.1 illustrates this dilemma. As the effectiveness of testing increases (measured by the total number of defects left undetected in the system, efficiency declines dramatically because the cost per defect increases.

Software testing can be defined in terms of the interfaces that a system or component under test exposes to its environment. Figure 21.2 illustrates the relationship between *interfaces*, *failures*, *defects* and *slips*:

- interfaces are contracts describing the expected service of a system in terms of:
  - inputs,
  - outputs, and
  - non-functional characteristics.

Interfaces may be specified in numerous ways and with varying degrees of rigour. Additionally, different observers of a system may have different understandings of the interface provided by a system;

- failures are observations of a system implementation's deviation from *expected* behaviour. Identification of a failure is subjective, because one observer may judge an event at an interface to be a failure when another does not;
- defects (or bugs, faults) are the adjudged or deduced causes of failure in a system, an incorrect variable assignment, or an invalid pointer reference, for example; and

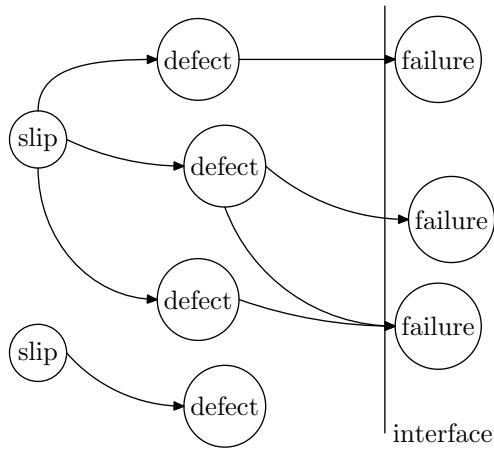


Figure 21.2: slip, defect, failure model

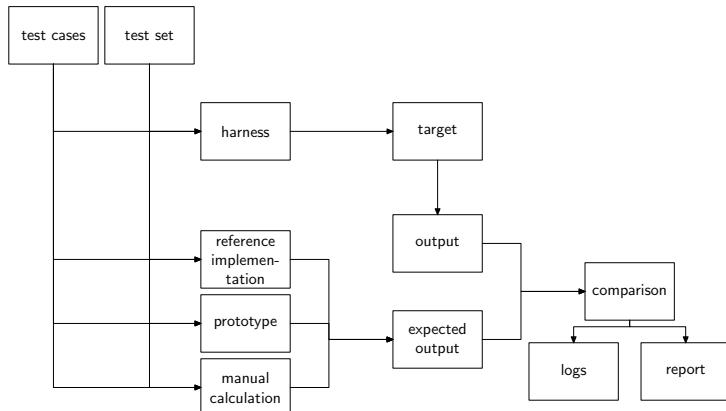


Figure 21.3: the software testing process

- slips(or errors) are the actions by programmers that introduce defects into a software system.

So testing is the process of exercising a system or component at its interface with the intention of causing the system to exhibit a failure. Notice that since the observation of failures is subjective, so is the presence or absence of defects in a system.

A really nice story that illustrates the subjective nature of failure concerns software used to simulate the explosive yield of certain types of bomb associated with characteristically vegetable shaped clouds (which I'll talk about in class).

Figure 21.3 formalises the stages of the testing process. The elements of the testing process are:

The testing process  
Testing terminology

- targets are the programs or applications under test;

- test cases are descriptions of sequence of actions taken to perform a single test on a system. The test case describes the expected behaviour to be exhibited by the system, often as relation between inputs and outputs;
- test sets are the collection of input data and expected outputs to be used in the implementation of a test case. The same test case can be invoked with many different test data values taken from a test data set;
- test harnesses are software programs used to invoke test cases on target applications. Many frameworks exist to support the implementation of test harnesses;
- test oracles are sources of expected outputs for given inputs for comparison with the outputs obtained from test targets; and
- test report is the description of the result of executing a test suite against an application.

Another way of thinking about testing as a form of scientific experiment in which you wish to deliberately invalidate your hypothesis. Hypothesis: system is correct, then conduct experiments in search for contrary evidence.

To prove or disprove the testing ‘hypothesis’ it is necessary to compare the outputs from the target program with expected values. The source of these values is called a *test oracle* because in principle they should always provide the correct expected output for a given input. There are several different types of test oracle:

- reference implementations are previous implementations of the interface under test;
- prototype are early versions of the software system or component under test; and
- manual calculation requires the software tester to deduce the expected output themselves.

In practice, obtaining a ‘true’ software test oracle can be difficult, not least because if there is already a perfect version of a software component available, you should probably be using it! Instead, many different ‘partial’ or ‘pseudo’ oracles may be used to produce the expected results for the test. When outputs are compared, the presence of differences between expected and actual results are reported, even though the defect may be in either the target, oracle, or both.

For example, a reference implementation may be used to test the functional characteristics of a target system. The new system may be under development to improve the non-functional characteristics such as performance.

There are two categories of approaches to developing tests in the search for defects.

## Test oracles

- {black, opaque} boxtesting has no access to the internal implementation details of the target under test. The only information provided is the specification of the system interface. As a consequence, the tester cannot make assumptions about the manner in which the implementation has been completed and must specify tests based on the stated range of inputs. A risk of bblack box testing is that the test cases will miss defects in parts of the system that are exercised infrequently; and
- {white, clear, glass} box testing exploits knowledge of the internal implementation details of a system in order to develop tests that cover as much of the system's behaviour as possible. A disadvantage of this approach is that the tester may be susceptable to the same assumptions as the original developers as to the use of the system.

Sometimes black box testing is necessary if the details of system implementation are unavailable, for commerical reasons, or because the source code has been lost. Black box testing may be used to reverse engineer the functionality of a system. Glass box is more usually called white box testing. However, L&L have a point that glass box is a more accurate analogy. By extension though, Black box testing should be called Opaque container testing!

## 21.2 Black box Testing

Black box testing is undertaken at a component's interface. Since, the tester has no knowledge of internal implementation details, tests are constructed from the specification of what the system *should* do.

Black box testing

There may be different types of system specifications used as a basis for black box testing including:

- formal specifications stating pre and post conditions and invariants;
- syntactic formal specification with informal semantic documentation; or
- informal description only.

Regardless of the type of specification available, black box tests are specified to check for:

- invalid parameters or ordering;
- out of range arguments to parameters;
- extreme (legal) arguments; and
- invalid sequences of operations.

The code below shows an example JavaDoc specification of an operation.

```

/**
 * Converts a String of binary symbols (0,1)* in two's
 * complement format, into the equivalent integer value.
 *
 * @param binString
 *         the string of binary symbols (0,1)*
 * @param isTwoComplement
 *         specifies whether the string of binary
 *         symbols
 *         is to be treated as a two's complement
 *         signed
 *         integer or as an absolute value
 * @returns a decimal representation of the binary input
 *
 * @throws NumberFormatException
 *         if the input string contains any
 *         character
 *         other than 0 or 1
 */
public int binaryToDecimal(String binString,
    boolean isTwoComplement)

```

## Equivalence partitions

In principle, it would be desirable to test all the possible combinations of inputs to an operation. Unfortunately, testing an operation exhaustively is very expensive, and thus likely to be highly inefficient. For example, there are:

- $2^{32}$  **int** values
- $2^{16}$  **char** values
- $(2^{16})^{2^{31}} = (2^{16})^{2^{147483648}} = 2^{34359738368}$  **String** values

Fortunately, for any given operation, many input values and combinations to an operation will be treated in the same manner. These equivalent values, or equivalent combinations of values are called *equivalence classes* or *partitions*. For each equivalence class, it is often sufficient to test:

- at and near the class boundary (if ordered); and
- randomly within the class.

There is an article in the risks digest that illustrates the inefficiencies of exhaustive testing very nicely<sup>1</sup>.

Establishing equivalence classes requires a tester to infer how different subsets of input values and their combinations will be treated by the implementation. Consider the example specification shown below. The code specifies an operation for calculating the tax on an individual's income.

## Example Equivalence partitions: calculating income tax

---

<sup>1</sup><http://catless.ncl.ac.uk/Risks/24.42.html> referencing <http://horningtales.blogspot.com/2006/09/exhaustive-testing.html>

```
public int calculateTax(double income);
```

The income tax in this particular territory is progressive, meaning that the rate of income taxes increases for larger portions of income.

- income is not taxed up to £10000.00 inclusive;
- income between £10000.01 and £24999.55 inclusive is taxed at 20%; and
- income over £24999.55 is taxed at 40%.

These partitions provide the natural equivalence classes for the `calculateTax` (`int`) operation. Notice that the specification doesn't state how negative income inputs should be handled. Presumably an exception should be thrown and a tester should include testing for this in their test suite and a comment concerning the omission in the test report.

Consider the next example specification shown below for an operation to determine whether an integer value represents a leap year.

```
public boolean isLeapYear(int year);
```

Example Equivalence partitions: leap years

The rules for determining a leap year are as follows:

- all years divisible by 4 are leap years; except
- all years divisible by 100 are not leap years; except
- all years divisible by 400 are leap years.

Again, the rules provide a convenient basis for specifying equivalence classes: For input `year`, we can infer that the operation should partition the inputs into the following categories:

- `year % 400 == 0` is leap year, e.g. 400, 800, 1600
- `year % 100 == 0 && year % 400 != 0` is not leap year, e.g 1900, 2000
- `year % 4 == 0 && year % 100 != 0` is leap year, e.g. 48, 1988
- `year % 4 != 0` is not leap year, e.g. 1967

When operations have more than one parameter, the input equivalence partitions of the individual parameters need to be combined to produce the *operation equivalence classes*. The combinations can then be categorised as either legal or illegal. Test cases can be developed for each operation equivalence class.

Establishing the dependencies between parameters and the resulting operation partition can be complex. The easiest way to start is by combining the parameters that should be independent of one another, but may have effects on other parameters in combination.

Consider the example operation shown below for converting temperature values between different metrics.

Combining and prioritising equivalence partitions

conversion	equivalence class boundaries
Kelvin → Celcius	$-\infty, 0.0, 273.15, \infty$
Celcius → Kelvin	$-\infty, -273.15, 0.0, \infty$
Kelvin → Fahrenheit	$-\infty, 0.0, 255.37, \infty$
Fahrenheit → Kelvin	$-\infty, -459.67, 0.0, \infty$
Fahrenheit → Celcius	$-\infty, -459.67, 0.0, 32.0, \infty$
Celcius → Fahrenheit	$-\infty, -273.15, -17.7, 0, \infty$

Table 21.1: equivalence class boundaries for temperature conversion

```
public abstract Double convertTemperature(
    Metric from, Metric to, Double temperature);
```

Table 21.1 illustrates the partition of the input combinations. The `from` and `to` parameters are independent (i.e. the value of any other parameter shouldn't affect how they are interpreted). The table shows how each possible combination of these two inputs affects the treatment of the final `temperature` parameter. The table shows the equivalence partition boundaries for each of the combinations.

Just as for single parameters, the combination of parameter equivalence classes can give a large number of operation equivalence classes (this is actually a product of the number of equivalence classes in each parameter). Test cases should be developed that prioritise high use and high risk operation equivalence classes.

Some defects only manifest themselves after the system has reached a particular *state* (remember that state is defined by the attributes of the objects in a system and the relationships between them). Some examples of failures that often take time to manifest themselves are:

- primitive operation arity, where the result of a calculation depends on the order of evaluation of primitive operations. Sometimes, incorrect ordering may not be apparent because for particular inputs for arity is not relevant. For example, in the calculation  $a + b/c$ , the result will be the same regardless of whether the addition or division operation is evaluated first, if  $a$ ,  $b$  and  $c$  are all equal to 1.;
- floating point precision problems particularly where division operations are evaluated earlier than necessary;
- out of memory defects are caused when manually managed memory is exhausted. A *memory leak* is caused when unneeded memory is not released by an application. Automatic memory management in languages like Java mitigate this problem, but don't eliminate it.
- shared memory defects occur when two independent threads or processes concurrently read and write shared memory inappropriately; and

## Sequencing operations

- timing defects occur when one operation is invoked before the surrounding environment is ready.

Consider the example expression shown below - what is the value of  $x$ ?<sup>2</sup>

```
int x = (1+2/3*4-5) %6+3/2+1;
```

Consequently, testing must also anticipate that a system will exhibit a failure only after a number of (possibly many) operations have been invoked. Testing for the absence of sequence defects is challenging and again, requires that testers identify high priority test cases.

## 21.3 White box Testing

White box testing addresses many of the limitations of black box testing. By exploiting knowledge of the underlying implementation of a system, tests can be constructed that exercise a quantifiable amount of an application's source code. Since the implementation of the algorithm is known when the test case is constructed, white box testing serves two particular purposes:

White box testing

- documentation of defects identified during an inspection; and
- prevention of defect introduction during system evolution.

The steps in constructing a test case for an algorithm (e.g. implemented as a method) are as follows:

1. identify the steps, conditions and transitions of the algorithm;
2. map flows through algorithms as a flow chart or activity diagram;
3. label each transition of the algorithm;
4. express the desired test case as a sequence of transitions through the algorithm;
5. determine the inputs to the algorithm that will result in the paths being executed; and
6. estimate code coverage of test suite in terms of *paths*, *edges* or *nodes*.

If no input can be identified to cause a particular transition to be traversed then the application contains *unreachable* code. Compilers for modern programming languages are able to check for some types of unreachable code.

The code shown below is an implementation of the `BinaryToDecimal` interface specified above.

---

<sup>2</sup>-2 is my guess.

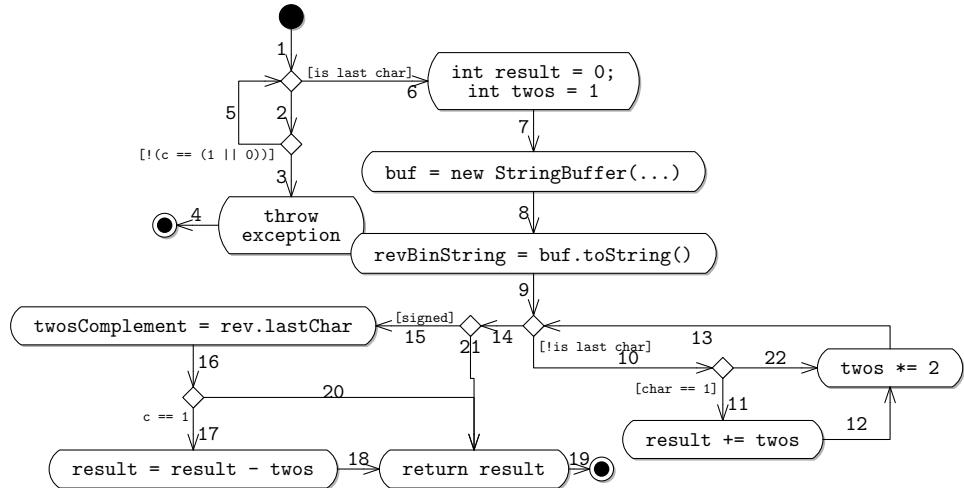


Figure 21.4: an implementation of the `BinaryToDecimal` operation

```

for (char c: binString.toCharArray())
    if (c != '0' && c != '1')
        throw new NumberFormatException(
            "Only binary symbols [0,1] permitted");
    int result = 0;
    int twos = 1;
    StringBuffer buf =
        new StringBuffer(binString).reverse();
    String revBinString = buf.toString();
    for (char c: revBinString.toCharArray()){
        if (c == '1')
            result += twos;
            twos *= 2;
        }
        if (isTwosComplement){
            //apply two's complement rule
            char twosComplement =
                revBinString.charAt(revBinString.length()-1);

            if (twosComplement == '1') result = result - twos;
        }
    return result;
}

```

## Activity diagrams

Figure 21.4 illustrates the steps of the program translated into a UML *activity diagram*. Each executable statement and control structure (**if**, **for**, **while** etc.) is treated as an atomic activity. Transitions between activities is drawn using arrows. The control structure conditions are also shown on the diagram.

Notice that I haven't shown the exact syntax for reasons of brevity on the diagram. Also, the statements themselves may represent the invocation of quite complex algorithms. If necessary, atomic activities can be *refined* into more

detailed sub-diagrams themselves. In general, however, you should not be concerned with developing test cases for components or libraries provided by other software developers, as they should be responsible for testing the correctness of their code. You may, of course, need to develop a black box test case that documents a defect you find in another developer's work.

The diagram is used to identify the different ways of transitioning through the algorithm implementation. Each transition arrow is labelled with a unique number. These numbers are then used to specify *paths* through the algorithm which represent the desired route of the test cases. The necessary inputs to achieve these paths can then be identified. For example:

Specifying test cases

- 1,2,3,4 (e.g. "a")
- 1,2,5,2,5,3,4 (e.g. "0a");
- 1,2,5,2,5,6,7,8,9,10,11,12,13,10,11,12,13,14,21 (??);
- ...

There are several levels of rigour to adopt for white box testing, expressed in the coverage of different aspects of an algorithm's activity chart. Covering all:

- nodes ensures that all statements are executed and that all conditions are (at least partially) evaluated. However, some of the transitions between statements may not have been exercised, so not all of the possible input contexts may have been tested. For example, the first test case proposed above exercises transitions 1,2,3,4, (and the intermediate nodes). However, the 'true' edge from the second condition is not exercised, so the condition has not been fully tested;
- edges ensures that all transitions between states have been exercised. However, this does not guarantee that all combinations of transitions have been exercised, so some inputs to a node may be untested;
- paths ensures that all possible routes through a program are exercised. Covering all paths is approximately the same rigour as testing all operational equivalence partitions. Unfortunately, the set of paths through an algorithm is often very large, particularly if the program contains a loop. Testing all paths is usually impractical.

Estimating coverage:  
nodes, paths and  
edges

As with black box testing, ensuring high coverage of system functionality during test case development can be very challenging. Two techniques to consider in developing white box tests are:

- automatic test case generation using frameworks like JTest which inspect programs are generate test cases automatically; and

- automated measurement of coverage using tools which measure what proportion of a code base was exercised by a test suite.

## 21.4 Categorising Defects

Categorising defects

Depending on what is known about a component or system, testing strategies can be focused on different categories of defect. There are several different ways of categorising defects, for example:

by specification:

- functional
- non-functional

or by occurrence:

- deterministic
- non-deterministic

or by error [Lethbridge and Laganière, 2005]:

- control
- numerical
- timing and coordination (see tutorial 23.6)
- external dependencies
- documentation

We will look at the different errors that can cause defects in the this section.

Control condition  
defects

### 21.4.1 Control Defects

**Condition** defects occur when the condition controlling which part of a control structure is executed is incorrectly formulated. Consider the following specification for a power supply unit automatic shut off.

The power supply must be off if the emergency stop has been pressed, or if the load exceeds 90% of capacity, except if the normal capacity limits are overridden, in which case, the load may not exceed 99% of capacity.

The code shown below is a possible implementation of the specification, however the complexity of the specification makes validating the correctness of the conditions and the consequent actions difficult.

```

if (powered
    && (capacity - load) / capacity > .9
    || (override &&
        (capacity - load) / capacity > .99 || stopPressed))
throw new PowerControlException();

```

An appropriate testing strategy for control defects is to determine and exercise all combinations of input equivalence classes in the conditions.

**Control construct scope** defects occur when the statement to be executed within a control structure is outside its scope, or in the wrong control block. The code below shows a scope defect in a function for performing a number of shuffles on a list of input values.

```

public List<String> controlDefect(
    List<String> input, int seed){

    List<String> results = new ArrayList<String>();
    results.addAll(input);

    Random r = new Random(seed);
    int shuffles = r.nextInt();

    for (int i = 0; i < shuffles; i++){
        Collections.shuffle(results);

    return results;
}

```

Control construct  
scope defects

The code is defective because of the extra ; semi-colon at the end of the **for** loop header, which pushes the `Collections.shuffle()` invocation outside of the control of the **for** loop. Consequently, the method will only shuffle the list of values once, regardless of the value of `shuffles`. Indeed, the collection will still be shuffled once, even in the values of `shuffles` is 0.

A difficulty with testing this code is the use of a randomly generated variable. Where possible, the seed used to invoke the random operations should be made explicit, in this case by calling the `shuffle` method with an explicit seed. This at least allows the result of a shuffle to be held constant for testing.

**Infinite loop** defects occur when the control condition has the potential to never be satisfied. Some compilers can statically check for simple occurrences of unreachable source code statements, however, infinite loop defects are often more subtle.

The example below shows the code for a control system for filling a bath with water.

```

private double water = 0.0;

```

Infinite loops defects

```

private double capacity = 100.0;

public void fillTheBath(double rate){
    while (water != capacity){
        water += rate;
    }
}

public void pullThePlug(){
    water = 0.0;
}

```

The intention of the code is fill the bath until it reaches capacity (set to 100.0 by default). However, if the rate value is not a factor of 100, the condition will not be satisfied, since the comparison checks that the water level is *exactly* the same as the capacity.

### Off-by-one defects

**Off-by-one defects** occur when a statement is executed one two few or one too many times. The code below is a simple implementation of an algorithm for calculating the factorial of a number.

```

public int factorial(int i){

    int result = 1;
    int j = 1;
    while (j < i){
        result *= j;
        j++;
    }
    return result;
}

```

Unfortunately, the loop executes one too few times, add the value of *i* to the multiplication.

### Pre-condition defects

**Pre-condition** defects occur when the state of an application is not properly checked before the operation is started. The code below shows an operation for controlling the speed of a car. The car's speed should only be increased up to the maximum speed, however, the condition at the start of the method doesn't check whether speed could increase beyond the maximum speed during the invocation of the method.

```

public class CarControl {

    private double max_speed;
    private double speed;

    public void accelerate(double rate, int duration){
        if (speed == max_speed) return;
}

```

```

    else speed += rate*duration;
}
}

```

Pre-condition defects are problematic either because they allow an operation to proceed when it shouldn't or because the operation is prevented from proceeding on legal input. When documented, the inputs that should cause an operation to throw an exception should be explicitly tested for.

Checking all the necessary pre-conditions can often be quite difficult, particularly if a application is *highly coupled* (see Section 19.2.6 for more details on reducing coupling), since an algorithm may be dependent on the state of many distantly connected variables from the surrounding application.

**Null and (non)-singleton** defects are parts of a program that mis-handle the unusual situations where an algorithm must manage none, one or many 'things'. This class of defects often occur in recursive algorithms when the base case for the recursion is identified by a single or empty set of values. For example, the binary search algorithm recursion should terminate when there is only one candidate left to be tested (whether it is the required key or not).

Look again at the `BinaryToDecimal` method listing above on page 436. How will the application handle the following inputs?

```

"1"
"0"
"
```

Null and  
(Non)-Singleton  
defects

The application will cope with the first two singleton situations. However, the application will not cope with the null condition when the `binString` parameter is empty. One of two things will happen, depending on whether the `twosComplement` parameter is `true` or `false`. If the parameter is `false`, then the method will return 0, since loop will not be executed, but the result accumulator is initialised. On the other hand, if the parameter is `true`, then the method will throw an `ArrayIndexOutOfBoundsException` when the method attempts to obtain the last character of the empty string.

Note that a null-member defect is distinct from a null reference defect (i.e. the cause of a `NullPointerException`).

#### 21.4.2 Numerical defects

**Precedence** defects can be present in expressions containing infix numerical operations, where it is necessary to choose which operation to evaluate first. Precedence rules specify which operation to evaluate first, and parentheses () are used to alter precedence for sub-expressions. However, it is possible to get precedence wrong, particularly for complex expressions.

Precedence defects

```

int z = 0;
int i = 1;
```

```

int y = (1+2/(z*4-5))%6+32/i;
System.out.println(y);

```

Precedence defects are hard to detect if some variables in the calculation are usually 0 or 1. As a consequence it is important to select unusual values for input parameters during black box testing.

**Memory overflow** defects are manifested when the result of a numerical calculation is larger than the space allocated to store it in a variable. As a consequence, the stored value will be modulo the size of the variable.

#### Memory overflow defects

Consider two examples shown below:

```

int x = Integer.MAX_VALUE+1;
int y = Integer.MAX_VALUE*Integer.MAX_VALUE;
//etc.

```

The value of x will be -2147483648, rather than 2147483648 and the value of y will be 1 rather than 4611686014132420609.

Memory overflow defects can be tested for by testing how an operation handles extreme values.

**Precision** defects are often manifested when there is a small difference between the expected outcome of a calculation and the result produced, often as a result of early rounding during the evaluation of an expression causing a loss of precision for that calculation. Note that there is a distinction between *precision* and *accuracy* in numerical calculations. Accuracy refers to how close a calculation is to the expected result. For floating point calculations, precision refers to how many decimal places can be reported.

#### Precision defects

Precision defects are often handled automatically by modern compilers which are able to re-order numerical operations to avoid loss of precision in the compiled version of a program. Consider the following example fragment from a JUnit test case:

```

Double op1 = 22.0;
Double op2 = 7.0;
Double val = op1/op2;

val = val +1;
val = val -1;
val = val*op2;

assertEquals(22.0, val,0.0);

```

Symbolically, the test should pass, since:

$$val = (op1/op2 + 1 - 1) * op2 = 22/7 + 1 - 1 * 7 = 22 * 7/7 = 22$$

However, the following output is reported by JUnit:

```
java.lang.AssertionError: expected:<22.0> but was  
: <21.99999999999996>
```

The difference is caused by the necessary rounding of the result placed into `val` in the first line. The intermediate addition and subtraction are necessary to make the example work, else the compiler will automatically re-order the sequence of operations (or even remove redundant operations) to eliminate the rounding.

Precision defects can be detected effectively by making the required precision of a calculation explicit in a test case.

#### 21.4.3 Documentation and External dependency defects

**Documentation** defects are manifested when an documentation is:

- absent because the documentation has not yet been provided;
- incorrect (e.g. out of date) a common occurrence where large amounts of documentation is managed manually, when the software requirements are subject to constant change, or when the project is behind schedule; or
- inconsistent i.e. the documentation conflicts with a description of another part of the system, or two descriptions of the same part of the system cannot be reconciled.

Documentation and External dependency defects

Documentation defects can be manifested in the:

- specification because the description of what the system *should* do does not meet the customer's real needs;
- implementation because how the system functions is different to the stated design; or
- dependency descriptions because the set of system dependencies evolves independently of the system itself.

Specification defects can be discovered as a result of black box testing, when the test report indicates that a test case has been incorrectly implemented as a result of an incorrect specification. Other types of documentation defect are often easier to discover during inspections, if documentation is absent for example.

The *agile* practice of *constant refactoring* encourages developers to add or improve documentation during frequent reviews.

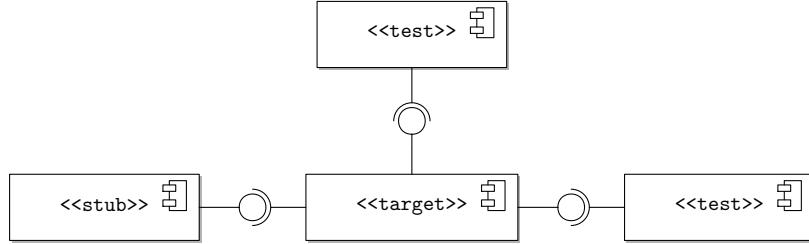


Figure 21.5: unit testing and stubs

## 21.5 Testing Strategies and Integration

We've now covered two approaches to developing tests for a target system. We now need to consider how the collection of test cases for different parts of the system is managed. Some definitions:

[More definitions](#)

- a test suite is a collection of test case implementations, which use data selected from a test set to test a system;
- a test integration strategy describes the strategy adopted for sequencing tests in a test suite and incrementally integrating the components of the system for testing; and
- a test plan is a document that combines, the test cases, strategy and a description of how test cases are realised as tests in a suite.

Unit testing is the smallest scale of testing that is undertaken on a system. A unit test is considered to be atomic, that is, no separate testing is undertaken on the sub-components of the part of the system under test. For object oriented programs, unit testing is typically performed on single classes (preferably represented as an interface).

Figure 21.5 illustrates the unit testing arrangement in an object-oriented program using a UML *component diagram*. Components are run-time entities in an object oriented system, performing a well defined purpose. Typically, a component is a collection of a number of long lived objects that exist for much of an object oriented program's life-cycle, and manages a number of shorter lived objects. Components are denoted as a labelled rectangles, with an identifier. Components are also often denoted with the 'plug' logo in their upper right hand corner.

Unit testing and Stubs

The target component exports two interfaces which describe discrete aspects of the component's behaviour. The exported interfaces are drawn as 'lollipops' attached to the component. The interfaces can be labelled with their name in order to identify them. These two components are tested separately using test components developed in a test harness. If a component is *dependent* on

an interface exported by another component, the dependency is denoted by a connection from the dependent to the interface.

The target component also requires functionality provided by another component. However, it is desirable to isolate unit testing where possible, so a *stub* which provides a ‘dummy’ version of the functionality required. Stub components are often implemented to return the same value, regardless of what input arguments were supplied to an operation. Once testing is complete, the stub can be replaced by a component implementation.

A test and integration strategy explains how the collection of individual tests from the test cases will be executed on the target system; and how the components of the system will be integrated during testing. There are numerous possible integration strategies, depending on the architecture of the system. Some guidelines for developing a testing strategy are:

#### Integration testing

- components with no internal application dependencies can be subjected to unit testing on their provided interfaces first. These are often at the boundary or bottom of a system architecture;
- optional components can be added later in the integration process;
- components that mainly provide presentational capability for data provided by other components can be tested using stubs.

Some common strategies are:

- top down begin with the user interface supported by a number of underlying stubs, and gradually add sub-systems and individual components;
- bottom up isolatable low level components are tested first by developing unit tests. These components are gradually integrated and tested at successively higher levels of abstraction until the user interface is added;
- middle out core functional components are tested first by providing stubs to external resource dependencies;
- outside in components at the system boundary are tested first by connecting them to their external dependencies and developing unit tests for their internal interfaces. Examples include modules that interact with system resources such as databases and network connections; and
- a mixture of all of the above is often the most commonly recognisable strategy, since different individual strategies will be adopted for different parts of the application as appropriate.

Figure 21.6 illustrates the components of a system to be subjected to integration testing. The system is a document management service that is interacted

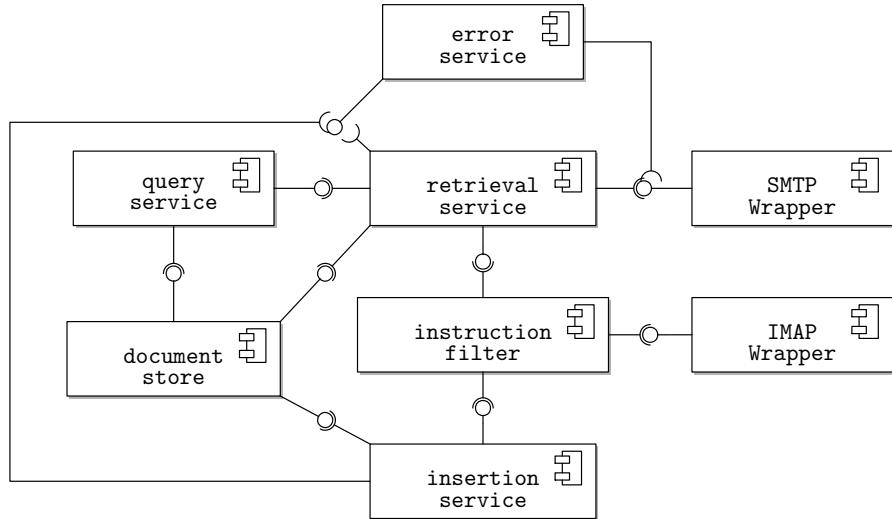


Figure 21.6: example integration sequence

### Example integration

with via an email client. Users can send emails to the service with attached documents for storage. The users can specify keywords to be associated with the document when it is stored by placing these in the email subject line. The user can also retrieve documents by sending emails with the command 'RETRIEVE' in the subject line followed by a logical expression over keywords.

A suitable testing strategy for the system might be:

1. document store (*inside out*). The component has no dependencies so can be test in isolation at its interfaces for storing and retrieving documents, and for getting documents that match a keyword
2. SMTP wrapper and IMAP wrapper (*outside in*) these two components can be connected to their external dependencies (an SMTP transport and IMAP retrieval server, respectively) and tested at their internal interfaces for sending responses and receiving command emails.
3. integrate insertion and retrieval with document store (*bottom up*). This will necessitate providing a stub for the query service, SMTP wrapper, and error service. However, now that the document store and email wrappers have been tested, this minimises the number of untested dependencies used at this stage.
4. integrate instruction filter (*bottom up*) with insertion and retrieval service, and provide a unit test that is also a stub for the filter to the IMAP wrapper, so that the test can simulate the retrieval of commands.
5. integrate the tested email wrappers and instruction management components (*outside in*). This means that testing must now occur by sending

emails to the external IMAP email server and observing the response to the SMTP server.

6. replace the error service and query service stubs with the real implementations, completing the system.

Several alternative variants to this strategy are possible. When specifying your own strategy, be careful to document your rationale for it.

Testing is part of a larger quality assurance process for a software development project. The purpose of testing is to detect and report defects manifested as failures. Figure 21.7 illustrates the role of testing within the test-fix cycle as an activity diagram.

When a failure is reported (and documented as a test case), the report is passed to a developer responsible for defect removal. The reporting of a defect may occur via a *bug tracking* repository specifically setup for the project. The developer then inspects the documentation and source code and hypothesises a cause of the failure (i.e. the nature of the defect). A correction is then implemented in the code and the software re-tested using the test case which manifested the failure. If the test passes, then the defect is considered to have been removed from the software. Otherwise, a cycle of hypothesis revisions and corrections occurs until the defect is removed.

During the final stages of a software release, a software application may be subject to release testing. During this process, test cases are developed internally to demonstrate that new features added to the system have been completed and that defects in the previous release have been removed. Following this *alpha* testing, software may be released to a community of users for further *beta* testing. This provides an opportunity for the software to be exercised by a more diverse range of users, with less pre-conceptions concerning the expected behaviour of the software. Beta testing can result in a number of further defects being detected by user reports. Software developers then need to replicate and formalise failure reports as test cases in the test suite for later resolution.

#### The Test-Fix Cycle

#### Testing and software evolution

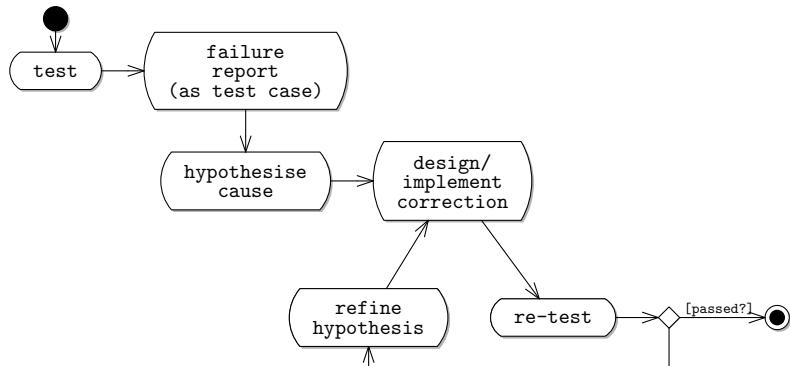


Figure 21.7: the test-fix cycle

The addition of new features, or the removal of defects from a system may also cause new defects, or cause previously corrected defects to be re-introduced. This is known as the *ripple effect* of software evolution. To manage the potential for defects to be introduced during development, it is necessary to maintain a suite of *regression* tests. These are tests previously developed to demonstrate the presence of a defect. Running regular regression tests, e.g. during project integration allows a team of software developers to monitor the quality of their software product.

## 21.6 Managing Test Cases with JUnit

JUnit<sup>3</sup> is a test harness framework for Java. The JUnit framework is well supported by the Eclipse IDE, both for creating test cases and running them. This section provides a short introduction to developing tests in JUnit, based on the BinaryToDecimal interface and corresponding implementation given on pages 432 and 436 respectively. Exercise 0 provides a more detailed tutorial on developing tests in JUnit.

The first step is to set up the test case. The code below shows the first steps for creating a test class (all JUnit tests are implemented as classes).

```
package uk.ac.glasgow.oose2.documentation.tests;

import static org.junit.Assert.*;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;
public class BinaryToDecimalUtilTest {
```

Populate the test with methods

The test class can be called anything, but good practice is to name the test <ClassUnderTest>Test. The test case should be placed into a sub-package of the class under test called tests. Test actions are specified by annotating methods so the appropriate annotations, @Test, @Before and @After should be imported. In addition assertions concerning outputs from the target are made using the static methods of the Assert class, so this should also be imported.

Test actions are implemented in methods as shown below.

```
public class BinaryToDecimalUtilTest {

    @Before
    public void setUp() throws Exception {
    }

    @After
```

---

<sup>3</sup><http://www.junit.org/>

```

public void tearDown() throws Exception {
}

@Test
public void testBinaryToDecimalString() {
    fail("Not yet implemented");
}

@Test
public void testBinaryToDecimalStringBoolean() {
    fail("Not yet implemented");
}
}

```

Any method annotated `@Before` will be invoked before each method annotated `@Test`. By convention, a method called `setUp()` is annotated with the `@Before` annotation. Similarly, a method annotated `@After` (by convention called `tearDown()`) will be called after each `@Test` annotated method.

The use of `@Before` and `@After` allow the construction of a fresh test *fixture* for each and every test action in the class. The fixture is the interface through which the test will interact with the class under test. The code below shows the provision of a fixture for the test class.

```

private BinaryToDecimalUtil b2dUtil;

@Before
public void setUp() throws Exception {
    b2dUtil = new BinaryToDecimalImpl();
}

@After
public void tearDown() throws Exception {
    b2dUtil = null;
}

```

Creating and  
removing fixtures

Notice that the fixture variable has an interface type, rather than a class. This means that the test methods can only interact with the target via the fixture interface. Also notice that the `tearDown()` method explicitly sets the `b2dUtil` attribute to null after each test method is called. This ensures that the actions that occur under one test do not affect subsequent tests.

The code below shows how to replace default `@Test` method implementations with assertions.

Creating tests

```

/*
 * defects: off-by-one in loop, wrong handling
 * of 2s complement.
 */

@Test
public void testBinaryToDecimalStringPositiveEven() {
    assertEquals(

```

```

        "Positive even conversion",
        4,
        b2dUtil.binaryToDecimal("0100"));
    }

/*
 * defects: off-by-one in loop, wrong handling
 * of 2s complement.
 */

@Test
public void testBinaryToDecimalStringPositiveOdd() {
    assertEquals(
        "Positive even conversion",
        11,
        b2dUtil.binaryToDecimal("01011"));
}

```

The code shows the use of the static `assertEquals()` method. The first argument is optional and allows a message to be reported if the assertion fails. The second argument is the expected value taken from the test set. The final argument is the actual value produced by the target.

JUnit can be used to explicitly test that a method will throw an exception when it is supposed to. The code below shows how the `@Test` annotation can be parameterised to indicate the expected class of exception that should be thrown.

```

/*
 * defect: null condition
 */

@Test(expected=NullPointerException.class)
public void testBinaryToDecimalStringNull() {
    b2dUtil.binaryToDecimal(null);
}

/*
 * Test empty string
 */

@Test(expected=NumberFormatException.class)
public void testBinaryToDecimalStringEmpty() {
    b2dUtil.binaryToDecimal("");
}

```

Finally, it can be tedious to run all the different test classes for an application separately. Test classes can be run together using a test suite class. The code below shows how to do this.

```

package uk.ac.glasgow.oose2.documentation.tests;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;

```

## Testing for exceptions

## Creating test suites

```

import uk.ac.glasgow.oose2.documentation.tests.
    BinaryToDecimalUtilTestIllegal;
import uk.ac.glasgow.oose2.documentation.tests.
    BinaryToDecimalUtilTestPositive;

@RunWith(Suite.class)
@Suite.SuiteClasses(
{
    BinaryToDecimalUtilTestPositive.class ,
    BinaryToDecimalUtilTestIllegal.class
})
public class AllTests {
    public AllTests(){
        System.out.println(
            "Beginning tests for BinaryToDecimal class... ");
    }
}

```

The `@RunWith` class annotation alters the JUnit test runner class for this class to be the `Suite` class. The `@Suite.SuiteClasses` annotation has a default parameter, which is a list of test classes to be invoked within this suite. The constructor of the test suite class can also be modified to log the suite's invocation.

## Summary

The key point of software testing is that it is almost always infeasible to test software exhaustively, nor particularly efficient. On the other hand software testing can make an efficient contribution to quality assurance processes. Black box testing is effective at assessing a software artifact's conformance with its specification, while white box testing is effective at ensuring higher coverage of source code. Wherever possible, automated techniques should be used to enhance the effectiveness of testing while minimising additional costs.

## 21.7 Exercises

1. Consider the operation specifications shown below:

```
/**  
 * A palindrome is a string that reads the same backwards as  
 * well as forwards.  
 * @return true, if the candidate is a palindrome  
 */  
public boolean isPalindrome(String candidate);  
  
/**  
 * Returns a new string, substituting all occurrences of  
 * oldChar in source with newChar. If matchCase is false,  
 * every occurrence of oldChar in the range a-z,A-Z is  
 * substituted for newChar in the same case as the occurrence  
 * of oldChar. Other characters are unaffected.  
 */  
public String substitute(String source, char oldChar,  
                         char newChar, boolean matchCase);
```

- identify the independent input variables and their equivalence partitions;
  - specify the operational equivalence classes by combining the input variable classes; and
  - propose test cases (with example data values) for the operational equivalence classes.
2. Equivalence classes from natural language specification from Lethbridge and Laganière [2005, pp. 382]:

“The landing gear must be deployed whenever the plane is within 2 minutes from landing or takeoff, or within 2000 feet from the ground. If visibility is less than 1000 feet, then the landing gear must be deployed whenever the plane is within 3 minutes from landing or lower than 2500 feet.”

- what is the method signature for deployLandingGear() ?
  - what are the equivalence classes for the parameters?
  - what are the dependencies between parameters?
  - what are the operation equivalence classes?
3. Construct an activity diagram for the isPrime() function:

```
public static boolean isPrime(Integer candidate) {  
    // naive implementation, for kicks.  
  
    if (candidate < 0) candidate = -candidate;  
    if (candidate == 0) return false;  
    if (candidate == 1) return false;
```

```

for (int i = 2; i < candidate; i++)
    if (candidate % i == 0) return false;

return true;
}

```

- label all the edges in the diagram;
- specify a testing strategy and identify high priority paths;
- propose test cases to exercise the identified paths; and

4. How would you test this system architecture?

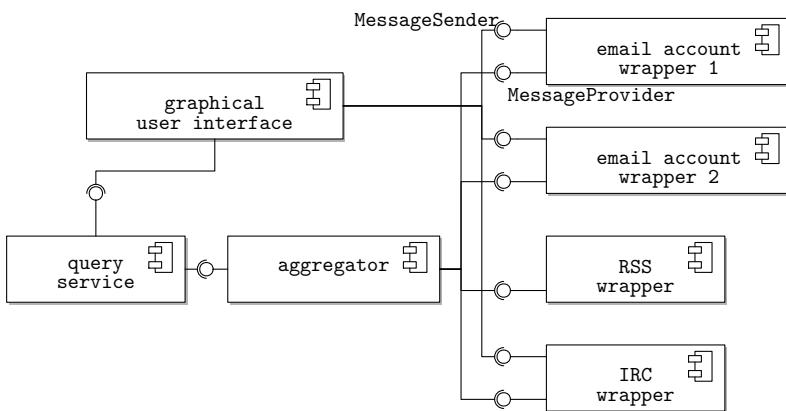


Figure 21.8: integration testing exercise

The diagram shows an architecture for a messaging aggregation application. The application collates messages from multiple messaging accounts, each of which may use a different communication protocol (email, RSS, blogs, IRC etc). Each communication standard is wrapped and accessed via two standardised interfaces for sending and receiving messages.

The messages are stored in the aggregator and queries can be constructed to view messages. The user interface also contains functionality for sending messages.

- Which components would you test first?
- Where are the boundary components?
- What stubs would you need to create?
- Which are the presentation components, how would you test them?
- How would you integrate your remaining tests, and in what order?
- How would you refine the architecture to make testing easier?

5. Consider the following Java method

```

public String caesarCipher(String plaintext) {
    if (plaintext == null) return null;

    String result = "";
    int shift = 3;

    // now encrypt the whole message
    for (int i = 0; i < plaintext.length(); i++) {
        int plainChar = plaintext.charAt(i);

        // set all out-of-bound characters to 'x'
        if (plainChar < 97 || plainChar > 122)
            plainChar = 23;
        else plainChar -= 97;

        // cyclically shift the char
        int cipherChar = (plainChar + shift) % 25;
        if (cipherChar < 0) cipherChar += 25;

        // get the character equivalent and add it to the
        // result
        result += Character.toString((char) (cipherChar +
            97));
    }
    return result;
}

```

Draw an activity diagram to trace the flow of control in the algorithm, treating each statement as an activity and each condition as a branch. You should also include the conditions as guards on transitions where appropriate.

Label each edge of the diagram with a unique number.

Propose three test cases suitable for the method based on your activity diagram. For each test case, state the

- initial input,
- expected output, and
- the list (in order of use) of the edges covered during execution.

Give a short justification of the test cases you have proposed.

## 21.8 Workshop: Agile Practices for Testing

This exercise has three purposes:

- to introduce two *agile* software engineering practices to improve the quality of your source code called *test first programming* and *pair programming*.
- to increase your familiarization with Scanners
- to increase your experience of Computer Aided Software Engineering (CASE) tools, specifically JUnit.

### 21.8.1 Agile Methods

Agile methods are a collection of *software development methods* that prescribe processes and practices for the professional development of software. You can find out more about agile methods from the web, or from this book:

#### PLACE HOLDER FOR beck05xp

For this exercise, you are going to try two different software practices.

#### Test First Programming

The purpose of testing is to identify incorrect behaviour in a software program. Testing involves the design and implementation of *test cases*, each of which describes a possible input for a program and the expected output. Test case suites (collections of test cases) are designed to cover the inputs to a program as comprehensively as possible. In principle, if the test suite is complete and correct, then any code which passes all the tests should also be correct. So, if we write the tests first, we shift the problem of writing correct code to writing correct test cases. The test cases also make the expected behaviour of the program under test explicit. For this exercise, you will use the JUnit package to develop a small suite of test cases, as practice for the assessed exercises. You can find out more about JUnit from the project website.<sup>4</sup>

#### Pair Programming

Source code review is a technique for identifying and correcting errors in software. The idea is that ‘many eyeballs’ scrutinising a source code listing are more likely to identify and correct mistakes, all without actually running the program. Pair programming combines source code reviews with the implementation itself. Pairs of programmers sit at a single terminal and work on one problem. One programmer (the pilot) controls the mouse and keyboard. The other programmer (the navigator) observes the work, identifies mistakes and suggests corrections.

---

<sup>4</sup><http://junit.org/>

### 21.8.2 Scanners

You were introduced to the Java utility Scanner in JP2. A Scanner instance reads a stream of characters, breaking them up into tokens. By default, a Scanner uses white space (blanks, tabs, end-of-line characters) to separate tokens; you may specify that different characters (or patterns of varying complexity) be used to separate tokens in the stream.

Of particular interest is the ability to create a Scanner from a String (a stream of characters from the character at index 0 to the character at index length-1).

### 21.8.3 Procedure

Your tutors will organise you into programming pairs. One of the pair should be the programmer and the other the navigator. The pilot will do the work of controlling the mouse and keyboard, while the navigator watches and checks for mistakes. Note, this is *not* an excuse for the navigator to take time out! For the first hour, you should spend the time working on the test cases. Swap round the roles for the second part of the exercise, when you do the implementation.  
<sup>5</sup>

### 21.8.4 The Problem

The United Kingdom went through a process of *decimalisation* of its currency in the 1970s, meaning that a decimal relationship between denominations of currency (pounds and pennies) was established, replacing the variable relationships that had been used until then. As a consequence it is often hard to work out the value of items priced in the old (sometimes called Imperial) currency. The programming task is to implement a utility for parsing and converting Imperial Pound Sterling currency amounts into the modern decimal Pounds Sterling amounts.

The relationships between the currencies is as follows:

- an Imperial penny (denoted 1d.) is the smallest type of imperial currency;
- an Imperial shilling (denoted 1s.) is worth 12 Imperial pennies;
- an Imperial pound (denoted £ 1) is worth 20 shillings (and thus 240d.);
- a decimal penny (denoted 1p) is the smallest type of decimal currency;
- a decimal pound (also denoted £1) is worth 100 decimal pennies;
- a decimal pound is worth the same as an Imperial pound; and

---

<sup>5</sup>In each lab session, there is a 50% probability of a triple. In this case, there will be two navigators watching one pilot and you should swap round after 40 minutes.

- as a consequence of the above, an Imperial penny is worth  $100/240$  ( $= 5/12$ ) decimal pennies.

An amount of Imperial money is written like this:



## Chapter 22

# Testing and Assurance of Large Scale, Complex Software Based Systems

### Recommended Reading

PLACE HOLDER FOR sommerville10software

Recommended reading (690)

PLACE HOLDER FOR obreshkov10software

### 22.1 Introduction

We investigated some core concepts in software testing in Chapter 21, including:

- Recap of basic concepts in testing (691)

## Contents

[part=24]

If any of these terms are unfamiliar to you, should review the OOSE course notes as a priority<sup>1</sup>.

In this series of lectures, we are going to extend this discussion to consider the application of the principles of testing to large scale software based systems. This section considers some general issues concerned with the testing of large scale systems; Section 22.2 describes the testing of non-functional or

---

<sup>1</sup><http://fims.moodle.gla.ac.uk/course/view.php?id=163>

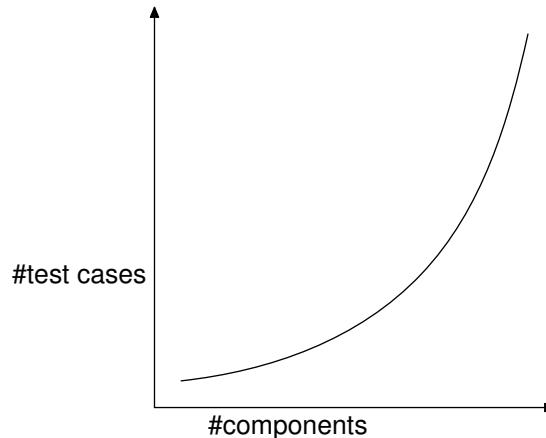


Figure 22.1: combinatorial explosion in software testing

emergent properties of systems; and Section 22.3 discusses the management of test programmes within large teams of software developers over potentially lengthy periods of time.

Recall from the OOSE course that planning for software testing means identifying a trade-off between:

- maximising the test coverage of a system to increase the number of defects discovered; and
- minimising the cost of conducting testing per defect discovered.

In addition, recall the problem of combinatorial explosion in software testing. The greater the number of possible input parameters to a system, the harder it becomes to ensure high coverage of a system. Figure 22.1 illustrates the problem of combinatorial explosion in software testing.

As the number of modules (lines of code, functions, classes, components or sub-systems) in a system increases linearly, the complexity of the system will increase *exponentially* because of the potential dependencies between those modules. Consequently, the number of combinations of modules and their potential combinations of inputs also increases exponentially.

The difficulty of scaling effective testing quickly becomes apparent for large scale systems. The issue of scale and complexity can affect a software project in many different ways:

- computational scale and complexity;
- number of software modules and inter-dependencies (see OOSE);
- number of software platforms;
- variations in build and configuration of deployments;

Combinatorial  
explosion in software  
testing (692)

Defining scale and  
complexity in software  
systems (693)

- duration and variation in inputs;
- software development team(s) size, location and experience;
- number of geographic locations and network connections; and
- number and variation in users.

Lets examine some particular aspects of scale and complexity in testing with some case studies.

Large scale software development efforts, consisting of hundreds of thousands, or millions of lines of code, are often divided amongst one or more teams of developers. Each team of developers is given responsibility for the development, testing and delivery of one or more sub-systems within the overall development effort. These *heterogeneous* software projects may be populated by:

- different business units within a single organisation;
- business units from different subsidiaries within the same umbrella organisation, such as a multi-national company; or
- project teams within different organisations brought together as part of a consortia.

Heterogeneous  
software projects  
(694)

Significantly, the greater the variation in organisational culture (i.e. the different outlooks, social backgrounds and management styles) between the different teams working on the project, the harder it can be to develop a consistent test programme for a software project. Each organisation will develop and deliver its own test plan, even though there may be agreed standards, natural variation will occur, so it becomes harder to achieve a consistent and coherent software quality for the overall system.

The ATLAS particle physics project (based at the Large Hadron Collider at CERN) is supported by a large scale software project [Obreshkov, 2010]. The project is characterised by (approximately):

Example: the ATLAS  
Project (695)

- 7 million source lines of code;
- 10 major sub-projects;
- multiple platforms, languages, compilers and configurations;
- 500 developers;
- 3000 users (scientists and engineers); and
- 40 countries.

The project team have developed an extensive build and test infrastructure for managing the collaborative software effort. This infrastructure includes several different test stagings:

- ATLAS Testing Nightly
- ATLAS Run Time Tester
- ATLAS Nightly Build
- Full Chain Test
- Tier0 Chain Test
- Big Chain Test
- SampleA Test

Testing the ATLAS Software (696)

Note that:

- testing is integrated as far as possible into the build process for the entire project toolset;
- individual tests are developed at the discretion of module users and developers;

ATN Test results report (697)

*Socio-technical* systems (sometimes called computer-based systems) incorporate both computer software and hardware, the computer system's users, and the surrounding organizational and cultural practices. Consequently, the system boundary is drawn around the organisation as a whole: the software and hardware systems are just some of the components to be considered in the overall system. This perspective introduces socio-technical issues of scale and complexity for testing a system.

From a socio-technical perspective, the organisation as a whole must be tested when a new software system is introduced, and not just the software itself (although this remains important). Introducing a new software system will cause the rest of the organisation to evolve in response. Consequently, the whole system must be tested for potential 'defects' which have been introduced by the system evolution. Unfortunately, testing socio-technical systems effectively is challenging for several reasons:

- the members of the organisation may not be available to take part in tests;
- obtaining realistic test scenarios may not be practical;
- identifying and covering the *real* working practices; and
- the organisation may evolve independently of the system.

Challenges testing socio-technical systems (698)

Sometimes referred to as joint systems in military contexts, or coalitions of systems in more recent literature, systems-of-systems represent the extreme end of the scale for challenges in software testing. A system of system represents multiple heterogeneous semi-autonomous systems that cooperate or are coordinated to produce emergent effects. Examples include electronic stock exchanges, military coordination systems (across multiple arms of a military or different militaries) air traffic control systems and supply chain control systems.

Significantly, the system-of-systems is not under the control of any one actor. Individual systems may evolve independently of neighbouring systems and the overall system-of-systems. Testing a system-of-systems is extremely challenging because:

- each of the member systems will have different cultures and practices;
- the system cannot be isolated or configured for a test;
- one of the member systems may evolve during a test; and
- the expected outputs for the system are not easy to define.

Systems-of-systems  
(699)

In summary, increasing scale in a software or computer based system reduces the coverage and effectiveness of testing efforts, even if a proportionate number of resources are applied to testing.

## 22.2 Testing Non-functional Requirements

Recall from Chapter 11 that non-functional requirements describe emergent characteristics or properties of the overall system that must be measured to be satisfied rather than observed as a provided feature as for functional requirements. Consequently, non-functional requirements cannot typically be checked for in the design documentation; they can only be tested once an implementation of a system has been realised. The implementation tested may be the final system, or a prototype. Alternatively, it may be possible to build a model from which the characteristics of the final system can be approximated. An advantage of testing non-functional requirements is that they may act as an aggregate for more complex functional properties that are too complex to test individually.

In one sense, developing tests for non-functional requirements is the same as conducting a scientific experiment. The tester formulates a hypothesis concerning the system under test (the requirement), and then constructs an experiment to determine whether the hypothesis is satisfied. A well formulated non-functional requirement should state the:

- property of the system to be measured;
- metric to be used;

Information needed to  
test non-functional  
requirements (700)

- operating conditions in which the requirement must be satisfied; and
- threshold the system behaviour must exceed to satisfy the requirement.

There are different classes of testing for non-functional requirements, described in the following sub-sections.

### 22.2.1 Reliability, Safety and Security Testing

Reliability testing  
(701)

The reliability of a system is the extent to which a system performs according to its specification.

Metrics include:

- probability of failure on demand provides an estimate of the probability of failure each time a system is accessed;
- mean time to failure, indicating how long it takes for a software system to fail. This is a useful metric for systems where access for repairs is difficult or expensive, such as space based systems. Ideally, the mean time to failure should be greater than the mission time for such systems; and
- down time per year, indicating how much time can be lost to system outages over a year. This is useful for high demand, continuous use systems, such as high volume transaction processing. Payment systems, for example, are often designed to have down times of less than a few seconds per year.

Examples of safety-critical contexts (702)

The operation of many systems occurs in conditions which are potentially harmful to the system or to humans who operate or depend on the system. These dangers may arise from an adverse environment in which the system operates, or the nature of the system itself, should it malfunction. Examples of software used in safety critical contexts include:

- automatic pilots for transportation, such as aeroplanes;
- equipment used in the operation of particle beam radio-therapy equipment;
- space based technology, such as satellites, space craft and planetary rovers; and
- nuclear power station management systems.

These systems are called *safety critical* because a principle concern for the system design and evaluation process is to gain concern that the operation of the system will not cause the system or its users harm. If software is used to

control the system's behaviour then assurance must also be gained about its safety.

Testing the safety of a system is concerned with the development of test cases which demonstrate that harm will not be caused in the adverse environment in which the system will be used.

Methods for testing the safety of a system include:

- model driven development, which allows implementation software and test suites to be generated automatically from platform independent models. Test cases are derived from assertions about legal and illegal states for the software model (see Chapter 25 for more information on the use of formal methods in software engineering);
- Cleanroom development which uses automated and statistical methods to select test cases for execution, and is used to calculate an estimate of a software system's pfd, and a confidence in the estimate; and
- Hazard and Operability Study (HAZOPS) derived from chemical engineering and similar activities. For software, a HAZOPS like process can be used to identify flows of information between software components. Questions can then be asked about the flows of information, such as what happens when the information is incorrect, or arrives too early.

Methods for testing system safety (703)

The emphasis in the use of formal methods for testing is to gain accurate measurements as to the reliability of a system, to provide input into a *safety case*. Safety cases are justifications that a system meets its safety requirements. In contrast, the purpose of a HAZOPS is to explore the consequences of a failure in one or more components for the rest of an information system. HAZOPS are used to gain assurance as to the reliability and safety of a system in the presence of failures.

As well as being adverse, a system environment may also be considered *hostile*. A hostile environment is assumed to contain active agents (sometimes called *threats* or *attackers*) whose goal is to deliberately penetrate, subvert and/or sabotage the deployed system. Examples of systems deployed in hostile environments include:

- electronic voting systems threatened by agents who wish to subvert the political result of an election;
- automated teller machine cash dispensers which are threatened by thieves who would like to gain access to their administrative user interfaces in order to steal money;
- friend-or-foe detection systems on automated weapon systems, which are threatened by enemies who wish to trick them into firing on friendly aircraft, or not firing on hostile aircraft;

Examples of systems operating in hostile environments (704)

- digital rights management systems, threatened by copyright 'pirates' who wish to make unauthorised copies of digital media for re-sale;
- network security systems such as firewalls threatened by attackers who wish to gain unauthorised access to vulnerable services on remote servers; and
- authentication systems threatened by attackers who wish gain unauthorised access to information resources.

Distributed systems in particular (or their components), are often considered to operate in hostile environments, because some or all of the other network participants are not known to the system itself.

System threats actively search for *vulnerabilities*, defects in the system that can be *exploited*. Consequently, it is necessary in these contexts to conduct tests to gain assurance that the system is resistant to such attacks. Security testing metrics include:

- attacker capabilities or knowledge required to penetrate the system;
- time and resources to penetration, giving an estimate of how long it would take an adversary, and how much resource would need to be applied in order to penetrate a system. The idea comes from assessments used for physical safes which are given a time and resource rating;
- unpatched vulnerabilities per line of code;
- patches issued per year;
- successful penetrations per year;
- attack surface exposure.

#### Security testing metrics (705)

Methods for security testing include:

- vulnerability testing including:
  - SQL injection testing;
  - port scanning;
  - fuzz testing, in which a software system is supplied with unusual or malformed (fuzzed) inputs that may cause the system to behave in unexpected (and insecure) ways. Fuzz testing is a way of checking the extent to which inputs are validated before they are passed on to business logic code. Fuzz testing is an effective means of detecting buffer-overflow vulnerabilities, for example;

and

- penetration testing, in which a *red team* or *attack team* is tasked with gaining unauthorised access to the resources protected by a system, equipped with the same resources expected of an attacker. Penetration testers may concentrate on technical vulnerabilities or also engage in social engineering to gain access to a system.

### 22.2.2 Performance Testing

Performance testing is concerned with assessing how a system copes with different rates of input.

Performance testing  
(707)

- throughput – the amount of transactions that the system can process in a given time;
- demand – the maximum rate of transactions that the system can process; and
- response – the average or maximum length of time taken to respond to a transaction request.

### 22.2.3 Threshold Testing

Once a property and metric has been identified, there are two types of test that can be performed with respect to the threshold.

Types of threshold  
testing (708)

- limit testing is concerned with demonstrating that a system behaves normally within a required limit; and
- stress testing is concerned with discovering what happens when a particular limit is breached.

Limit testing is used to demonstrate that a system behaves as expected within required operating limits. Conversely, stress testing is used to *discover* how a system behaves beyond the operating limits specified. Typically, stress tests are used to demonstrate that a system continues to behave *reliably* or *predictably* when operating outside of normal limits. For example, a system's response to excessive demand may be revert to a 'safe mode' offering only a limited service to its clients. During the 9/11 terrorist attacks, for example, the BBC's News Website experienced unusually high requests from users attempting to obtain up-to-date information. The site reverted to a simpler appearance with less images and video, which required less bandwidth to transmit.

Alternatively, it may be required that the system performs a 'graceful' safe shutdown, rather than fail completely. Database servers, for example, are often equipped with Uninterruptable Power Supplies (UPS) which are essentially large battery packs. If a power loss is experienced, the battery power is used while the server completes any outstanding transactions on the database and then initiates a system shutdown.

## 22.3 Managing Testing within Teams

The testing process for a large system should begin alongside the requirements elaboration stage of a software development project, as use cases and their corresponding scenarios are developed. As for other aspects of large scale software development, software testing is a typically a collaborative effort within a team.

For larger software projects, software testing may be the responsibility of one or more quality assurance, or test teams. The role of these test teams is to develop tests against the system and component specifications provided by requirements, design and implementation teams, according to organizational standards. This arrangement reflects a key principle of testing within teams: *it is better to have someone else test your implementation.*

It is also a good idea to make one person responsible for ensuring that software testing and other quality assurance activities get done. Quality assurance is not always a popular role, but it is an important one in producing high quality software products.

As for other aspects of a software project, the test programme needs to be agreed and documented so that it can be shared with others. A project *test plan* complements a requirements specification by documenting the steps that will be taken to demonstrate that the final system design and implementation satisfies its specification. A test plan consists of:

- a policy for implementing and documenting unit tests, including general test procedures;
- unit test cases describing the testing of the smallest functional modules of the system (typically individual classes);
- integration tests and strategy, describing the order in which units are integrated into sub-systems and tested;
- a *regression test* schedule for unit and integration tests;
- acceptance test cases, describing the test cases that have been derived from the overall system requirements specification (See Section 22.3.1; and
- an acceptance test demonstration plan (See Section 22.3.3).

The use of unit and integration tests for defect testing are described in the OOSE lecture notes. The following sections discuss other aspects of a software team project plan necessary to complete an acceptance test demonstration.

### 22.3.1 Developing Acceptance Tests

The purpose of *acceptance* or validation testing is to demonstrate that a system meets the requirements of the customer. In contrast, defect testing, as

Managing testing within teams (709)

Contents of a test plan (710)

	<b>defect tests</b>	<b>acceptance tests</b>
<b>goal</b>	discover defects to be remedied	demonstrate that the system meets its specification
<b>derived from</b>	architecture and design documents	requirements specification

Table 22.1: defect vs acceptance testing

<b>use case</b>	<b>test case</b>
description	system fixture and interface
includes	nested test cases
scenarios	test data set
pre-conditions	pre-test setup state
post-conditions	test-pass conditions
non-functional requirements	properties and metrics for non-functional tests
priority	regression test plan

Table 22.2: deriving acceptance tests from use cases

described during the OOSE course is used to discover defects to improve the quality of a system. Defect testing is often based on component specifications and internal system design information. Table 22.1 summarises the difference between acceptance and defect testing.

Acceptance tests are derived from the overall system specification, including in particular the system's the use cases, scenarios and non-functional requirements. Table 22.2 illustrates the mapping from the parts of a use case to a test case:

- the use case description identifies the system boundary object to be interacted with during testing. The description also gives the procedure to be followed when executing the test case, including the order in which inputs should be supplied to the system;
- included use cases indicate that one or more test cases derived from other use cases *must* be invoked as part of the overall procedure for the use case under test;
- extended use cases indicate that one or more test cases *may* be invoked as part of the use case under test. Recall that an extension is an optional part of a use case that is not always invoked;
- pre-conditions are used to guide the pre-test setup of the system under test, including the construction of any fixtures for unit tests;

Acceptance vs.  
defecting testing  
(711)

Deriving acceptance  
tests from use cases  
(712)

Identifier	TC5.2.1
Use case	add keeper
Scenario	primary
Setup	System initialised with data/test-users.db and data/test-books.db
Interface	src/uk/ac/glasgow/minder/MinderSystem5
Includes	TC5.1.1 (login primary) TC5.5.1 (search for user primary)
Procedure	JUnit Test Case: src/uk/ac/glasgow/minder/tests/accept/keeper-admin/ AddKeeperPrimary
Inputs	searchString="Singer, J" confirm=true
Outcome	user Singer is set as a keeper

Figure 22.2: documenting acceptance tests

- post-conditions are used to describe the situations in which the test case is passed or failed, depending on provided inputs; and
- use cases scenarios can be used to define the set of inputs for a test case. Collectively, these inputs are known as the test case *test data set*. Scenarios describe the different routes through the activities of a use case, depending on the inputs supplied and the state of the system. If a comprehensive set of scenarios has been developed, then a corresponding set of test cases (one per scenario) should provide good coverage of the system functionality.

Notice also that elements of a test data set may be shared between different use cases. It may be more sensible to group common test data into a separate section of the test plan document and reference the test data sets as variables. This section is referred to as the test plan test data set.

Figure 22.2 illustrates an approach to documenting acceptance tests, using the add keeper use case as an example. The documentation is based on the approach to documenting use cases described in Chapter 12.

The first three rows of the document record the test case meta-data, including an identifier, the use-case and scenario the test case is derived from. The test case records any setup steps that should be taken before the test case is invoked. The includes section records that specific test cases for the login and search user use cases must be invoked as part of the add keeper use case. Finally, the procedure section records that the test case (AddKeeperPrimary) has been implemented as an automatic test against the MinderSystem facade, using specific inputs derived from the primary scenarios for add keeper, login and search user. The outcome states what checks are made in the automatic

## Documenting acceptance tests – example (713)

Identifier	TC-NF-2
Use case	system-wide
Requirement	NF 2
Property	Demand
Metric	Concurrent users
Threshold	$\geq 5$
Setup	System initialised with data/test-users.db and data/test-books.db
Interface	src/uk/ac/glasgow/minder/MinderSystem
Included	TC5.1.1 (login primary)
Procedure	JUnit Test Case: src/uk/ac/glasgow/minder/tests/accept/nf/Demand2

Figure 22.3: documentation test cases for non-functional requirements

test suite.

Figure 22.3 illustrates a second example of a test case document, this time of the key information to be documented for testing non-functional requirements. Each piece of information is mapped directly from the add keeper use case for the branch library management system. In the example, the documented test case is implemented as an automated JUnit test case. The test is conducted against a facade interface (`MinderSystem`) which would normally be interacted with from the user interface.

Documentation for testing non-functional requirements (714)

### 22.3.2 Regression Testing

As has already been discussed, the purpose of defect testing is to discover defects in a software program. Software test cases may be developed as part of the implementation process (following test first coding practices, for example) or when a failure is observed by a user and needs to be documented. It can be tempting to discard a test case once it is successfully passed by the system. However, it is not unusual for previously removed defects to be re-introduced into a system as the software evolves. *Regression* testing is used to determine whether a change in the software has resulted in the introduction of a defect.

The regression testing process is shown below.

Regression testing (715)

1. Commit new version of software
2. Build and configure
3. Select regression tests to execute
  - priority
  - randomly

- source of change
4. Conduct test.
  5. Open tickets associated with failed tests, due to:
    - changed requirements
    - re-introduced defects

As we have discussed, effective testing is expensive, so conducting regression tests also adds to this costs. It is not always practical to re-run every test case each time a software system is changed. The use of automated testing can ease this problem, but there are still practical limits. The testing strategy adopted by the ATLAS team illustrates this issue. Even though the tests are largely automated, not all tests are run each time a commit is made. However, there are periodic builds and tests of the entire tool chain for the project when major versions are released. Similarly, the project is allowed to proceed even though not all tests will pass for each build of a an individual package.

### **22.3.3 Conducting Acceptance Test Demonstrations**

Live demonstrations are often included in software project contracts in order to show to a customer that the specification for a system has been satisfied. Acceptance test demonstrations are similar to prototype demonstrations (See Section 16.4.1) except that:

- a fully functional and interactive system is employed; and
- the primary goal of the demonstration is to show that the system meets its specification, rather than to discover new or incorrect requirements.

Consequently, an acceptance test demonstration should:

- have a clear plan (see below);
- ensure that key use cases are demonstrated satisfactorily;
- ensure that any key non-functional requirements are demonstrated as being realised. Some non-functional system properties may not be demonstrable within the time allotted (such as mean-time to failure), so other mechanisms may be required for this;
- be practiced before the final demonstration, ensuring that the key tests can completed within the time allotted for the tests;
- be conducted within a clearly defined system environment, including an understanding of how any dependencies will be configured and input data to be used; and

- stick strictly within the limits of the agreed requirements specification, while allowing flexibility at the request of the customer within these limits.

An acceptance test plan should include the following:

- an identification of roles to be undertaken during the demonstration. These roles are similar to a prototype demonstration, including:

- demonstrator,
- narrator (describing and explaining the actions taken by the demonstrator),
- secretary and
- question handler.

These roles may not be distinct in a demonstration. It may be appropriate for example for the role of demonstrator and narrator to be given to a single person, or for the role of question handler and narrator to be merged. Alternatively, the team may decide to allow the customer to act as a demonstrator, driving the system for themselves;

- a description of how the system will be configured for the acceptance test demonstration, noting in particular any parts of the demonstration that will not be the same for a real deployment;
- a script of key acceptance test cases (based on use case scenarios) to be tested during the demonstration. The script should provide a quick reference for a demonstrator for the data inputs and actions to be taken at each stage of the demonstration. The script should prioritise test cases for high priority use cases, while leaving lower priority use cases for later in the demonstration, if time permits.

The key purpose of the acceptance test plan is to *prepare* the demonstration team to deliver a professional presentation of the system's features. The plan may not actually be used at all during the demonstration because the process of preparing the plan has provided all members of the demonstration team with an understanding of the steps that will be taken.

#### **22.3.4 Automating Acceptance Tests**

There are a number of frameworks for automating

### 22.3.5 Evaluating Test Plans

There are several different approaches to evaluating the effectiveness of test programmes:

- Document and software inspections;
- Statistical analysis; and
- Mutation testing.

## Summary

In summary, issues of scale and complexity constrain effective testing of large scale systems more than efficiency. Testing emergent properties rather than functional features provides a means of evaluating the overall behaviour of a system, when testing individual components effectively is impractical. An alternative strategy is to automate as much of the testing process as possible to reduce testing costs.

## 22.4 Workshop: Parameterized Tests and Theories in JUnit

//TODO

# Chapter 23

## Modelling System State

### Recommended Reading

PLACE HOLDER FOR sommerville10software

PLACE HOLDER FOR bennett06object

PLACE HOLDER FOR harel88visual

### 23.1 Introduction

Previous lectures have investigated how UML can be used to model different aspects of a software system.

Previously in PSD  
(718)

- Initially, we explored how to model the system and its boundary with the environment using use cases (Chapter 11).
- Chapter 12 discussed how to refine and model the interactions that occur between a user and the system with activity diagrams, and Chapter 14 discussed the use of class diagrams to document the elements of a problem domain that must be represented within a system.
- Finally, Chapter 14 discussed how the activities at the system boundary described by activity diagrams can be extended into the system itself using sequence diagrams. Sequence diagrams are useful for discovering the collaborations that must occur between objects in a system architecture in order to realise different use cases.

Recall that an object oriented system consists of a collection of objects, the relationships between them and the internal state of each object's primitive attributes. Collectively, this is the system's *state*. This chapter explores the use of UML notation to model changes to system state when a user takes an action with a system.

Effects of operations on a system (719)

When a use case is invoked by a user, it results in a sequence of operation calls between objects in the software system. These operation calls:

- query an object for information;
- perform calculations;
- emit a message or generate an event; or
- alter the state of the system by
  - creating new objects,
  - destroying objects that are no longer needed,
  - set primitive attribute values of an object, or
  - create associations between objects.

Modelling state in UML (720)

Some of the side effects of a use case will be recorded in the post-conditions in the use case description. However, the requirements specification is not concerned with all changes in state, as many are design considerations. *State Machine Diagrams* are a UML notation for modelling the effect of internal and external events (user actions) on a system's state. They are an extension to the *state transition network* (STN) formalism.

State transition networks (721)

## 23.2 State Transition Networks

State transition networks are themselves an extension of the finite state machine notation. A state transition network consist of:

- a finite number of one or more *states*;
- exactly one *start* or *entry* state;
- zero or more *stop* or *exit* states;
- an alphabet of *events*;
- a set of *transitions* between states, with each transition conditioned on an event; and
- an optional *operation* associated with each transition.

STN example 1 (722)

Figure 23.1 illustrates an example state transition network for a light switch button. The button has:

- one event *click*;
- two states *off* and *on*;

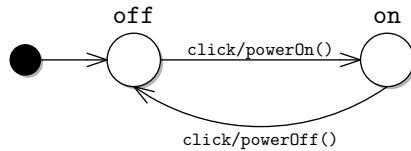


Figure 23.1: a simple state transition network for a light switch

- an entry state off;
- two transitions off → on and on → off, both of which are dependent on the click event; and
- two operations for controlling the power to the light when the button is pressed powerOn() and powerOff().

Notice that the example STN does not have any terminal states.

Figures 23.2 and 23.3 illustrate how the message passing between objects on a sequence diagram can be mapped to effects on a particular object's state on a state transition network.

Figure 23.2 illustrates the sequence of messages that are passed between a client and a server in order to establish a network connection. CP is a *stateful* protocol, which means that data can only be transferred once both parties have completed a connection *handshake*. To do this, a TCPClient sends a SYN request to a listening TCPserver. The server then responds with a SYN+ACK message, confirming that both requests a connection for the server to the TCPClient, and acknowledges the TCPClient's SYN message. Finally, the TCPClient sends an ACK message confirming the TCPserver's connection to the TCPClient.

Figure 23.3 shows the effects of the TCP protocol messages on both the TCPClient and TCPserver's state as the messages arrive. The TCPserver is enabled by the LISTEN signal.

The TCPClient is instructed to establish a connection with the CONNECT signal. This causes the sendSyn() operation to be invoked, and the SYN message

STN Example 2 –  
TCP connection  
establishment (723)

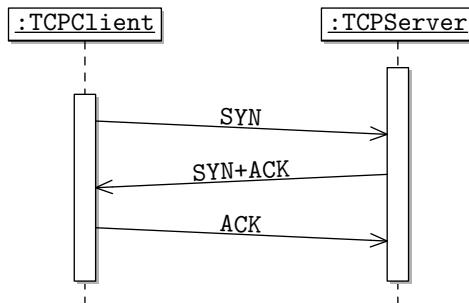


Figure 23.2: the TCP connection establishment sequence

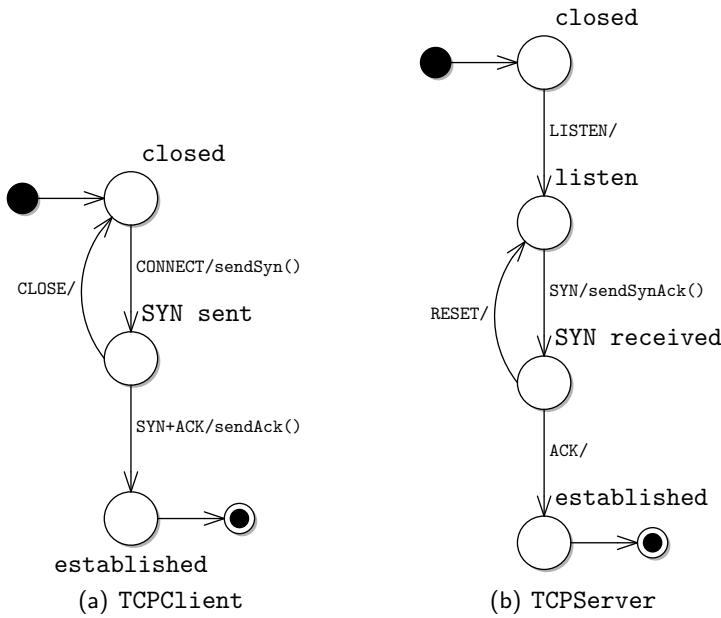


Figure 23.3: the TCP connection establishment state transition networks for the TCPClient and TCPServer objects

to be sent to the TCPServer. When the TCPServer receives this message it transitions to the SYN received state and issues the SYN+ACK message back to the TCPClient. On receiving this message, the TCPClient transitions to the established state and issues an ACK message. Finally, the TCPServer receives the ACK and transitions to the established state as well.

The state transition networks in Figure 23.3 also show how the TCPServer or TCPClient objects cope with errors, such as lost messages. The TCPClient can have the connection process terminated before it is established if the CLOSE signal is received. This may cause the user of the TCPServer to timeout waiting for an ACK, so this object can also terminate the connection if a RESET signal is received.

State transitions can be used to model lots of other situations, such as:

- finite state machines;
- turing machines;
- a telephone call;
- a biological system; and
- the behaviour of a user interface.

There are several extensions to the STN formalism that address some of the limitations of the existing features, particularly with regard to the problem of

STN for TCP  
connection  
establishment (724)

Other uses of STNs  
(725)

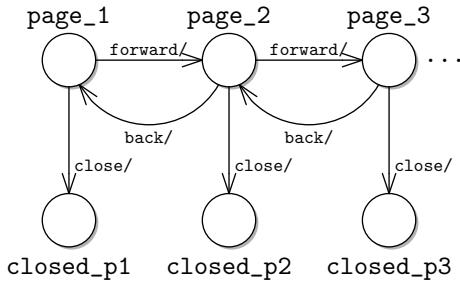


Figure 23.4: state explosion in an STN

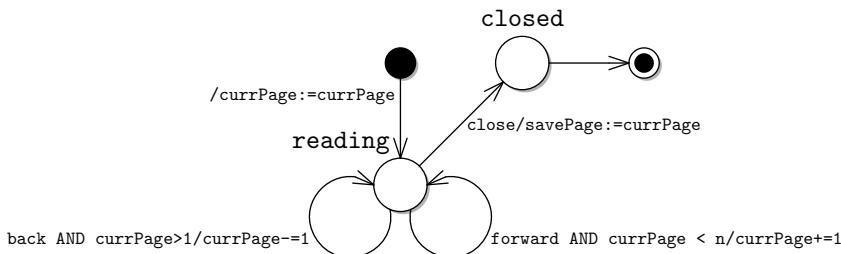


Figure 23.5: using local variables in STNs

*state explosion*, that is, the number of unique states that any non-trivial system may be in quickly becomes two difficult to manage in a state transition network. Figure 23.4 gives an illustration of this problem.

State explosion (726)

The figure illustrates an STN for part of an electronic book reader. Each time the reader moves forward or backward one page, the system enters a new state (the page on display). In addition, the reader saves the current page when the book is closed, so a state will be needed for each page the book can be closed on. This means that the system will have at least twice as many states as the book has pages. This is problematic not least from a manageability point of view, but also because it means that a separate design will be needed for each book in the reader that has a different number of pages.

This is clearly an unsatisfactory solution, caused by the fact that very similar states have to be all treated as unique. Figure 23.5 illustrates how the use of local object attributes can be used to reduce the number of states in a model by grouping similar states together.

STNs with variables (727)

The figure shows the same electronic book reader as above, although this time the system has only a `reading` and `closed` state. Each time the user presses `forward` (and there is another page in the book) the reader adds 1 to the current page count (`currPage`) and returns to the `reading` state. When the user closes the book, the `currPage` value is stored in the `savePage` attribute for next time.

Figure 23.6 illustrates another approach to reducing the number of states shown on an STN. The figure shows the use of *sub-network* or *recursive states*,

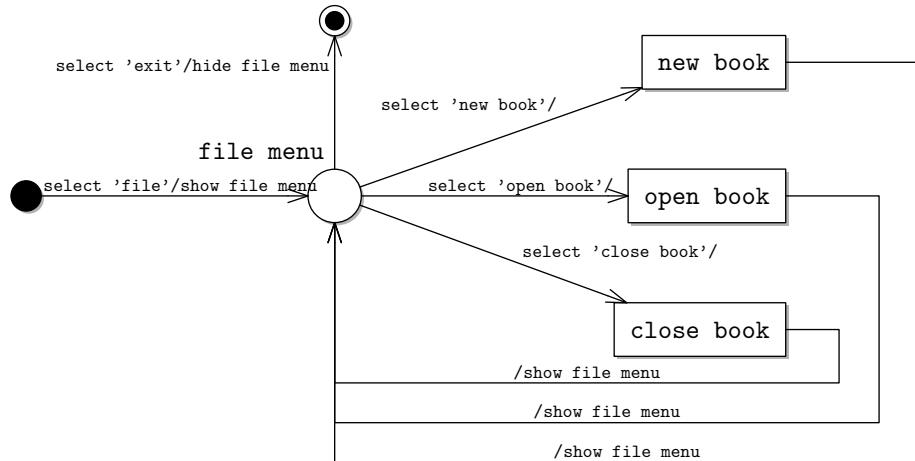


Figure 23.6: STN with recursion

which are entered and processed just like a normal state transition network. When the sub-network terminates, the calling network carries on as before.

The figure illustrates the states of a file menu for the electronic book reader. When a user selects 'file' the file menu state is entered. From this state, the system may move to one of three sub-networks, depending on the choice made by the user. When the sub-network activity finishes, the file menu is shown again, and the system moves back into the file menu state.

Despite these extra features of STNs, they still have a number of limitations. In particular, they are poor at expressing:

- systems with large numbers of states;
- global actions, such as escape or help;
- error handling mechanisms; and
- concurrent and interleaved operations.

Notice that in the TCP example above, two different state machines were used for the TCPCClient and TCPServer. In summary, STNs are unsuitable for realistically sized systems with a large number of states.

### 23.3 The State Machine Diagrams Notation

State Machine Diagrams were proposed as a means of addressing some of the limitations of state transition networks. Harel [1988] proposed state machine diagrams as state diagrams with:

- *depth*, such that states can encapsulate other sub-states;

STNs with recursion  
(728)  
Limitations of STNs  
(729)

State Machine  
Diagrams as  
extensions of STNs  
(730)

- *orthogonality*, such that states can be partitioned into sub-groups; and
- *broadcast communication*, to support concurrency.

The basic state machine diagram notation is illustrated in Figure 23.7 and consists of:

- states enclosed in a labelled lozenge (rounded rectangle);
- one or more exit states denoted by a disc surrounded by a circle;
- sub-states denoted within the super-state lozenge;
- transitions between states guarded by events; and
- an optional internal initial transition for a state to one of its sub-states, denoted by a solid black disc.

State Machine  
Diagram notation  
(731)

Notice that transitions can occur from any state to any other state in the state machine diagram, regardless of the level of nesting. In the figure, an `anEvent` event will cause the the first substate to transition to its enclosing state. However, the initial state for the enclosing state is the first sub-state, so the overall state transition will end there. An `anotherEvent` will cause a transition to the second substate and immediately exit the model.

Figure 23.8 illustrates the notation for state machine diagrams using a state machine diagram for a DVD player tray. The tray has two substates `Open` and `Closed`, which are grouped in the `DVDTray` state.

When the tray starts (i.e. when the power is turned on), it defaults to the closed position, due to the inner start state transitioning to the `Closed` state. After that, each press of the eject button causes the tray to be either opened or closed. Regardless of what state the player is in, each time the button is pressed, the light on the DVD player is flashed, due to the invocation of the `flashLight()` operation. If the power is turned off, then the state machine diagram terminates.

State Machine  
Diagram example  
(732)

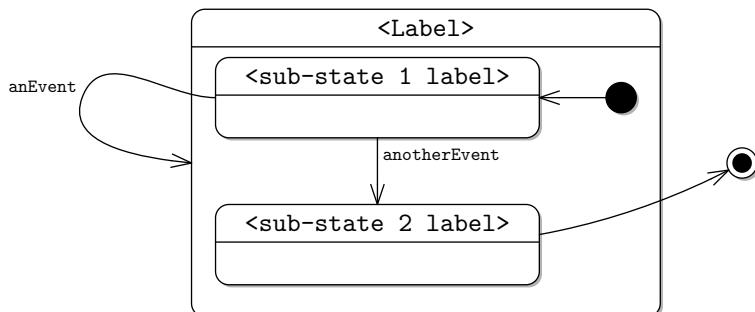


Figure 23.7: the state machine diagram notation

state transition  
networks vs. state  
machine diagrams  
(733)

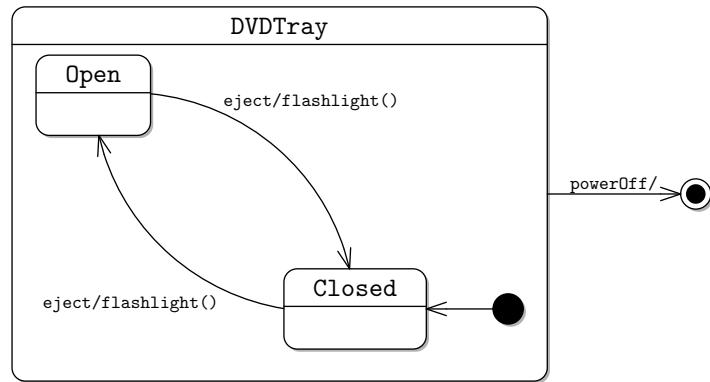


Figure 23.8: a state machine diagram for a DVD player tray

The state machine diagram in Figure 23.8 doesn't really illustrate the advantage of using the state machine diagrams notation over using a state transition network. Figure 23.9 illustrates how it can be beneficial to utilise the grouping and nesting of states to reduce the complexity of a state model.

Figure 23.9(a) illustrates a state model represented as a state transition network. The model consists of three states (A, B and C), a start state (A) and a terminal state. There is an alphabet of five events e, f, g, h and k. Finally there are six transitions labelled with events from the alphabet:  $A \xrightarrow{e} B$ ,  $B \xrightarrow{f} A$ ,  $C \xrightarrow{f} B$ ,  $A \xrightarrow{g} C$ ,  $B \xrightarrow{h} C$  and  $B \xrightarrow{k} \text{end}$ .

Notice that states A and C are quite similar. Both states transition to state B on an f event. If the two state were merged it would substantially reduce the complexity of the model. The g event could be modelled as transitioning back

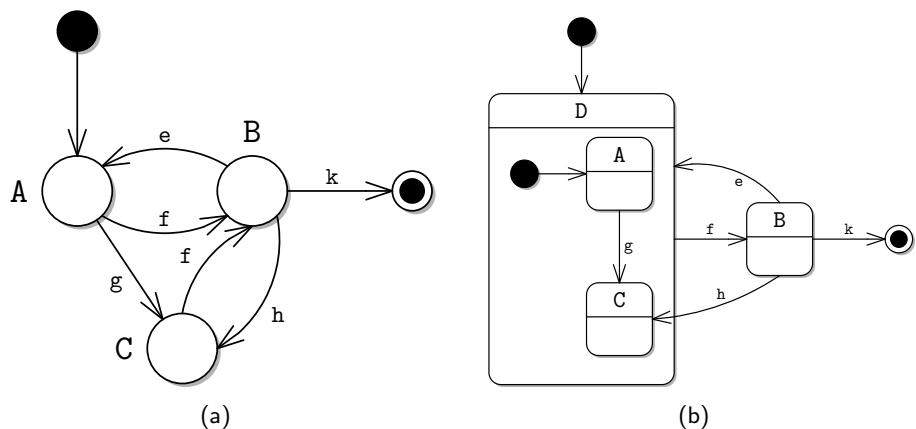


Figure 23.9: state transition networks (Figure 23.9(a)) vs. state machine diagrams (Figure 23.9(b))

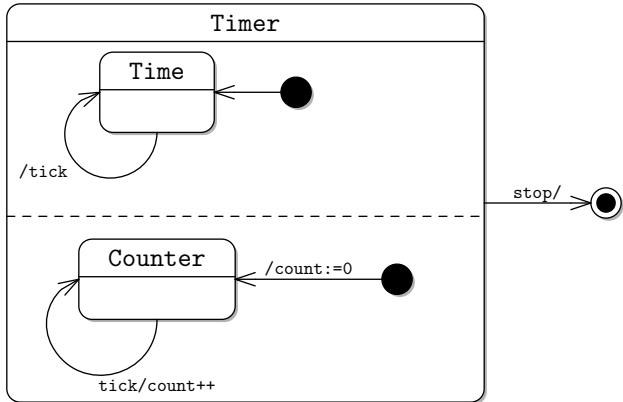


Figure 23.10: modelling concurrent states and broadcast events with state machine diagrams

to the combined state. Unfortunately, the transitions from B to the two other states are different, making the merger impossible in the STN notation.

Figure 23.9(b) illustrates the same state model as a state machine diagram, taking advantage of the benefits of nesting and grouping of states in the state machine diagram notation to simplify the transitions that occur. In the figure, transitions  $A \xrightarrow{g} C$ ,  $B \xrightarrow{k} e$  and  $B \xrightarrow{h} C$  are unchanged.

However, new state D has been introduced, which encapsulates states A and C. State A is the default start state for state D, so is automatically entered when D is entered, either when the state model starts, or when a  $e$  event occurs in state B. If an  $f$  event occurs in either state A or C, then the state machine will transition to state B, exactly as for the STN in Figure 23.9(a).

### 23.3.1 Concurrency and Message Broadcast

Figure 23.10 illustrates the use of concurrency and broadcast of events on state machine diagrams. The figure illustrates a Timer that counts the ticks of time once it has been started. The timer is divided into two concurrent states: a Time state which broadcasts ticks (presumably at regular intervals of time) and a Counter state, which increments the count variable each time a tick event is broadcast.

The division of the Timer state is indicated by a dashed line across the Timer's compartment. Notice that both of the substates have begin states denoted. In the case of the Counter, the transition from the start state causes the count variable to be reset. The Timer is stopped when the stop event is received.

Figure 23.11 demonstrates another example of concurrency in state machine diagrams, using the TCP connection establishment protocol illustrated in Figures 23.2 and 23.3. In the figure, the state transition networks from Figure 23.3 are grouped together in a single state TCP Connection Establishment.

Concurrent states and  
broadcast events  
(734)

Combined TCP  
connection state  
machine (735)

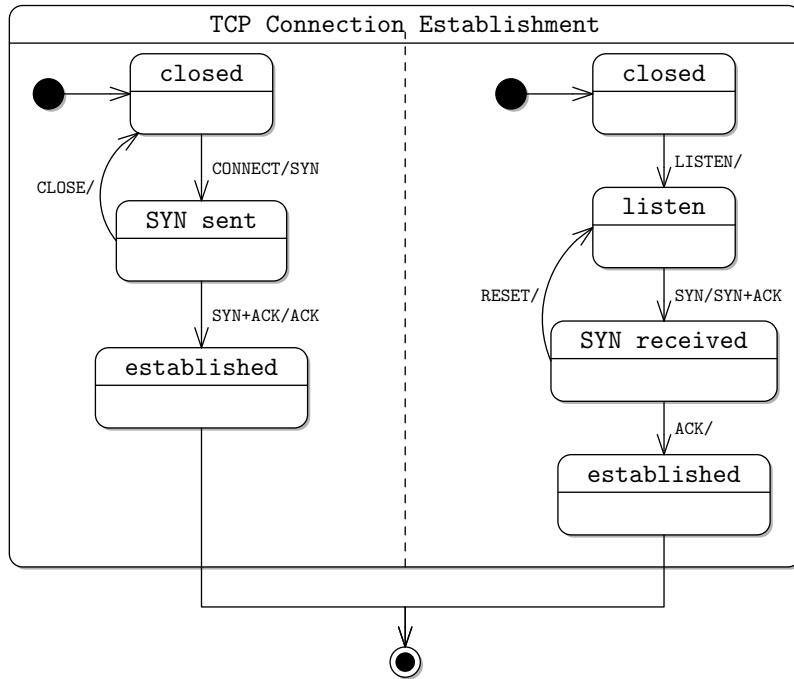


Figure 23.11: combined TCPClient and TCPServer state machine diagram for the TCP connection establishment model

Both substates start in their respective closed states. The client will transition to the SYN sent state when a CONNECT event is received, causing it to broadcast a SYN event. However, this SYN event will not be accepted unless the TCPServer is in the listen state. If this is the case, the TCPServer will accept the SYN event and broadcast the SYN+ACK event. In turn, the TCPClient receives the SYN+ACK and broadcasts a final ACK as it transitions to the established. The TCPServer receives the ACK and also transitions to the established state. The TCP Connection Establishment state then terminates.

### 23.3.2 Conditions on Transitions

Other features of the state machine diagrams notations (736)

You may have noticed a couple of problems with the DVDTray example state machine diagram in Figure 23.8:

- the state machine diagram doesn't allow the tray to be either opening or closing, instead, transitions are assumed to be instantaneous; and
- if the power to the DVD player is switched off, then the tray may be left in an open state. However, the model assumes that the DVD tray will always start in the closed state.

Persisting state, conditions and internal actions (737)

There are several extensions to the State Machine Diagrams notation that can be used to improve the model, illustrated in Figure 23.12.

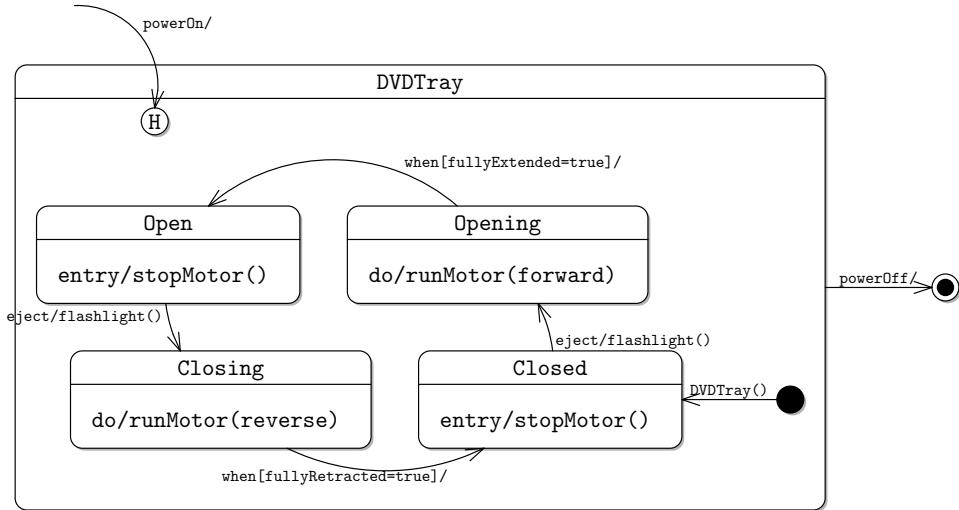


Figure 23.12: an extended DVDTray state machine diagram

The figure illustrates the use of conditions on state machine diagram transitions. Conditions are useful for causing state transitions as a result of internal events, rather than externally received messages or events. Conditions can be used on a transition without an accompanying event. Alternatively, conditions can be used in combination with events, in which case, the condition must hold *and* the event must occur when the transition happens. This can be useful when a state may transition to two different states as a result of the same event, depending on some other condition in the system.

The full label for a transition is as follows:

`<event>[<condition>]/<action>`

Conditions on transitions (738)

In figure 23.12, two extra states **Opening** and **Closing** have been introduced into the state chart to capture the intermediate state for the tray between being fully closed and fully open. The DVDTray state will transition from **Open** to **Closing** when the **eject** event is received. However, the state will not transition to the **Closed** state until the tray is fully retracted, i.e. when the **fullyRetracted ==true** condition holds. Similarly, there is an intermediate state **Opening** which holds after the **eject** event from a **Closed** state, until the **fullyExtended==true** condition is satisfied.

### 23.3.3 Internal Actions

Figure 23.12 also illustrates the use of internal actions on state machine diagrams. Internal actions are useful to indicate the effects that transitioning through a state can have without drawing out a complete state machine diagram. There are several standard flags for internal actions to be invoked at different stages in a transition through a state:

Internal actions (739)

- `entry<action>` as state is entered following a transition;
- `exit/<action>` before a state is departed due to a transition; and
- `do/<action>` while a state is active.

The figure illustrates the use of `entry` and `do` actions in the sub-states of the DVDTray state machine diagram. Whenever the tray enters the Open or Closed states the motor is stopped to prevent damage from occurring. Similarly, while the tray is in the Closing and Opening states, the motor is run either in reverse or forward directions.

### 23.3.4 State Persistence

Finally, figure 23.12 also shows how the sub-state of a state machine diagram can be saved and restored between exits and entrances back into the state, using a *history* state. History states are denoted using a circle labelled `H`. If a transition occurs to a history state, then the state of the surrounding state machine diagram is restored to the state when it was last exited.

In the figure, whenever the power is turned on, signalled by the `powerOn` event, the DVDTray state machine diagram is entered in the history state. Consequently whatever state the DVDTray was in when it was exited (due to a `powerOff` event) is restored, and the state machine diagram operation continues as before.

## 23.4 Using State Machine Diagrams in the Design Process

State Machine Diagrams are integrated into the UML and can be used to refine the details of a system's internal design based on previously developed UML models such as use cases, domain class models and sequence diagrams. The use of state machine diagrams for refining a design will be illustrated using the Book class from the branch library system requirements specification, developed in Chapters 11 and 12.

Part of the branch library system domain model class diagram concerning copies of books is illustrated in Figure 23.13. Recall that the library may hold many copies of the same book (25 copies of Sommerville's *Software Engineering* 9th Edition, for example). The abstraction-occurrence design pattern was applied to books, so that a `BookCopy` represents a real world book (the occurrence), and `Book` represents the attributes in common of many different copies of the same book (the abstraction).

Book related use cases (740)

Many of the use cases in the branch library system might affect the state of a copy of a book, as illustrated in Figure 23.14. These use cases encompass the

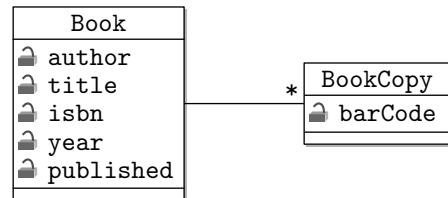


Figure 23.13: domain model for Book and BookCopy

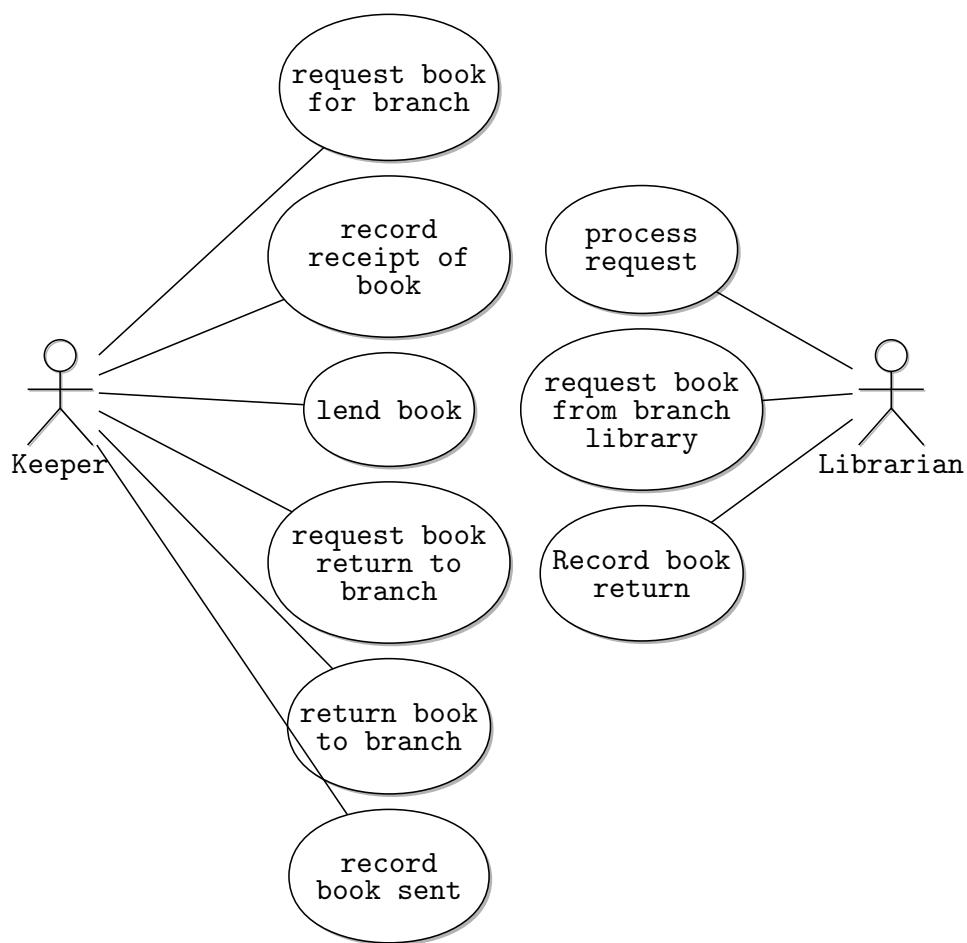


Figure 23.14: Book use cases

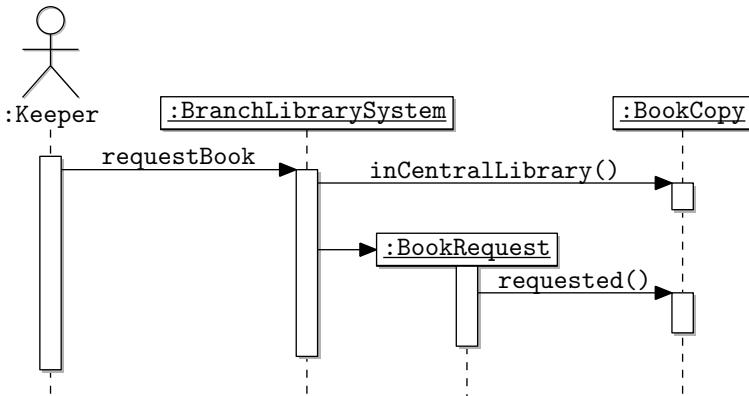


Figure 23.15: a sequence diagram for the request book **for** branch use case

Book related use cases (741)

process request sequence diagram (742)

different states a BookCopy may be in as it is moved from the central library to a keeper in the branch library, lent out to borrowers and so on.

The next step is to investigate what effect the invocation of a use case has on the internal state of a book. It is assumed that every book copy is initially held by the central library. If a book is available, then a Keeper may make a request for a book to be moved to the branch library. The sequence of interactions caused a request for a book is shown in Figure 23.15.

When a Keeper makes a request, the BranchLibrarySystem checks whether the book is available in the central library. If the book is available, then a new BookRequest instance is created for the specified book. The constructor for the BookRequest then calls the requested() method on the BookCopy object.

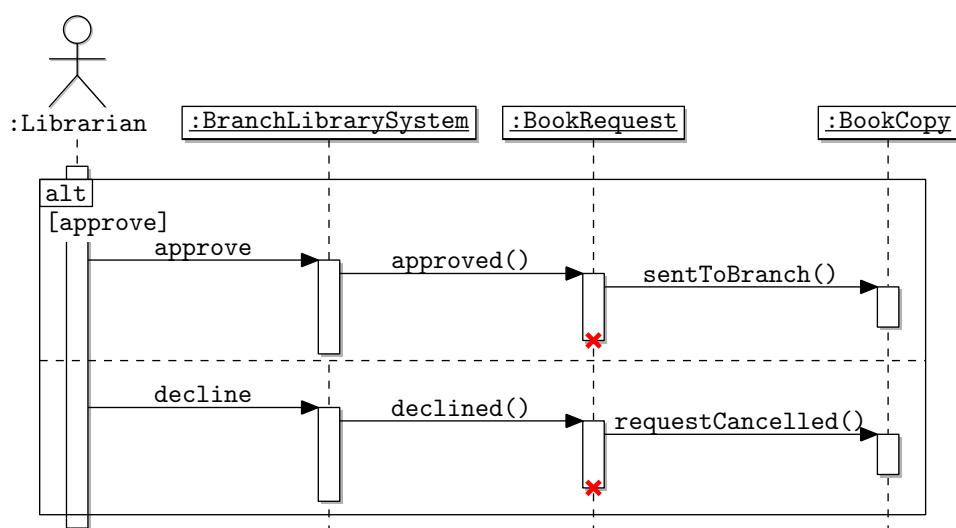


Figure 23.16: a sequence diagram for the process request use case

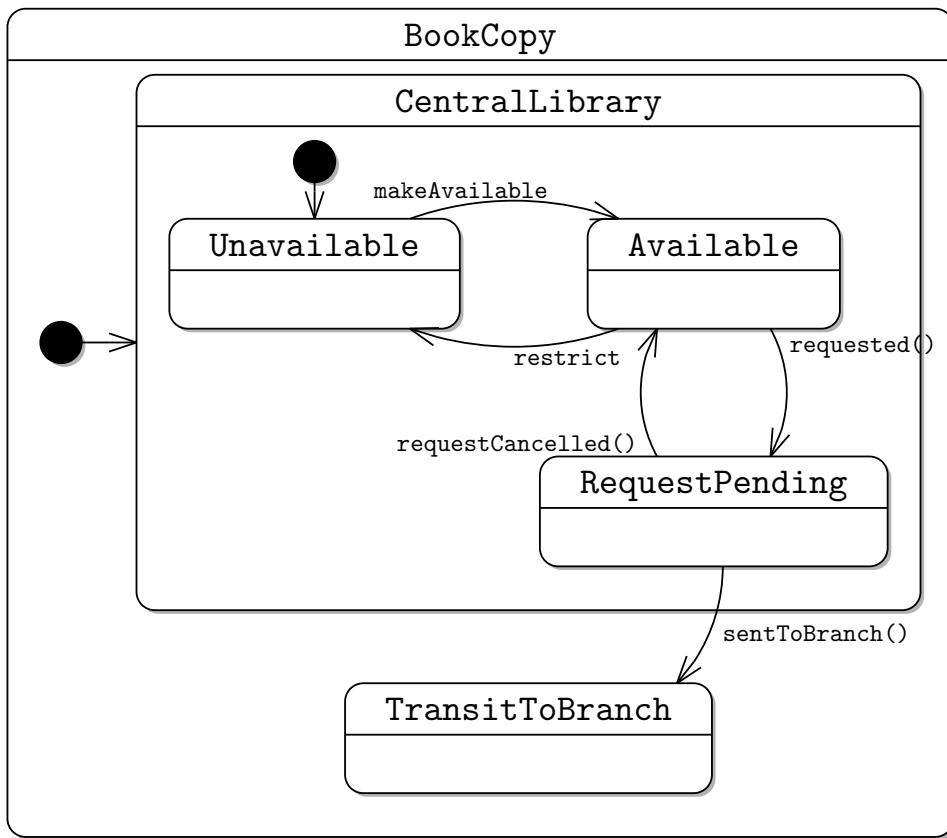


Figure 23.17: an initial state machine diagram for BookCopy

Figure 23.16 illustrates a sequence diagram for the process request use case, in which a CentralLibrarian chooses whether to approve or decline a request for a copy of a book to be sent to a branch. In either case, the action destroys the BookRequest object, because the request process has been completed. If the book request is approved, then the BookCopy is notified that the book has been sent to the branch, otherwise the pending request is cancelled.

At this point, it is possible to begin the state machine diagram for the BookCopy as illustrated in Figure 23.17. The state machine diagram illustrates the necessary states that we have established for the BookCopy.

When a copy of a book is in the central library, it is either Available or Unavailable for transfer to the branch library. Notice that we do not model the reason why the book is unavailable in the central library. This may be because it is on loan to a central library user, or because it is part of a special collection that can never be transferred or lent out of the central library.

When a request is made on an Available BookCopy using the `requested()` method, the BookCopy transitions to the RequestPending state. Notice that this means that another request cannot be made for the book until the BookCopy

Initial state machine  
diagram for the  
BookCopy (743)

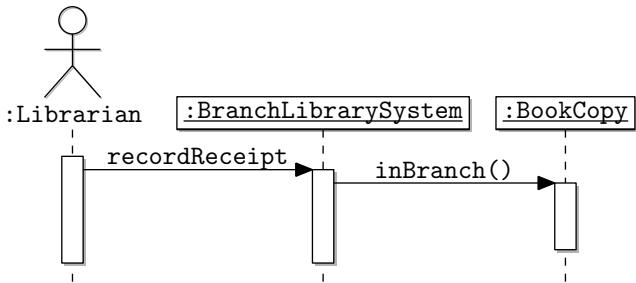


Figure 23.18: the record receipt of book sequence diagram

returns to the Available state. This may occur when a request is cancelled, for example.

The next stage is to consider the states that the book enters when it reaches the branch library. Figure 23.18 shows the sequence of operations that are invoked when a Keeper invokes the record receipt of book use case.

This and other sequence diagrams can be used to discover other operations that alter the state of the BookCopy, giving the final state machine diagram shown in Figure 23.19.

The figure shows that the BookCopy state has been extended to include two more substates, BranchLibrary, for when the BookCopy is somewhere in the branch library, and !TransitToCentral, for the intermediate situation when a BookCopy has been returned by a Keeper to the central library, but has not yet been received.

Of these two substates, the BranchLibrary is the more complex, consisting of two concurrent groups of substates. The group on the left handles the state transition which occurs when the central library requests that the BookCopy be returned from the branch. The group on the right handles the transitions which occur when a BookCopy is lent within a branch.

The initial state for this lending group is WithKeeper, which is active when a Keeper records that they have received the BookCopy from the central library. The Keeper can lend the book to a Borrower at any time, provided that the book is not in the CentralLibraryRequested state. The book can also be returned to the WithKeeper from the OnLoan state, caused by a returned() event. If the BookCopy is in the OnLoan state, it can also be in a further substate, indicating that the Keeper has requested that it be returned.

There are two possible exits from the BranchLibrary state. In the normal case, a transition occurs from WithKeeper to TransitToCentral when the BookCopy's Keeper records that the book has been returned (note this complements the RequestPending → TransitToBranch transition). Then, when the BookCopy arrives at the central library, it transitions to the Available state. Alternatively, a BookCopy in the BranchLibrary state may transition directly to the Available state. This may occur if the BookCopy arrives unexpectedly at

record receipt of  
book sequence  
diagram (744)  
Final BookCopy state  
machine diagram  
(745)

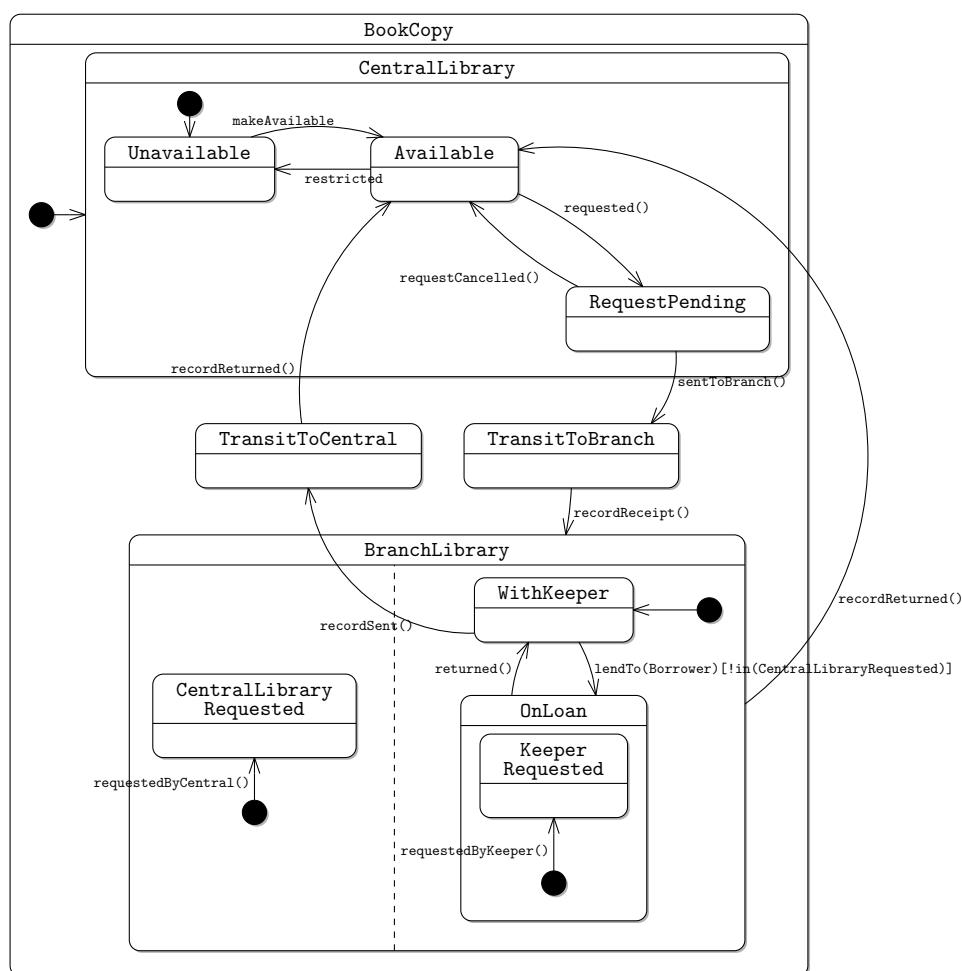


Figure 23.19: the final BookCopy state machine diagram

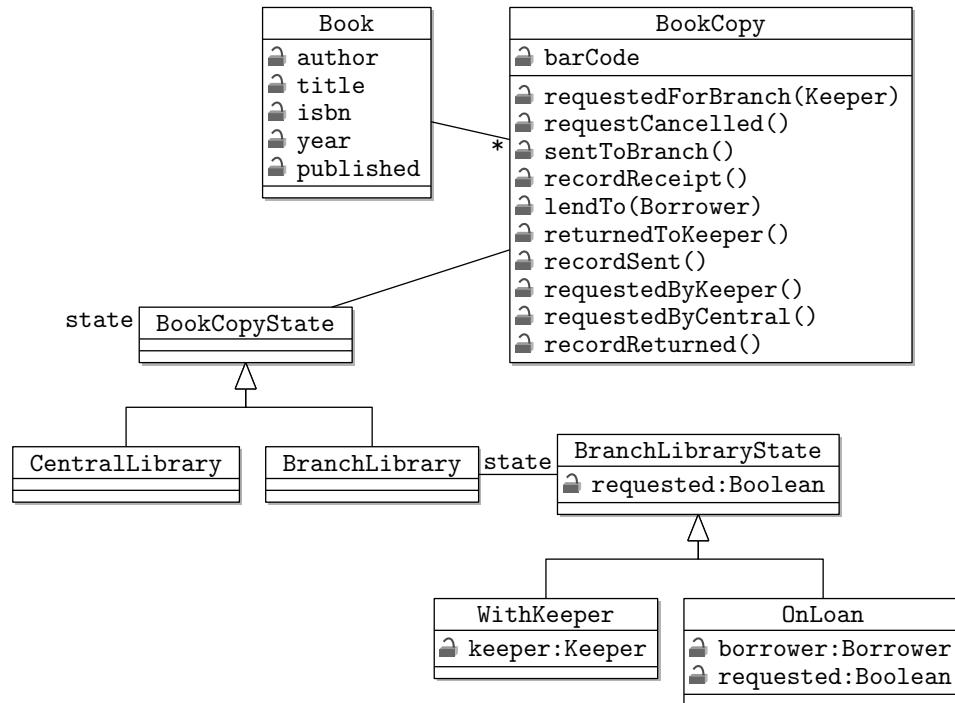


Figure 23.20: Revised BookCopy class incorporating state behaviour

the central library, and the Librarian checks the BookCopy in directly.

The completion of the state machine diagram means that additional attributes for storing state, and operations for altering state can be added to the BookCopy class, extending and refining the system design. Figure 23.20 illustrates the addition of methods identified on sequence diagrams and on state machine diagrams to the BookCopy class.

#### Revised BookCopy class (746)

A new abstract class, BookCopyState has been added to the model to represent the different states that a book may be in. Two of the concrete sub-classes, CentralLibrary and BranchLibrary are also shown, representing the specific states that BookState can be in. The BranchLibrary is also shown to have an association to another abstract class representing the substates for a BookCopy in the BranchLibrary state. These substates include WithKeeper and OnLoan. State specific attributes, such as the Keeper of a BookCopy in the BranchLibrary state are assigned to the class representing that state.

Notice that the substate of OnLoan, whether the BookCopy has been requested back by the Keeper has been denoted as a Boolean attribute. This is a convenience, because the OnLoan state only has two substates. A similar approach is taken for the requested attribute of the BranchLibraryState denoting whether a BookCopy has been requested back by the central library.

## Summary

State Machine Diagrams are a popular specification technique, providing a more convenient and intuitive notation than mathematical modelling languages for reactive systems such as CSP. The extensions over state transition networks can help reduce the complexity of a state machine diagram model considerably, by supporting encapsulation of internal transitions.

However, the very flexibility of state machine diagrams has the potential to introduce considerable complexity into state models:

- the use of unrestricted broadcast to facilitate concurrency means that every state in a model will witness broadcast events, whether they are pertinent to that state's potential transitions or not. This means that every state is *coupled* to every other state that broadcasts events;
- broadcast events at one level of abstraction in a model can cause a transition at any other level of abstraction; and
- transitions may occur between directly between states at different levels of abstraction, rather like a GOTO like statements in imperative programming languages, famously considered harmful by Dijkstra [Dijkstra, 1968]. Figure 23.9 illustrates this problem, showing a direct jump from B to C, even though C is nested in D. Permitting this form of transition is rather like allowing a method call to specify where in the method body the call should start.

Disadvantages of the state machine diagram notation (747)

Figure 23.21 illustrates a state machine diagram model of the display for a digital watch, re-drawn from the model presented by Harel [1988, pp.525]. The figure illustrates a relatively small application, and yet the number of states, partitions and transitions makes the overall behaviour difficult to comprehend. In addition, transitions in one part of the state diagram can flag events which causes changes elsewhere.

Harel's watch (748)

Consequently, state charts can still be complex to model and comprehend, particularly as the number of states increases. Many other formal state based notations, such as CSP utilise the notion of *communication channels*, so that a state can only witness events on channels on which it is listening. This approach partitions the communication networks within the state model.

There are a number of automated tools available which support the development of executable models of systems based on state machine diagrams. These support model driven development and model checking techniques.

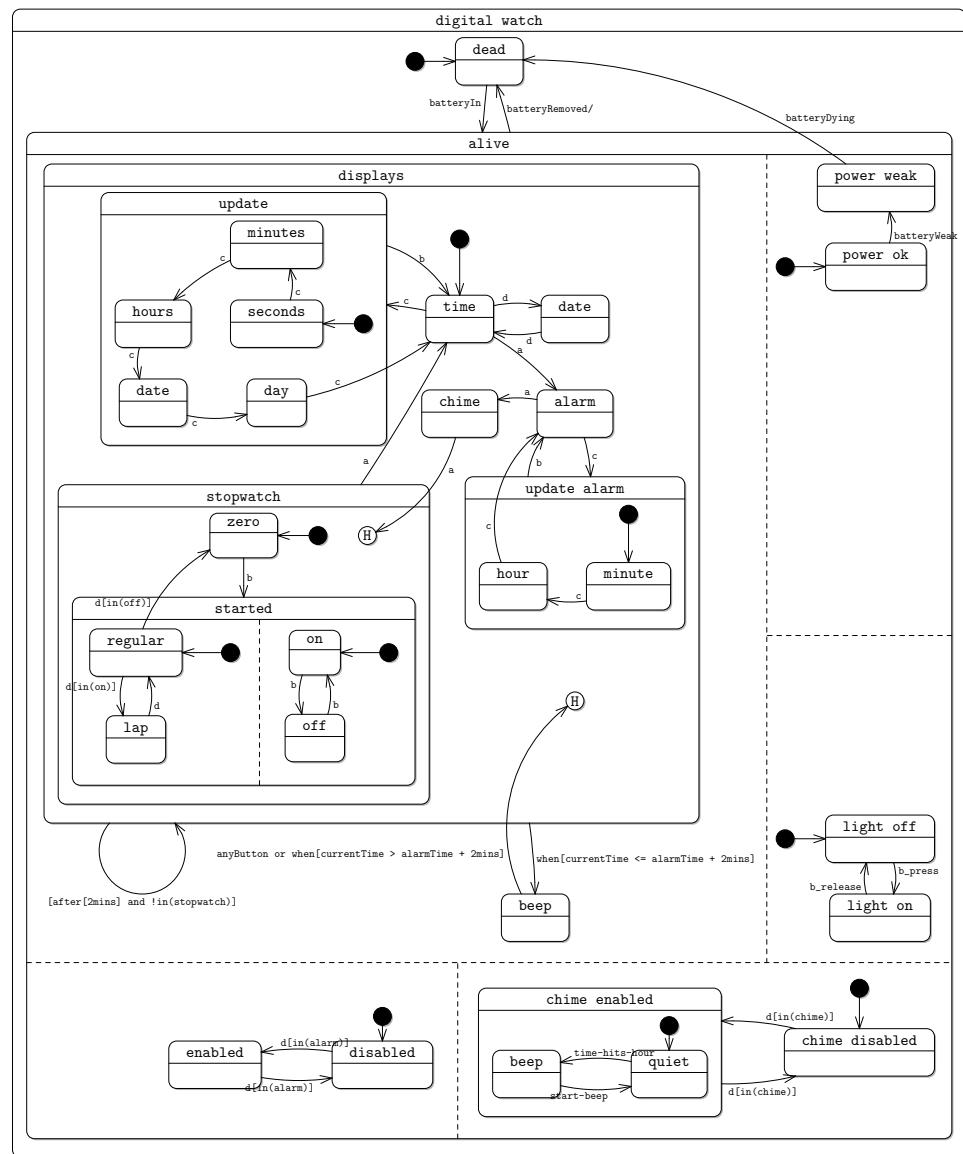


Figure 23.21: Harel's watch, re-drawn from Harel [1988].

## 23.5 Exercises

1. Consider a system to handle room bookings with a class `Booking`, specified as shown in Figure 23.22.

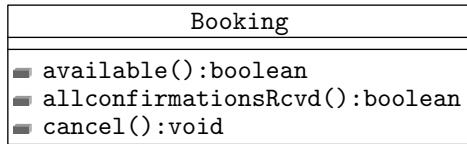


Figure 23.22: the `Booking` class

Once created, a `Booking` enters an unconfirmed state. This causes an invocation of the `available()` method. If the room is available (indicated by a **true** result from the `available()` method), the booking becomes tentative otherwise it becomes pending.

In the case of bookings in the pending state, availability is checked periodically. Thereafter, if the room is available, the booking moves to tentative. In the case of tentative bookings, the meeting is announced to participants.

Once the last participant has confirmed that they have received the announcement (indicated by a **true** result from the `allconfirmationsRcvd()` method), the meeting moves to the booked state. A booking can be cancelled at any time by invoking the `cancel()` method, unless it is already in the cancelled state; invocation of the `cancel()` method causes the object to move to a cancelled state.

- (a) Use a state machine diagram to show the lifecycle of `Booking`.
  - (b) Draw a state transition table to implement this state machine diagram.
  - (c) What other behaviour do you believe this object might usefully have?
2. Consider an order processing system with a class `Order`, specified as shown in Figure 23.23.



Figure 23.23: the `Booking` class

Orders for items in the company's stock are received from customers. Once created, an order must be both paid (indicated by a `paid()` message) and there must be stock available (indicated by a `stockAvailable()` message) before it can be despatched. However, the ordering of receipt of these messages is not significant. The `Order` can also be cancelled at any time. Once despatched or cancelled the `Order` object is destroyed.

Use a state machine diagram to show the lifecycle of an instance of `Order`. Try solutions with and without partitioning states.

3. Consider the two state machine diagrams in Figures 23.10 and 23.11. How would you group the two state machine diagrams together so that when the `TCPServer` is in the `SYN received` state, it can be sent a `RESET` event if the `count` variable reaches a given value representing a timeout?

## 23.6 Workshop: Using Threads

threads

Sometimes it is useful to have more than one thing happening at a time in an application. For example, a user interface must be rendered to the screen while simultaneously tracking user interactions. This tutorial describes the use of multiple program *threads* in Java. Threads are separate program sequences in a Java application, each with their own method call stack. However, all threads in a single application share the same memory space, allowing them to interact with each other.

Motivation (752)

### 23.6.1 Implementation in Java

Threads in Java are represented by the Thread class, which implements the Runnable interface. Figure 23.24 illustrates the thread class structure, showing the constructors and methods of the Thread class. A Thread can be constructed with a (potentially non-unique) name which is used to identify the thread in debugging tools.

In addition, a thread can be constructed using an instance of Runnable. A Runnable instance defines the behaviour of an application thread in the body of the run() method. The run() method is invoked when the thread is started, using the start() method. When the run() method completes, the thread terminates.

There are three different styles of implementing threads in Java:

Threads in Java (753)

- extend the Thread class;
- implement the Runnable interface; or
- anonymously extend the Thread class.

In each case the tasks for the developer are the same:

- implement the Runnable.run() operation
- invoke the Thread.start() method

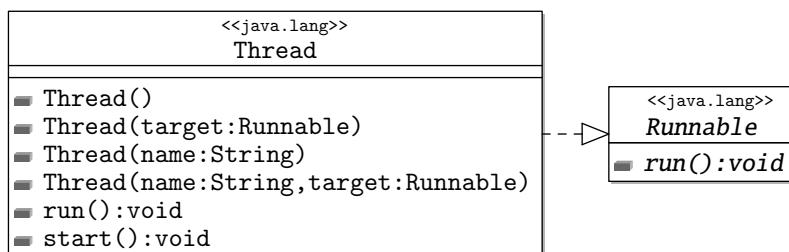


Figure 23.24: thread architecture in Java

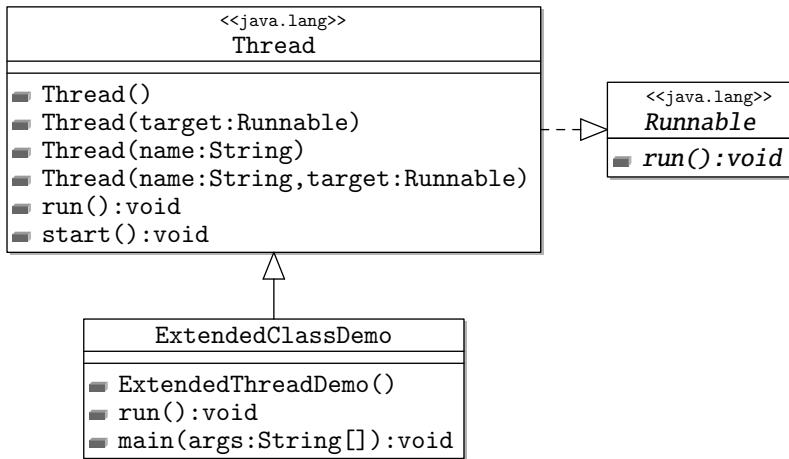


Figure 23.25: extending the thread class

This section will describe each of these approaches in turn.

### Extending the Thread Class

Figure 23.25 illustrates how to implement threads by extending the `Thread` class. The `ExtendedThreadDemo` class overrides the `run()` method in the `Thread` class to provide some useful behaviour for the thread. The method in the `Thread` class is implemented, but has an empty body, so the thread terminates immediately.

The Java code below shows the implementation of the `ExtendedThreadDemo` class.

```

public class ExtendedThreadDemo extends Thread {

    public static void main(String[] args) {
        args = new String[]{"bob", "sue"};
        for (String arg : args) {
            Thread t = new ExtendedThreadDemo(arg);
            t.start();
        }
    }

    /**
     * Provides access to the java.lang.Thread(String) super
     * constructor.
     *
     * @param name the name of the thread
     */
    public ExtendedThreadDemo(String name) {
        super(name);
    }
}

```

```

/**
 * Override the run() method in Thread to do something
 * interesting.
 */
@Override
public void run() {
    Random r = new Random();
    while (true) {
        int sleepTime = r.nextInt(5000);
        try {
            Thread.sleep(sleepTime);
        } catch (InterruptedException e) {
            System.err
                .println("Leave me alone, I'm sleeping!");
        }
        System.out
            .println("Hello world, I'm a thread called ["
                + getName() + "] "
                + "and I've been asleep for ["
                + sleepTime
                + "]ms. Thanks for waking me up!");
    }
}

```

The main() method constructs two ExtendedThreadDemo instances with the names "bob" and "sue". When each thread is initialized it is immediately started.

The run() method contains a **while(true)** loop to ensure that the method never terminates. At the beginning of each iteration, the thread is put to sleep for a random number of milli-seconds (between 0 and 5000) using the static sleep() method. This statement is wrapped in a **try-catch** block in case the thread is interrupted. This is necessary because it is possible that another thread will attempt to cancel the thread's sleep. As can be seen in the **catch** block, the thread won't be particularly happy if this occurs.

Assuming that the thread is not interrupted, a message is printed out and the loop repeats. A sample output from the program is shown below (the ordering will change each time the program is run).

Sample output (755)

```

Hello world, I'm a thread called [sue] and I've been
    asleep for [806]ms. Thanks for waking me up!
Hello world, I'm a thread called [sue] and I've been
    asleep for [1082]ms. Thanks for waking me up!
Hello world, I'm a thread called [sue] and I've been
    asleep for [1965]ms. Thanks for waking me up!
Hello world, I'm a thread called [bob] and I've been
    asleep for [4453]ms. Thanks for waking me up!
Hello world, I'm a thread called [sue] and I've been
    asleep for [869]ms. Thanks for waking me up!

```

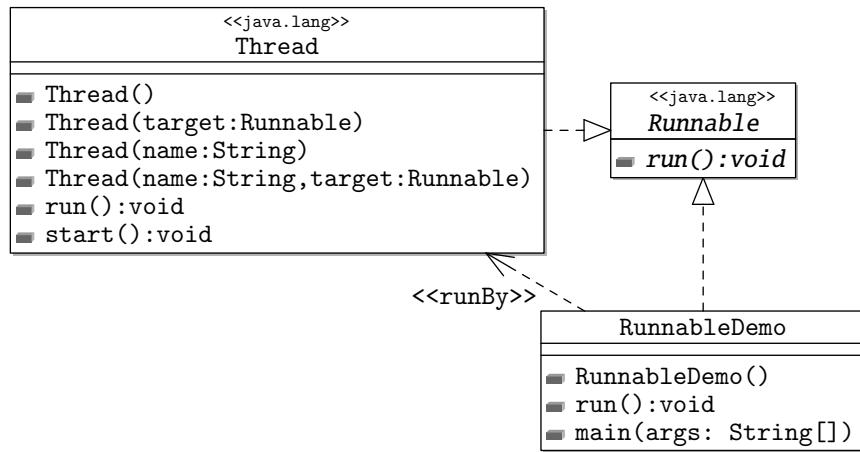


Figure 23.26: implementing the `Runnable` interface

```
Hello world, I'm a thread called [bob] and I've been
asleep for [448]ms. Thanks for waking me up!
```

### Implementing the `Runnable` Interface

The second approach to implementing threads is to implement the `Runnable` interface and then pass an instance of the implementation to an instance of `Thread`. This approach is useful when your class is already part of another inheritance hierarchy and you can't directly extend the `Thread` class. Figure 23.26 shows this arrangement in a class diagram.

Implementing  
Runnable -  
`RunnableDemo` (756)

The Java code below shows how to use this approach with the same example behaviour given above.

```

public class RunnableDemo implements Runnable {
    public static void main(String[] args) {
        for (String arg : args) {
            Runnable r = new RunnableDemo();
            Thread t = new Thread(r, arg);
            t.start();
        }
    }

    /**
     * Implement the run() method in Runnable to do
     * something
     * interesting.
     */
    @Override
    public void run() {
        Random r = new Random();
        while (true) {
    
```

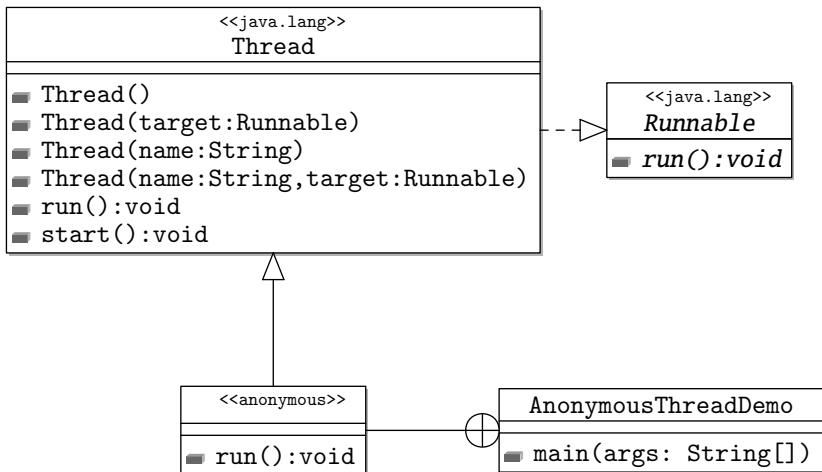


Figure 23.27: creating an anonymous thread class

```

int sleepTime = r.nextInt(5000);
try {
    Thread.sleep(sleepTime);
} catch (InterruptedException e) {
    System.err
        .println("Leave me alone, I'm sleeping!");
}
System.out
    .println("Hello world, I'm a thread called ["
        + Thread.currentThread().getName()
        + "] and I've been asleep for ["
        + sleepTime
        + "]ms. Thanks for waking me up!");
}
}
}
}

```

Notice that the implementation is almost identical. The only difference is in the `main()` method, where instances of `RunnableDemo` are created and assigned to a `Runnable` variable. These instances are then passed as an argument to the `Thread` class constructor, and the thread is started as before.

### Creating Anonymous Threads

Figure 23.27 illustrates the third way in which threads can be implemented using an anonymous inner class. This approach is useful for 'simple' threads with short behaviours that are dependent on the attributes of their host class.

The Java code below shows how to implement an anonymous thread with the same example behaviour again.

Anonymous Thread -  
`AnonymousThreadDemo`  
(757)  
`AnonymousThreadDemo`  
(758)

```

public static void main(String[] args) {
    for (String arg : args) {
        Thread t = new Thread(arg){
            private Random r = new Random();

            @Override
            public void run() {
                while (true) {
                    int sleepTime = r.nextInt(5000);
                    try {
                        Thread.sleep(sleepTime);
                    } catch (InterruptedException e) {
                        System.err
                            .println("Leave me alone, I'm sleeping!");
                    }
                    System.out
                        .println("Hello world, I'm a thread called ["
                            + getName()
                            + "] "
                            + "and I've been asleep for ["
                            + sleepTime
                            + "]ms. Thanks for waking me up!");
                }
            }
        };
        t.start();
    }
}

```

This time, the `run()` method is defined within the body of the anonymous thread class, when the `Thread` constructor is called.

### 23.6.2 Timing and Coordination Defects

So far, the example threaded programs have used simple examples that have not accessed shared resources. However, threads must often be able to communicate via shared resources in order to perform a useful task. When this occurs it is necessary to mediate access to the resources in a controlled way. When access to shared resources is not mediated correctly, timing and coordination defects can be introduced to an application.

The code below illustrates an example consumer-producer problem for multiple threads. The first class is a resource, which has a single method for setting the `value` attribute. The attribute can only be set to be one more or one less than the current value, and should not be set to greater than or less than the specified maximum or minimum (it is a bad implementation to make the point!). The method contains some crude debugging code to report what changes are made to the `value`.

Coordination defects  
example (759)

```

class Resource {
    public final int max = 100000;
    public final int min = 0;

    public int value = 50000;

    public void setValue(int value) {
        System.out.println();
        int currentValue = this.value;
        int diff = Math.abs(value - currentValue);
        String tName= "["+Thread.currentThread().getName()+"] ";

        System.out.println(tName + ": Changing [" +
            + currentValue + "] to [" + value + "] .");

        if (diff != 1){
            System.out.println(tName + ": oops! ["+diff+"] .");
            System.out.flush();
            System.exit(0);
        } else this.value = value;
    }
}

```

The next two classes are the resource producer and consumer threads respectively. Both classes have a constructor for assigning the shared resource. The Producer class will attempt to increase the amount of resources available up to the maximum. Each time the **while** loop in the `run()` method executes, the current value of the resource is obtained and cached. If the value of the resource is less than `max`, the value is set to be one plus the old value. The Consumer class decrements the amount of resource available in the same way.

```

class Producer
    implements Runnable {

    private Resource r;

    public Producer(Resource r) {
        this.r = r;
    }

    public void run() {
        while (true) {
            int value = r.value;
            if (value < r.max)
                r.setValue(value + 1);
            else
                r.setValue(value);
        }
    }
}

```

```

class Consumer
implements Runnable {

    private Resource r;

    public Consumer(Resource r) {
        this.r = r;
    }

    public void run() {
        while (true) {
            int value = r.value;
            if (value > r.min)
                r.setValue(value - 1);
            else
                r.setValue(value);
        }
    }
}

```

The program is set up with a single resource shared by one consumer and one producer. In principle, the value of the resource should never go above `max`, or below `min`, and should never change by more than one. However, the example output shown below indicates that this isn't always the case.

Example output (760)

```

[C]: Changing [50000] to [49999].
[C]: Changing [49999] to [49998].
[C]: Changing [49998] to [49997].
[P]: Changing [49997] to [50001].
[P]: oops! [4].

```

So how has this happened? The Producer thread read the value of the resource and was then paused by the virtual machine to allow the Consumer thread some computation time. The Consumer thread then completed three loops, decrementing the value of the resource by one each time. The virtual machine then paused the Consumer thread and resumed the Producer. The Producer then carried on its interrupted loop and attempted to set the resource value to 50001 (one more than what it believed was the current value). This caused the resource to detect a problem and halt.

### 23.6.3 Synchronization

A solution to this problem is to *synchronize* the threads so that there are portions of the program that can only be entered by one thread at a time. The **synchronized** keyword in Java can be used in conjunction with a shared resource

to allow two or more threads to synchronize. The Java code below shows how this can be done for the consumer producer problem.

Synchronization (761)

```
class SyncProducer
implements Runnable{
private Resource r;

public SyncProducer(
    Resource r){
    this.r = r;
}

public void run (){
    while(true){
        synchronized(r){
            int value = r.value;
            if (value < r.max)
                r.setValue(value+1);
        }
    }
}

class SyncConsumer
implements Runnable{
private Resource r;

public SyncConsumer(
    Resource r){
    this.r = r;
}

public void run (){
    while(true){
        synchronized(r){
            int value = r.value;
            if (value > r.min)
                r.setValue(value-1);
        }
    }
}
```

The two while loops contain a **synchronized** statement whose scope encompasses all the statements of the method that must be executed together. This means all three statements which alter the state of the value attribute are treated as *atomic*. The two threads synchronize on the r shared resource attribute. When one thread reaches the **synchronized** statement, it waits until it can obtain a lock on the shared resource. Once the lock is obtained, the body of the **synchronized** block is executed and the lock is released. Since, only

one thread can obtain a lock on a resource at any one time, this mechanism guarantees that a thread won't be prevented from executing all its synchronized statements together.

### 23.6.4 Synchronization Defects

Unfortunately, the use of synchronization to resolve timing defects can also introduce synchronization defects. This occurs when two or more threads need exclusive access to more than one resource. There are two types of synchronization defects:

Deadlock and Livelock

(762)

- deadlock: two or more threads cannot proceed because they are blocking each other; and
- livelock: one or more threads are trapped in the same cycle of processes.

Figure 23.28 illustrates an example of deadlock using a sequence diagram (see Section 15.2). In Figure 23.28(a) a thread obtains a lock first on resource *r* and then on *p* to perform some work. Deadlock does not occur because only one thread is involved in the sequence. In 23.28(b) two threads with the same behaviour simultaneously attempt to lock resources *r* and *p*. The thread on the left starts first, and gains a lock on *r*. However, while this is happening,

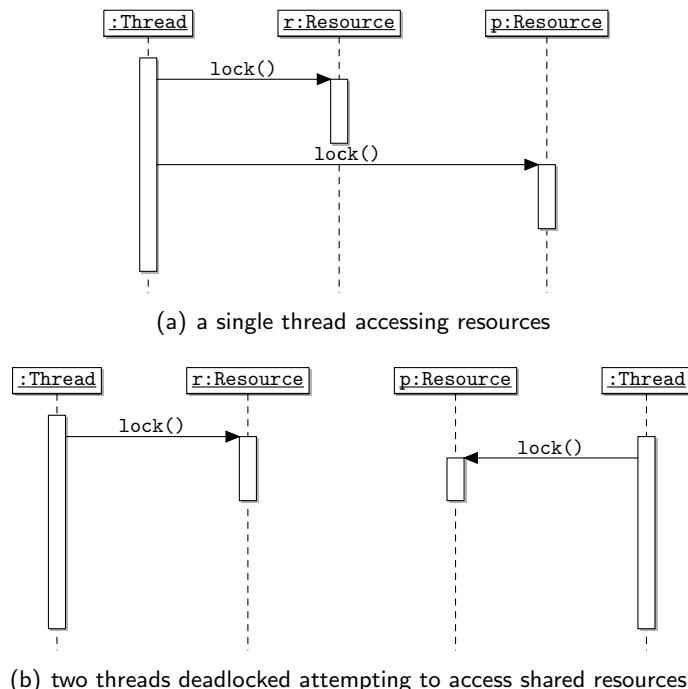


Figure 23.28: deadlock defects

the thread on the right is unable to obtain a lock on resource `r`, so instead attempts first to obtain a lock on `p`. Unfortunately, this now means that the left hand thread is blocked waiting for a lock on resource `p` and the thread on the right is blocked waiting for a lock on `r`, both held by the other thread. As a consequence, neither thread will be able to proceed any further and are deadlocked.

The code below shows how this scenario can be implemented in Java using the `Lock` interface and `ReentrantLock` class.

```
public class DeadLock extends Thread {

    private final Lock r;
    private final Lock p;

    public DeadLock(Lock r, Lock p, String name) {
        super(name);
        this.r = r;
        this.p = p;
    }

    public void run() {
        String tName = Thread.currentThread().getName();

        while (true) {
            // get the locks on the resources
            if (r.tryLock()) {
                System.out.println("[ "+tName+" ] got r trying p ...");
                p.lock();
            } else {
                p.lock();
                System.out.println("[ "+tName+" ] got p trying r...");
                r.lock();
            }
            // do some processing
            System.out.println("[ "+tName+" ] ... has acquired r and
                p.");
            // release the resources
            r.unlock();
            p.unlock();
        }
    }

    public static void main(String[] args) {
        Lock r = new ReentrantLock();
        Lock p = new ReentrantLock();
        DeadLock d11 = new DeadLock(r, p, "d11");
        DeadLock d12 = new DeadLock(r, p, "d12");
        d11.start();
    }
}
```

```
    d12.start();  
}  
}
```

The resource is modelled as a lock. Two threads of the class DeadLock are constructed and started in the `main()` method. The `DeadLock.run()` method uses the `tryLock()` method to test whether a lock is obtainable on the `r` resource. If the lock on `r` is obtained, the thread then blocks until it has a lock on `p`. If the lock cannot be obtained immediately, the locks are tried in reverse order. Once the locks are both obtained, the thread reports this and then releases both locks.

Example output (763)

The output below shows the consequences of this implementation.

```
[d11] got r trying p ...  
[d11] ... has acquired r and p.  
[d11] got r trying p ...  
[d11] ... has acquired r and p.  
[d11] got r trying p ...  
[d11] ... has acquired r and p.  
[d11] got r trying p ...  
[d11] ... has acquired r and p.  
[d11] got r trying p ...  
[d12] got p trying r ...
```

For the first four loops, the `d11` is able to successfully obtain both locks, presumably because `d12` is still being started. However, once both threads attempt to lock the resources simultaneously, both threads become dead locked trying to obtain a lock on the missing resource.

Deadlocks and livelocks are difficult defects to eliminate from programs. However, good design practice should be to minimise the amount of shared resources between threads to reduce the potential for both coordination and synchronization defects. Good design practice is also to isolate resource access in particular modules which mediate data access carefully.

### 23.6.5 Summary

Threads can be messy - use them with care!

## **Chapter 24**

# **Aspect Oriented Development**

### **Recommended Reading**

#### **24.1 Summary**

## **24.2 Exercises**

## Chapter 25

# Formal Methods in Software Engineering

### Recommended Reading

PLACE HOLDER FOR sommerville07software

PLACE HOLDER FOR wordsworth99getting

PLACE HOLDER FOR glass04mystery

PLACE HOLDER FOR barnes06engineering

PLACE HOLDER FOR bowen95ten

PLACE HOLDER FOR mackenzie01mechanizing

PLACE HOLDER FOR warmer03ocl

Very good tutorials on OCL at:

- <http://atlanmod.emn.fr/atldemo/ocltutorial/>
- <http://www.csci.csusb.edu/dick/samples/ocl.html>

Wordsworth reviews progress made in the use of formal methods in the period between 1985 and 1999 and suggests reasons for lack of widespread use of formal methods. The paper proposes ways to 'infiltrate' formal methods into software development. The Glass paper responds to this by suggesting that there may be a problem with formal approaches to software development if they have to be infiltrated into work practices.

Barnes et al describes a sizable effort at formally specifying a security control system in Z. However, the report documents only limited uses of formal proofs concerning the correctness of the resulting design.

Mackenzie is a sociologist with an interest in the history of computing. The book is a history of the ‘tussle’ in software engineering, computing science and mathematics over the nature of ‘proof’ and its role in system development.

## 25.1 Introduction

Certain problem domains require higher assurance as to the correctness of a software system. Some types of problem domain demand high assurance systems are:

- *real time systems* are developed to monitor and effect changes in their environment. Software developed for real time systems must be guaranteed to respond to stimuli in a timely manner. Examples include Controller Area Network (CAN) bus controllers in automotives and auto-pilot systems in aircraft;
- *safety critical systems* are developed to operate in hostile environments, in which failure may result in serious harm to human users or damage to or even complete loss of the system. Examples include control systems on space craft (which are extremely difficult to repair once in operation) and nuclear power station protection systems; and
- *security systems* are developed to protect high value resources from malicious threats such as tampering or eaves-dropping. Examples include Automatic Teller Machines, voting systems and building access control systems. Alternatively, selected components of a system may be developed against security criteria. Examples include security kernels [Klein et al., 2009] and voting systems [Cansel et al., 2006, Storer and Lock, 2009].

Of course, many systems may need to be assured against multiple criteria. A fly-by-wire controller in aircraft, for example, is an example of both a real time and safety critical system. Recent research has demonstrated the security vulnerabilities of CAN based systems [Koscher et al., 2010].

High assurance problem domains are characterised by the need for a low defect rate, implying the need for:

- a precise description of a system’s characteristics for verification; and
- a rigorous method for assessing the system against the specification.

So far we have used a collection of semi-formal notations from UML for specifying and designing the features of a system. These include:

- natural language to give early use case descriptions and scenarios;

The need for formal specifications (766)

Approaches to specification (767)

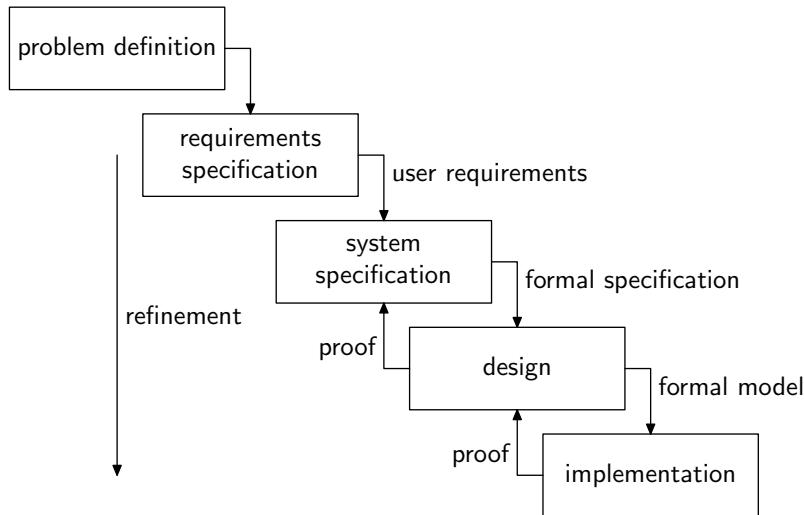


Figure 25.1: separation of elicitation and specification in the software process

- structured language as an intermediate step to pseudo code and graphical notations; and
- diagrams including use case, activity, sequence and class diagrams from UML.

In addition to these, mathematical notations can be used to give precise meaning to specifications, and can sometimes be directly translated into implementation code.

## 25.2 Formal Specification in Software Development

Figure 25.1 illustrates a waterfall software process containing a number of activities we are already familiar with from Chapters 3 and 4. However, an additional step has been introduced between requirements specification and system design.

In this revised process:

- the *requirements specification* describes the desired state of the world after a system is implemented, i.e. once the customer's problem has been solved;
- the *system specification* describes *what* the system should do to solve the customer's problem; and
- the *design* describes how the system satisfies the specification.

Using formal methods, each of these stages of software development occurs as a forward transformation of intermediate software products. Sometimes this

Re-visiting the  
Software Process  
(768)

process is described as top-down refinement because the process begins with an abstract specification to which greater detail is gradually added as design decisions are made.

Transformations may be performed manually by software developers or automatically using tools such as model generators or compilers. Rather than performing validation at the end of the waterfall, each intermediate transformation is verified by:

- inspections;
- testing and model checking; or
- mathematical proof techniques.

Both testing and mathematical proofs are approaches to verifying the correctness of the transformations between stages of the software development process. However, the large number of potential inputs to a software system mitigates against complete testing, and thus offering strong assurance of correctness via this method. Formal methods are intended to give stronger guarantees of the correctness of transformation by offering proof that one artifact is a faithful representation of another.

Given the potential advantage of using methods that formally provide for correct software, it is perhaps surprising that such approaches do not have a wider acceptance by software developers. Unfortunately, despite the attraction, there are inherent limits to the use of formal methods in software development.

*Accidental limitations* of formal methods refer to current obstacles to their use that may be mitigated by future developments, including:

- Skills
- Tools
- Scale and performance

*Intrinsic limitations* refer to the inherent characteristics of formal methods that can only be partially remedied by future progress in the area:

- The use of mathematical notations exacerbates the problem of requirements validation, because the specification is presented in a language that a customer may struggle to comprehend. As a consequence, achieving a shared understanding between customer and developer of what the system needs to do becomes harder and makes 'sign off' of the specification more difficult to achieve.

To mitigate this problem, formal specifications can be accompanied by natural language explanations of their meaning and intent. Meaning is a natural language description of what the specification says; intent is

the aspect of the problem that the specifier believes they are addressing. As with all documentation, however, there is a risk that the formal specification may evolve faster than the documentation;

- Similarly, formal methods do not address the problem of requirements validation (notice in Figure 25.1 that there is no reverse arrow from system specification to requirements specification). In one sense, formal methods shift the problem of verifying a design and implementation to one of verifying the specification with respect to the customer's needs. As a consequence, the developed system still needs to be validated through acceptance testing processes.
- The pre-dominance of re-usable software libraries makes the pure top down refinement approach of formal methods difficult to apply, particularly at the more detailed levels of design and implementation. As discussed in Section 4.1, specification is constrained by the design of pre-existing software infrastructure. Customer requirements may need to be adapted to fit within these constraints.
- The complexity and scale of a typical software program demands the use of automated tools to produce the transformations between stages of refinement and/or the corresponding proofs of correctness. Such tools are themselves complex software programs, subject to defects, so the assurance gained is only as good as that of the tool which produced it.

In summary, all of these limitations can mean significant costs for a software development project that adopts formal methods. Formal specifications should be used when they are appropriate, i.e. they are beneficial to the software development task.

### 25.3 Using Formal Specifications

There are two main approaches to formal specification:

- *process algebras* describe the legal behaviours of a system and the transitions between them over time. The formal language is used to specify the legal *traces* of sequences of events for a process. Process algebras usually need some representation of time, which can be either discrete or continuous; and
- *state algebras* are used to describe the legal states of a system and the transitions between them. The formal language is typically used to indicate the state maintained by the system and the state of the system before and after discrete operations.

<b>process algebras</b>	Hoare Logic Communicating Sequential Processes (CSP) and numerous dialects
<b>state algebras</b>	Vienna Development Method (VDM) Z (after Zermelo-Fraenkel Set Theory, with types) Object Constraint Language (OCL) B (refinement method)

Table 25.1: some formal method notations

Approaches to Formal Specification (771)  
State based algebraic specification (772)

Some formal methods, such as Circus [Woodcock and Cavalcanti, 2001], combine both process and state algebra notations. Table 25.1 lists some of the most popular formal method notations.

We will focus on state based algebras during this lecture. In general, state based algebraic specification proceeds by identifying the:

- high level components to be specified;
- internal state of the component that must be exposed in the specification;
- legal combinations of internal state values for the component. This is the set of conditions that must always be true about an implementation of the specification and is called the *state invariant*;
- signatures of operations on the component, including the inputs and outputs; and
- the pre and post conditions for each operation, i.e. what state the component must be in before the operation is invoked, and what state it must be in afterwards.

## 25.4 The Object Constraint Language

In principle, we could formally express the specification of an object oriented system by writing out all the possible legal states. We could, for example, draw UML object diagrams for every possible combination of object relationships and attribute values that we wish to allow in the system. However, the specification would quickly become extremely verbose for any non-trivial system. And indeed, would negate the need for a corresponding implementation!

A far better alternative is to express the legal combinations of relationships and attributes as constraints. A constraint is a boolean assertion about a system design that can be evaluated (to either true or false) on the real system when it is executed. If it can be demonstrated that all of the constraints are or (even

better) will always be satisfied by the implementation (by being evaluated to true), then the system implementation can be said to satisfy its specification.

We have already seen several examples of constraints in the UML diagrams developed in Chapters 11, 12 and 14:

Constraint techniques  
in UML (773)

- multiplicities on relationships restrict the number of instances of a class that can be involved in an association;
- guards can be used on activity and sequence diagrams to control the flow of events; and
- notes can be used to informally describe more complex constraints.

The Object Constraint Language (OCL) is an extension to the UML which can be used to formally specify the more complex constraints on an object oriented system. OCL is useful when it is not feasible or sensible to express constraints using multiplicities or by re-structuring the system.

OCL can be used with a number of UML diagram types, but for these lectures we will mainly look at using OCL for describing constraints about:

Constraints in OCL  
(774)

- the *invariant* legal states of relationships and attribute values on class diagrams. Invariants are assertions that must always be true about a system;
- the *pre-conditions* about the system which must be satisfied before operations are invoked; and
- the *post conditions* which must be satisfied when an operation terminates.

Note that this means we do not use OCL to alter the state of a running object oriented system in the way that we can with imperative languages like Java or C++. Instead, OCL is used to declare how the implementation *must* behave if it is to satisfy its specification.

### 25.4.1 Writing Constraints

OCL syntax uses the ASCII character set. This makes the notation more verbose than similar languages, such as Z, but also generally easier to interpret.

OCL *constraints* are constructed from smaller building blocks called *expressions*. Expressions are *evaluated* to simpler value types when a specification is checked. An OCL *constraint* is an expression that, when evaluated, results in a **Boolean** value of either **true** or **false**.

## Types and Values

Types in OCL include:

- **Boolean**: {true, false}
- **Integer**: ..., -1, 0, 1, ...
- **Real** : ..., -0.1, ..., 0.0, ..., 0.243, ...
- **String** : ..., "hello", "worRld", ...
- UML classes: Book, Borrower...
- **Collection**: of values of the above types

Expressions and ty  
(775)

## Comments

Good formal specifications should be accompanied by comments containing natural language explanations of what the constraints assert. This can ease the task of explaining (and debugging) a formal specification. In OCL documentation can be provided by annotate constraints with comments:

```
-- OCL only has inline comments like this.
```

these are the same as inline comments in Java:

```
// Comments like this in Java
```

## Contexts and Identifiers

We will use the class diagram in Figure 25.2 to illustrate the construction and use of different types of constraint on a UML class diagram. The figure illustrates a design for a coursework management system. Students are registered for particular courses. Each course has a number of coursework assignments that a student completes by making a submission. Notice the submission-coursework relationship is an example of the abstraction-occurrence design pattern.

A constraint must be associated with a particular context on the class diagram. The context is the starting class, operation or attribute from which other parts of the constraint are evaluated. The identifier of the context is preceded by the **context** keyword. In addition, the type of the constraint must also be specified.

The types of constraint we will use are either invariants (keyword **inv**), operational pre-conditions (**pre**) and operational post-conditions (**post**). Constraint types are denoted after the context identifier and before the constraint expression. Two equivalent invariant constraints are:

Running example  
(777)

Expression context  
and type (778)

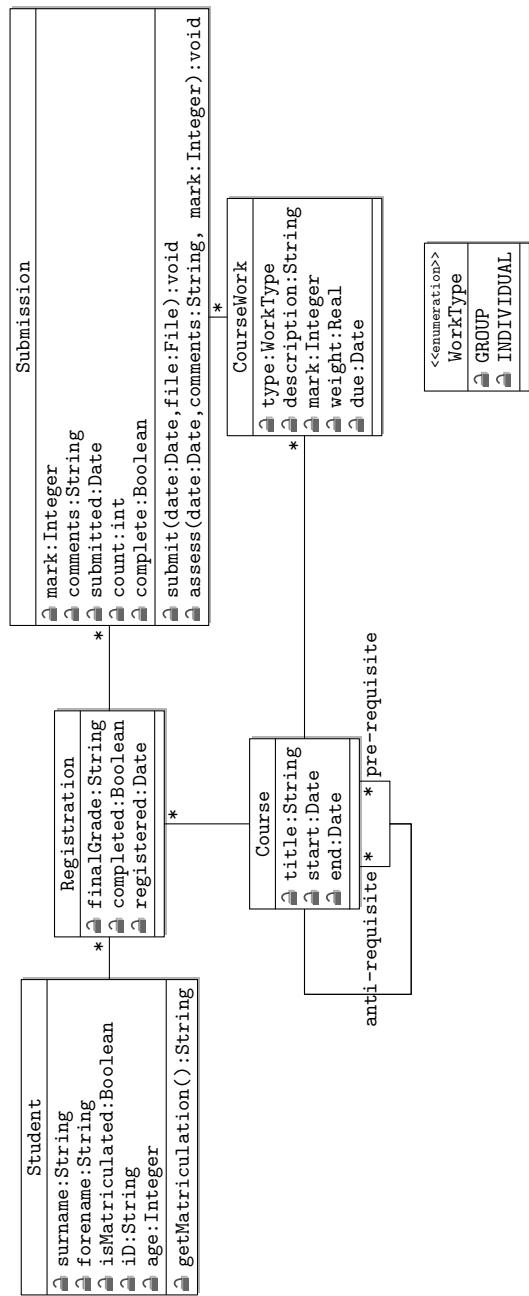


Figure 25.2: running example – coursework management system.

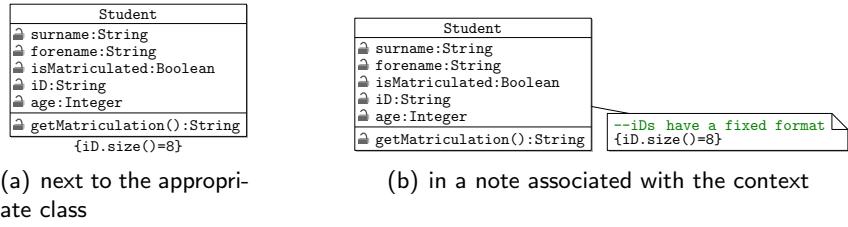


Figure 25.3: placing constraints directly on the class diagram

```
context Student inv: iD.size()=8
```

and:

```
context Student::iD:String inv: size()=8
```

Constraints can also be placed directly on a class diagram. Figure 25.3 illustrates two options for annotating class diagrams with constraints. In Figure 25.3(a), the constraint is placed next to the associated context class. In Figure 25.3(b) the constraint is placed in a note associated with the context class.

Identifiers are used in constraints to navigate across a class diagram. Identifiers can be used to refer to attributes or to associations with other classes.

For example, consider the (incomplete) constraints shown below:

- attributes can be referenced directly in their own context (i.e. from within the same class as the context) or from other classes:

```
context Student inv: surname
```

```
context Registration inv: finalGrade...
```

```
context Registration inv: student.surname...
```

- associations can be referenced from a local to a remote context:

```
context Registration inv: course...
```

```
context Course inv: registrations...
```

```
context Submission inv: registration.finalGrade...
```

Several intermediate association labels can be used together to navigate across a class diagram if necessary. For example, we can navigate from the Submission context, to the Registration of the student who made the submission, to the Course the submission was for, to the collection of Coursework that all students must submit for a course:

```
context Submission inv:  
registration.course.courseWork...
```

Figure 25.4 illustrates the navigation of the class diagram made by this series of identifiers.

Navigating the class diagram (780)

## Operators

Just as in imperative programming languages, operators are used to combine literal values and sub-expressions into expressions. The operators available in OCL are:

- comparison

- equality (the same as == in Java, note that this is *not* assignment):

```
4.0 = 4.0
```

```
players.size() = 11
```

- inequality (the same as the != operator in Java):

```
course1 <> course2
```

- less than: <
  - Less than or equal to: <=
  - greater than: >
  - greater than or equal to: >=

Read comparisons from left to right.

Comparison operators (781)

- arithmetic (following the same rules concerning integers and figures as other languages):

```
4 * 3.0 / 2 + 3 - 5
```

- logical

- conjunction (same as && in Java):

```
x > 4 and 4 > 3
```

- disjunction (same as || in Java):

```
x > 4 or 3 > 4 + 5
```

- negation (same as ! in Java):

```
x > 4 or not 3 > 4 - 3
```

- implication (true, unless it is raining and the grass is not wet):

```
rainingToday implies wetGrass
```

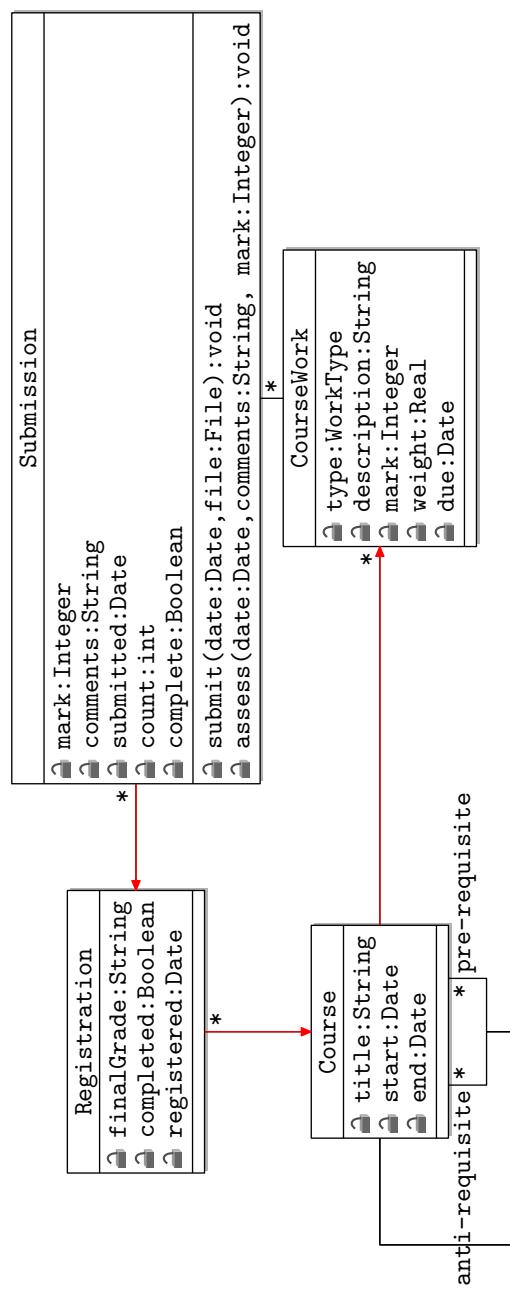


Figure 25.4: navigating the class diagram

## Strings

String values have a number of pre-defined operations that can be useful for many constraints:

Strings (783)

- literals

```
"Storer"  
"Requirements Engineering"
```

- length:

```
"Software Engineering".size()
```

- sub-strings:

```
"Software".substring(0,1) = "S"
```

- as attributes:

```
title.size() < 100
```

```
UNIVERSITY_NAME = "University of Isla"
```

## Putting It All Together

We have now covered enough of the OCL notation to consider some example constraints that are appropriate for the course management system in Figure 25.2.

Examples (784)

```
-- surname field must be less than 100 characters  
-- in length  
context Student  
inv: surname.size() < 100  
  
-- a submission cannot be given a higher mark  
-- than set for the course work.  
context Submission  
inv: mark <= coursework.mark  
  
-- a student must be registered on a course  
-- before it starts  
context Registration  
inv: date.before(course.start)  
  
-- all coursework must be submitted before  
-- the end of the course  
context Submission  
inv: submitted.before(registration.course.end)  
;
```

Note that we are embedding organisational policy into the system specification in the constraint below - that coursework will not be accepted for marking after the due date unless the student has an extension.

```
-- all coursework must be submitted before
-- the due date, unless the student obtains an
-- extension
context Submission
inv: submitted.before(courswork.due) or hasExtension
```

Accessing a ‘null’ valued association (even if it has an arity of 1) results in an empty set - this does *not* mean that finalGrade is an empty length string.

```
-- an item of course work must be associated
-- with a course.
context CourseWork
inv: not course->isEmpty
```

The constraint below may not be necessary if submissions are not created until the student makes a submission. Formal specification requires iterative development to get right, just like other software development methods. Redundant constraints can be removed once the problem domain is better understood.

```
-- coursework cannot be marked if it has not been
-- submitted.
context Submission
inv: submitted->isEmpty implies
    mark->isEmpty and comments->isEmpty

-- a grade cannot be assigned until the student
-- completes the course.
context Registration
inv: completed = false implies finalGrade->isEmpty
```

## 25.5 Collections

The navigation of an association with multiplicity greater than one is evaluated to a *collection* of instances of the remote class type. For example, navigating the registered association between Student and Course will give the collection of courses a student is registered for. If we imagine the navigation occurring at runtime, then we might display the collection of courses for a particular student as:

```
registered = {Software_Design,
  Requirements_Engineering}
```

Just as in other programming languages, there are a number of different types of collection.

- Collection (or bag)
- Sets
- Ordered sets

- Sequences

These have properties equivalent to the mathematical terms. Confusingly, Lethbridge and Laganière [2005] denote a sequence with a `{ordered}` association decoration.

The `ClassName.allInstances()` operation gives all the instances of a class in the model. For example:

```
context Course
inv: allInstances() = {Software_Design,
  Requirements_Engineering, ...}
```

## Collection Properties

All collections, regardless of type, have a number of common properties. Collection properties can be:

Collection properties (786)

- information about the current state of the collection; or
- other collections of the elements of the collection themselves.

Collection properties are operations on the collection, accessed using the `->` symbol. Some properties are:

- `collection->isEmpty() : Boolean`

which asserts whether the collection is empty or not

- `collection->size() : Integer`

which asserts the current size of the collection, for example:

```
context Student inv:
  registered->size() < 20
```

asserts that no student can be registered on more than one course. Note that:

```
(Collection.size() = 0) = Collection.isEmpty()
```

should always hold.

More Complex Properties (787)

- `collection1:T->intersection(collection2:T):Collection:T`

Gives a set containing only values that are in collection and collection2. This is the same as:

$$collection1 \cap collection2$$

- `collection1:T->union(collection2:T):Collection:T`

Gives a set containing all values in collection1 and collection2. This is the same as:

$$collection1 \cup collection2$$

- `collection:T->select(v:T | b(v) ):Collection:T!`

Selects every element v of collection for which `b(v)` is true. This is the same as:

$$\{v : T | v \in collection \wedge b(v)\}$$

- `collection:T->forAll(v:T | b(v)):Boolean!`

Asserts that `b(v)` holds for every element of collection. This is the same as:

$$\forall v : T \bullet b(v)$$

- `collection:T->exists(v:T | b(v) ):Boolean`

Asserts that `b(v)` is true for at least one element of collection. This is the same as:

$$\exists v : T \bullet b(v)$$

Some example uses of properties are given below, together with a comment describing their meaning.

Example properties  
(788)

```
-- selects all students whose surname is McAdam
-- note this is not a constraint.
Student.allInstances()
    ->select(s:Student | s.surname="McAdam")
```

```
-- Asserts that all students must be over eighteen
context Student
inv: allInstances() ->forAll(s | s.age >= 18)
```

```
-- A course cannot be both an anti-requisite and
-- a pre-requisite for another course.
context Course
inv: prerequisite->union(
    anti-requisite)->isEmpty
```

```
-- only submissions for group coursework are
-- shared.
context CourseWork
inv: type=INDIVIDUAL implies
    submissions->forAll(s1,s2 |
        s1.registration <> s2.registration
        implies s1 <> s2)
```

Properties that are also collections can be chained together.

## 25.6 Constraints on Operations

As for classes and attributes, operations can also be the context for a constraint. Operational constraints are useful for expressing the pre and post conditions of the operation. The constraint shown below illustrates how to use OCL to denote operational constraints.

```
context Student::getMatriculation():String
let initial:String = surname.substring(0,1)
pre: isMatriculated = true
post: result = iD.concat(initial)
```

Constraints on  
operations (789)

The example illustrates several new features of constraints:

- an operation is denoted as a context using the double colon :: symbol, followed by the operation's signature, just as for attributes;
- *variables* can be used to represent intermediate expressions in more complex constraints using the let keyword, particularly where the same expression is used in several places. In the example above, the variable `initial` is asserted as being equal to the first character of the student's surname when the operation is invoked. Note that variable identifiers are asserted as being equal in value to the expression, rather than being an assignment;
- *pre conditions* specify the situation before an operation can be invoked using the `pre` keyword. In the example, the pre-condition constraint states that a student must be matriculated before the `getMatriculation()` operation can be invoked; and
- *post conditions* specify the state the system must be in after an operation has been invoked, and the return value from the operation, based on the state of the system before the operation was invoked and any input values. The post condition in the example (denoted using the `post` keyword) states that the return value (`result`) of the operation must be the student's ID value appended with the first letter of their surname.

Note that the operational constraints do not control how the operation is invoked, only what specification the implementation must satisfy.

A larger example of an operational constraint is shown below:

Large example (79)

```
context Registration::calculateGrade():String  
  
pre: completed=true  
  
let finalMark:Real =  
    submissions->collect(  
        s | s.mark*s.courseWork.weight)->sum()  
  
let totalMark:Real =  
    course.courseWork->collect(  
        c | c.mark*c.weight)->sum()  
  
let percent:Real = finalMark/totalMark
```

The `collect` property is a variant of `select` that returns a collection which is the result of applying the `collect` expression on each element in the collection. In the example, the application of `collect` collects the weighted mark of each item of coursework associated with a `registration` instance.

The `sum` property can be used on collections of integers and reals, giving the sum of all elements in the collection. In the example, the `sum` property is used to obtain the student's (associated with the registration for the course) final mark.

```
post:  
    if finalPercent >= .8  
        then result = "A"  
    else if percent < .8 and percent >= .7  
        then result = "B"  
    else if percent < .7 and percent >= .6  
        then result = "C"  
    else if percent < .6 and percent >= .5  
        then result = "D"  
    else result = "E"
```

The constraint is used to specify the `calculateGrade()` operation which translates a percentage mark into a grade in the range "A" to "E".

The examples shown below introduces some more features of OCL which are useful when specifying the changes in system state which occur when an operation is invoked.

```
-- whenever a new submission instance is created  
-- the number of attempted submissions should be 0  
context Submission::count:Integer  
init: 0
```

The constraint shows the use of the `init` keyword to assert the initial value of an attribute when an instance of the class is constructed (or when the class is loaded in the case of class attributes). In the example, the `count` attribute is initialised to 0 for the number of times a student has submitted the same item of coursework.

```
-- pre and post conditions for submitting
```

```
-- course work
context Submission::submit(date: Date, file:File)
pre: date.before(courseWork.date)
post: completed = true and count = count@pre+1
```

The constraint shows the use of notation for using the state of an attribute before an operation is invoked within a constraint for the state of the attribute *after* the operation has completed using the @pre decorating of the attribute identifier. In the example, the post condition states that the count attribute for the submission instance after the operation is completed is asserted to be one plus the value of the count attribute immediately before the operation was invoked.

```
-- define value of derived attribute
-- a submission is considered complete if the
-- mark is greater than 25%
context Submission::complete:Boolean
derive: marks / courseWork.marks > .20
```

The derive keyword is used to declare the value of a derived attribute. A similar invariant constraint is:

```
context Submission
inv: complete = (marks / courseWork.marks > .20)
```

However, note that a derived value is calculated by the OCL engine. An invariant is checked against a value that is calculated by the implementation.

## 25.7 Model Driven Development

Formal descriptions of constraints can provide substantial benefits in clarifying the specification for a software system, particularly if the formal constraints are accompanied by natural language descriptions of their intent. However, the real power of formal specifications can be leveraged when they are used to drive software development and verification processes.

During *model driven development* (sometimes referred to as model driven architecture), software developers concentrate of specifying models of what software is supposed to do, rather than implementing the behaviour directly. The key artifact in a model driven development is a *Platform Independent Model* of a software system. For example, an abstract PIM could be developed using UML class diagrams annotated with OCL constraints. A PIM model can be automatically:

Model driven  
development (792)

- checked for problems including, syntax errors, consistency between constraints and redundant constraints;
- checked against an instance, or snapshot model of the system; and
- transformed into an implementation for a given *target platform*.

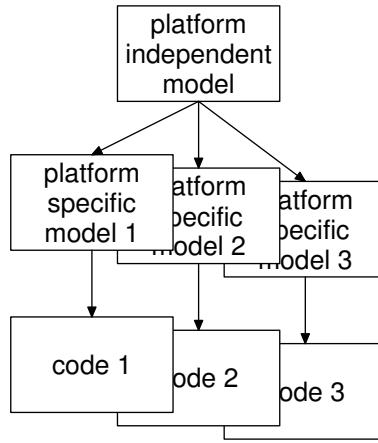


Figure 25.5: model driven development

Model driven development with UML and OCL (793)

There are several tools available for developing and validating PIMs built in UML/OCL, for example:

**OCL Environment (OCLE)**, which we will use in Workshop 25.9:

<http://lci.cs.ubbcluj.ro/ocle/>

**OCL Tool for Precise UML Specifications (Octopus)** which appears to be no longer supported:

<http://octopus.sourceforge.net/>

**Model Development Tools-OCL (MDT-OCL)**, part of the eclipse model development tools framework:

<http://www.eclipse.org/modeling/mdt/?project=ocl>

**DresdenOCL**:

<http://www.dresden-ocl.org/index.php/DresdenOCL>

Most tools are provided as a plug-in to the Eclipse IDE. Cabot and Teniente [2006] provide a survey of MDA tools, including those supporting UML/OCL.

Most UML/OCL tools also provide mechanism's for deriving executable object oriented programs from a PIM. Figure 25.5 illustrates the general transformations that are performed on a platform independent specification model to generate an executable implementation. Initially, the PIM is transformed into a *platform specific model* (PSM). Transformations are specified by a mapping or *platform definition model* describing how the PIM should be instantiated in a PSM.

Abstract data types specified in the PIM are translated into equivalent types available on the target architecture. For example, the abstract Date type for an

<b>portability</b>	the model is sufficiently abstract to be automatically translated into different implementations for different object oriented platforms
<b>interoperability</b>	system components can be composed at the abstract model level and then deployed to particular platforms
<b>maintainability</b>	system evolution only requires changes to the model since implementations can just be regenerated from the model

Table 25.2: benefits of model driven development

attribute on a UML class diagram might be translated to the `java.util.Date` type on a UML class diagram of a Java program design.

Finally, the PSM is transformed into executable code, for example into a collection of Java source code files. Any OCL constraints could be translated into Java `assert()` statements, or similar validation chunks of code, either at the beginning or end of a method body.

Model driven development has several benefits for the software development process, as summarised in Table 25.2.

Benefits of model  
driven development  
(794)

## Summary

Formal methods and notations are used in problem domains where high assurance as to the correctness of a system with respect to its specification is required. However, formal methods do not solve the problem of ensuring that the specification itself is correct with regard to the real needs of the customer. The cost and time commitments required to follow formal development methods can mitigate against their use, particularly when the problem domain itself is rapidly evolving.

OCL is a declarative specification language used to express constraints that evaluate to `true` or `false`, it cannot be used to affect the state of an object oriented program. OCL can be used to specify program invariants and operational pre and post-conditions on class diagrams.

## 25.8 Exercises

1. The class diagram in Figure 25.6 shows two classes extracted from a (very simplified) Library system.

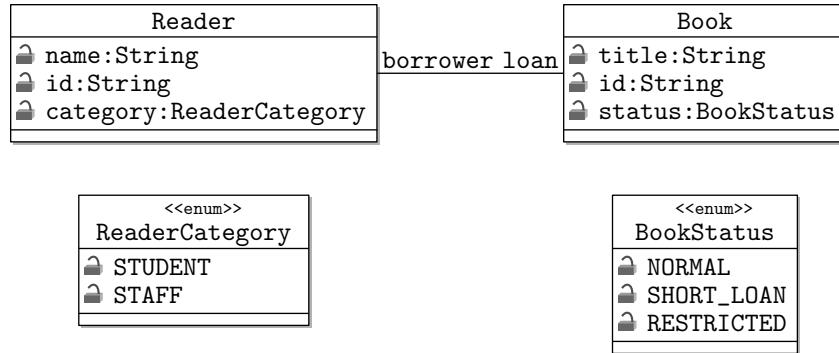


Figure 25.6: library system

- (a) Write OCL constraints for the following rules:
    - i. A book's title is restricted to no more than 25 characters.
    - ii. If a book's status is currently restricted then nobody can borrow it.
  - (b) The Library has rules for borrowing books based on the category of the reader:
    - student readers are permitted to have a maximum of 10 loans at any given time; and
    - staff readers are permitted to have up to 20 normal loans and 5 short loans at any given time.
    - i. Write these rules as OCL constraints.
    - ii. Consider how you might express these rules diagrammatically using subclasses of `Reader` and `Book`.
  - (c) The Library changes its rules for student readers, so that they are permitted a maximum of 10 loans, not more than 3 of which can be short loans.
    - i. Write OCL constraints to express these new rules.
    - ii. Is it possible to express these extended rules diagrammatically?
2. The class diagram in Figure 25.7 is for a company project management database.  
It focuses on the association between company staff and proposals for new projects that they are involved in reviewing (association: `reviewers/`

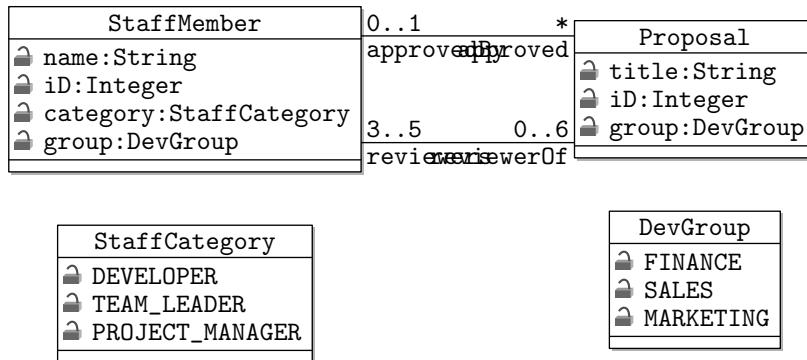


Figure 25.7: project proposals

`reviewerOf`) and which member of staff approves a project, if it goes ahead (`approvedBy` / `approverOf`).

- (a) Identify the constraints that are imposed on this model by the two associations given.
  - (b) It is company policy that no `Proposal` may be reviewed by a staff member who is a member of the same development group (`DevGroup`) that submitted the proposal. Express this rule in OCL.
  - (c) It is also company policy that every team of reviewers for a proposal must include at least one team leader. Express this rule in OCL.
  - (d) Following the review process a project may be approved but only by a project manager. Express this constraint in OCL.
3. The class diagram in Figure 25.8 is extracted from the class diagram for an electronic voting system.
- (a) Explain the constraints that are imposed on the relationships between the classes in the system.
  - (b) Give a natural language explanation of the OCL statement associated with the `Election` class.
  - (c) An election description is something like “Election of a Member of Parliament to serve in the constituency of Kirkcaldy and Cowdenbeath”. Write an OCL constraint to restrict the length of the election description to 500 characters.
  - (d) The number of votes cast in the election should be the same as the number of voters on the marked roll. Express this rule in OCL.
  - (e) A voter should only be able to cast at most one vote per election. Express this rule in OCL.

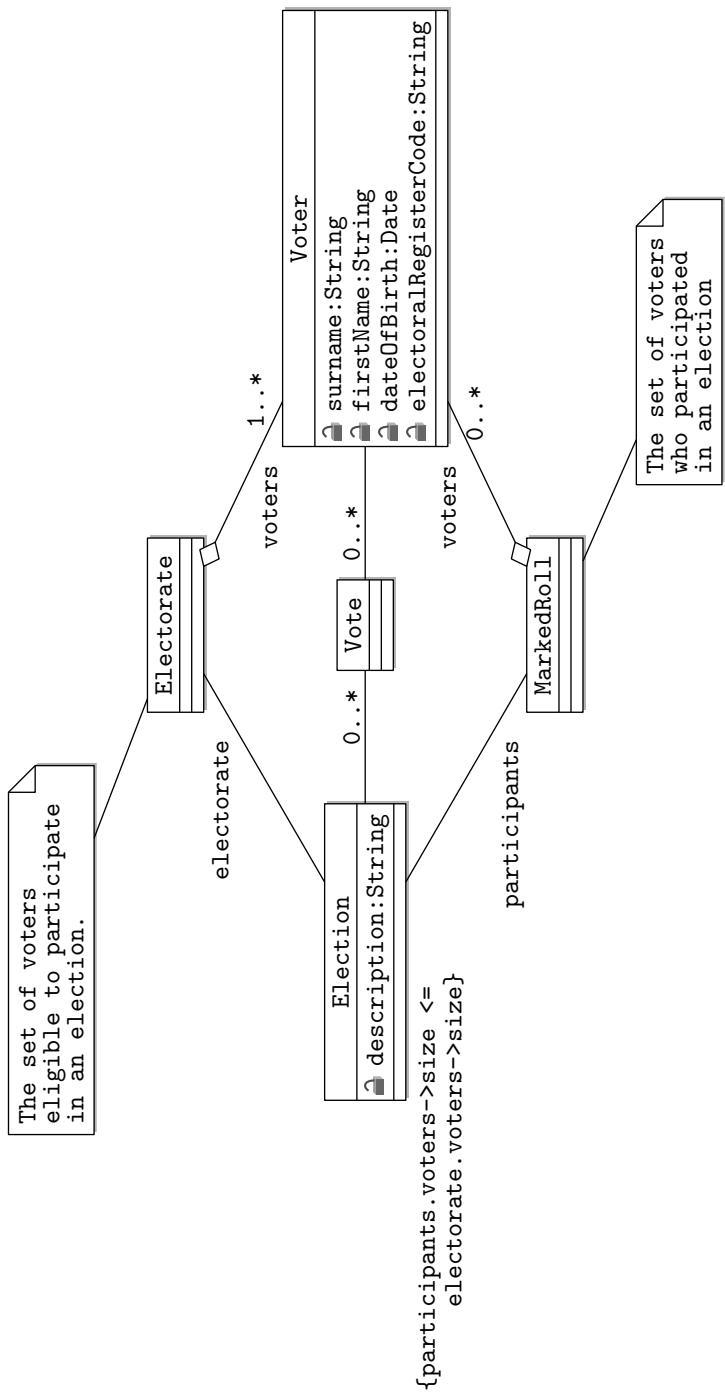


Figure 25.8: voting system

4. Read the following papers

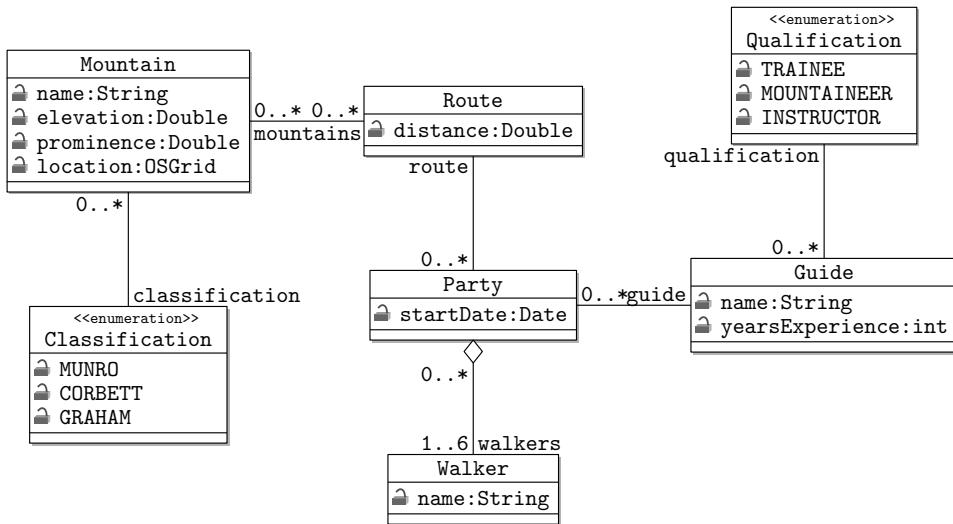


Figure 25.9: mountain walks system domain model

**PLACE HOLDER FOR wordsworth99getting**

**PLACE HOLDER FOR glass04mystery**

Do you think the approach advocated by Wordsworth is acceptable/workable? Is Glass' critique of the approach valid? Explain your answer.

5. An outdoor travel company is developing a system for managing guided walks amongst the mountains of Scotland. The class diagram in Figure 25.9 shows part of the domain model for the system.
  - (a) Explain the constraints that are imposed on the relationships between the classes in the system.
  - (b) All walks are to last for one day only, so it has been decided to limit all routes distances to 30 miles or less. Express this rule in OCL.
  - (c) The travel company maintains safety rules concerning the qualifications of guides on routes. In particular, any route that includes at least one Munro must have a guide who is qualified as an instructor. Express this rule in OCL.
  - (d) During development the company realizes that the party-guide-walker constraints should be adapted to allow for larger parties. Parties can now consist of up to 30 walkers. There must still be at least one guide for every six walkers in a party. There must still be at least one suitably qualified guide, according to the rule specified in part 5c, to lead the party.

Explain how you would adapt the model defined so far (both the UML and the OCL) to accommodate the new arrangement, and express any new OCL constraints required.

- (e) A colleague proposes to replace the Classification and Qualification enumeration classes in the class diagram with sub-classes of Mountain and Guide respectively. For example, the Classification enumeration will be replaced with three sub-classes of Mountain (Munro, Corbett and Graham). Your colleague argues that this approach will improve clarity in the diagram and reduce the number of OCL constraints required.

State whether you agree with this approach and justify your argument.

## 25.9 Workshop: Model Driven Development

This workshop gives an introduction to model driven development using a light-weight tool, OCL-Environment (OCLE). OCLE is a pure Java application that can be run from a terminal or from within Eclipse. You can find out more about OCLE from the project webpage.<sup>1</sup>

### 25.9.1 Setup

Download a copy of OCLE from Moodle:

```
http://fims.moodle.gla.ac.uk/file.php/128/workshops/ocle.zip
```

The bundle includes the user manual and some example projects that come with the distributions.

Extract the archive into your workspace. From now on, this tutorial will assume that you have extracted the archive to a the directory path:

```
/users/level3/l3tws/workspace/ocle-2.0.4
```

You will need to adapt this path to your own circumstances. Open the start up script file:

```
/users/level3/l3tws/workspace/ocle-2.0.4/run_linux.sh
```

and change line 5 of the file to:

```
OCLE_HOME=/users/level3/l3tws/workspace/ocle-2.0.4
```

Save the modified script. Execute this start up script from the OCLE\_HOME directory:

```
./run_linux.sh
```

This should start the OCLE application. Notice that the application GUI is split into four panels as shown in Figure 25.10. The top left panel contains navigation trees for examining projects and models. The panel directly below it is used to show the properties of a selected element (class, operation, attribute, object and so on). The panel along the bottom of the application window is used to show output. This panel is divided into tabs for:

'LOG', which records user actions;

'Messages', where compilation errors are shown;

OCL output, where the result of evaluating an OCL expression is shown;

'Evaluation' which shows where a constraint is violated in a UML model; and

'Search results'.

---

<sup>1</sup><http://lci.cs.ubbcluj.ro/ocle/>

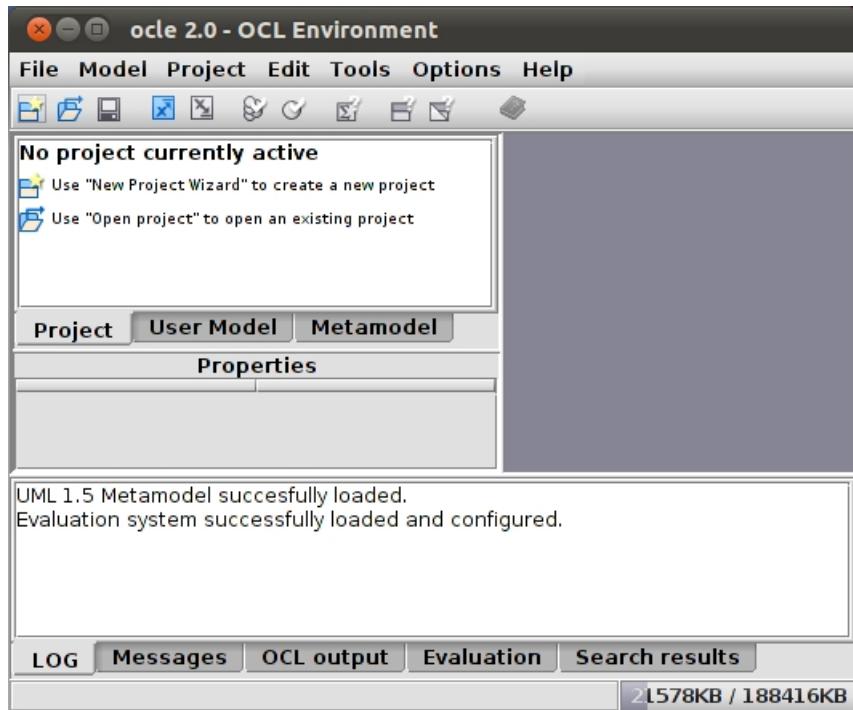


Figure 25.10: the OCLE application after start up

Finally, there is a panel for editing UML models and constraint files. You will also need to download and extract the StudentLifeElipse project archive from Moodle. The archive contains a class diagram and collaboration diagram for the coursework submission part of the StudentLifeElipse application.

### 25.9.2 Exploring a Project

The first step when developing a software application using OCLE is to create a new project, which will store the UML models, OCL constraint files and snapshot object diagrams of the system. Together, these artifacts can be used to generate Java source code for the application that can be compiled and executed.

Creating UML models in OCL is essentially the same as doing so in ArgoUML, Umbrello and many other UML diagram editors, so we won't focus on this in OCLE. Instead, we will use a pre-prepared model of the StudentLifeElipse project used as an example during lectures. To open the project, choose the:

Project → Open

menu item or choose from the tool bar. Then navigate to the file:

```
studentlifeellipse/studentlifeellipse.oehr
```

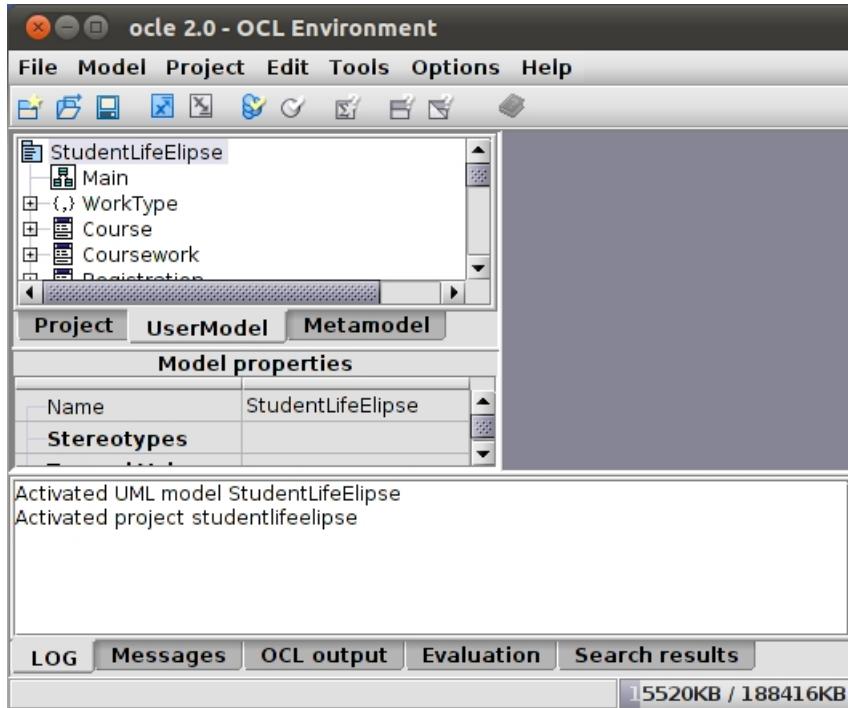


Figure 25.11: the OCLE environment after loading the StudentLifeElipse project

in the archive for the StudentLifeElipse project downloaded during setup. Click 'Open' for the selected file. OCLE will load the project and associated model and constraint files. The GUI should look something like Figure 25.11. You should see the following message in the LOG tab:

```
Activated UML model StudentLifeElipse
Activated project studentlifeelipse
```

Notice there are three tabs on the navigation panel on the left hand side of the GUI, for 'Project', 'User Model' and 'Meta Model'. We will be working with the 'User Model' and 'Project' tabs.

You can open the UML class diagram which describes the StudentLifeElipse project by selecting the User Model tab (which displays the project as a hierarchical tree) and then double clicking on the 'Main' class diagram node (directly beneath the root). A class diagram will appear in the main window, showing the classes of the StudentElipseCycle project and the relationships between them. Figure 25.12 shows the OCLE application with the class diagram open.

There is also an instance diagram which describes a 'snapshot' usage of the application with some objects in the context of the University of St Glasburgh. The StudentLifeElipse model and constraints can be checked against the instance diagram. To open the instance diagram expand the  Collaboration icon in the User Model tab (you may have to scroll down the model tree) and

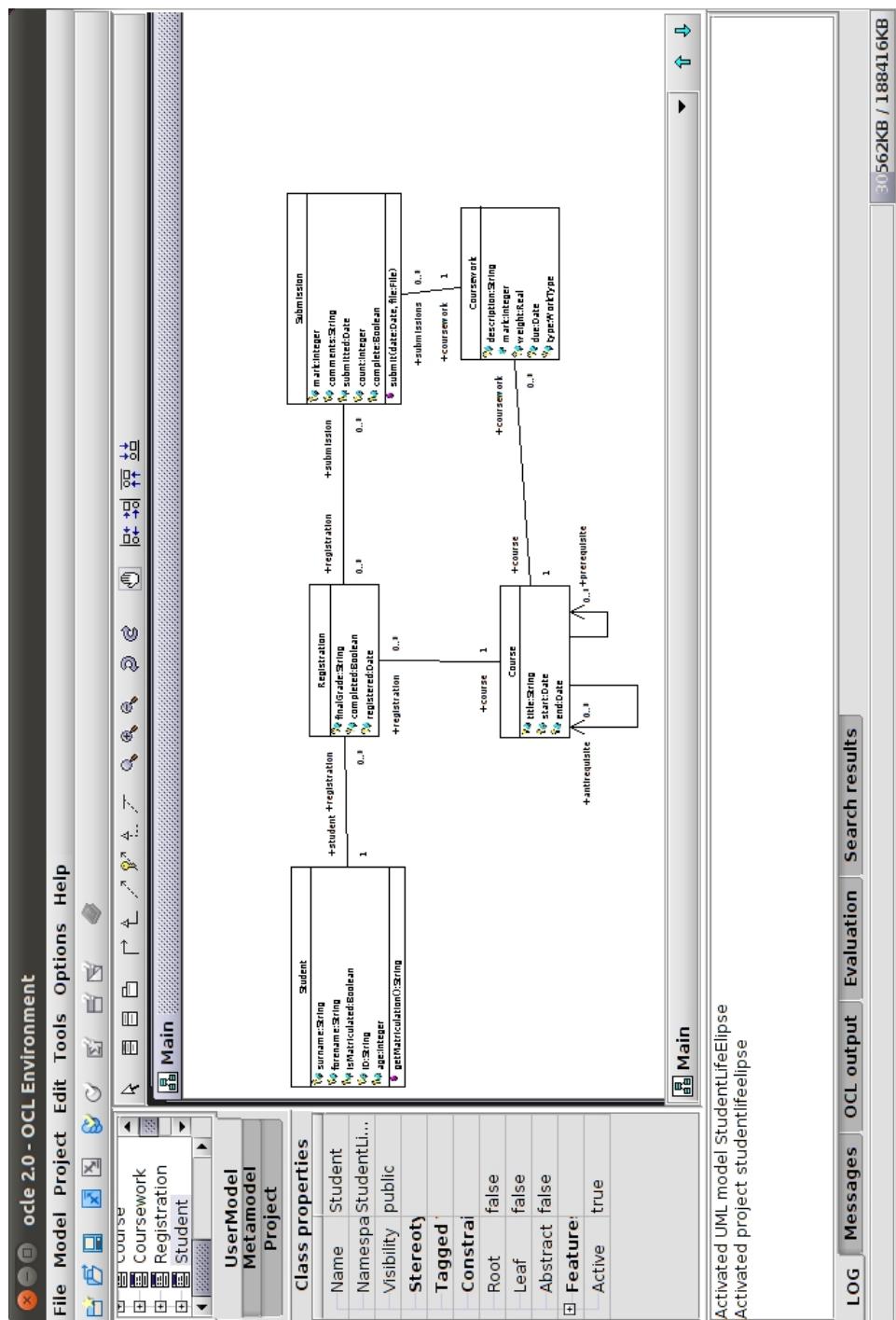


Figure 25.12: the StudentLifeElipse class diagram

double click on the  UofStGlasburgh instance model icon. Figure 25.13 shows the instance diagram containing two students, and a submission for an exercise for an object oriented software engineering course.

The constraints that have been specified so far are contained in the file:

```
studentlifelipse/studentlifelipseModelLevel.bcr
```

the extension is short for ‘business model constraints’. The constraint file is automatically loaded with the project. To open the file, double click the  studentlifelipseModelLevel.bcr file under the Project tab. You may have to open the ‘Constraints’ node in the project hierarchy. Figure 25.14 shows the constraints file open in the editor.

### 25.9.3 Checking Constraints

The constraint file needs to be compiled before it can be checked against the UML model. Choose the:

Tools →  Compile active file

menu item. Alternatively, click on the button marked with the  icon on the tool bar. You should see the following message in the LOG tab.

```
Compiling...successfully completed
```

Now validate the model against the constraints. Choose the:

Tools →  Check model

menu item. You should see the following messages (with adjusted path name) in the LOG tab:

```
Opened file /users/level3/l3tws/ocle_2.0/Temporary/
Diagrams/UofStGlasburgh.xml.
13 evaluations requested.
13 have been performed, 1 problem(s) found. Please check
the 'Evaluation' tab.
```

This indicates that each object in the model was checked against the invariant constraints in the active OCL file. One evaluation is conducted per invariant constraint, per instance. There are five invariant constraints for the student context and five student instances (giving ten evaluations). There is also one constraint each for Course, Coursework and Submission, with one instance of each in the model, giving a total of 13 evaluations.

Notice that OCLE has reported a problem with one of the instances. Clicking on the Evaluation tab and expanding the Errors tree reveals the problem, as shown in Figure 25.15. The figure shows that an evaluation failed for one of the instances of student.

Double clicking on the:

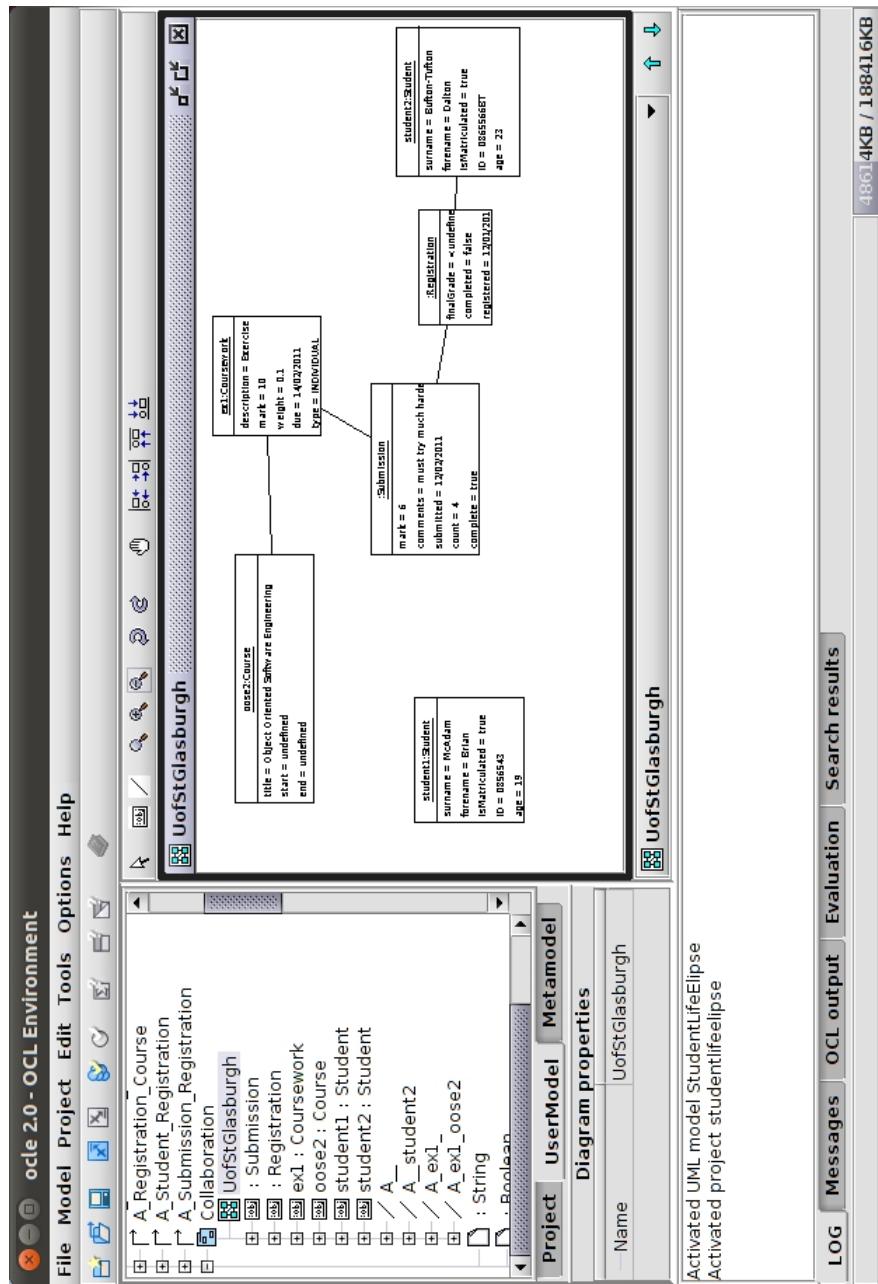


Figure 25.13: the UoStGlasburgh instance model

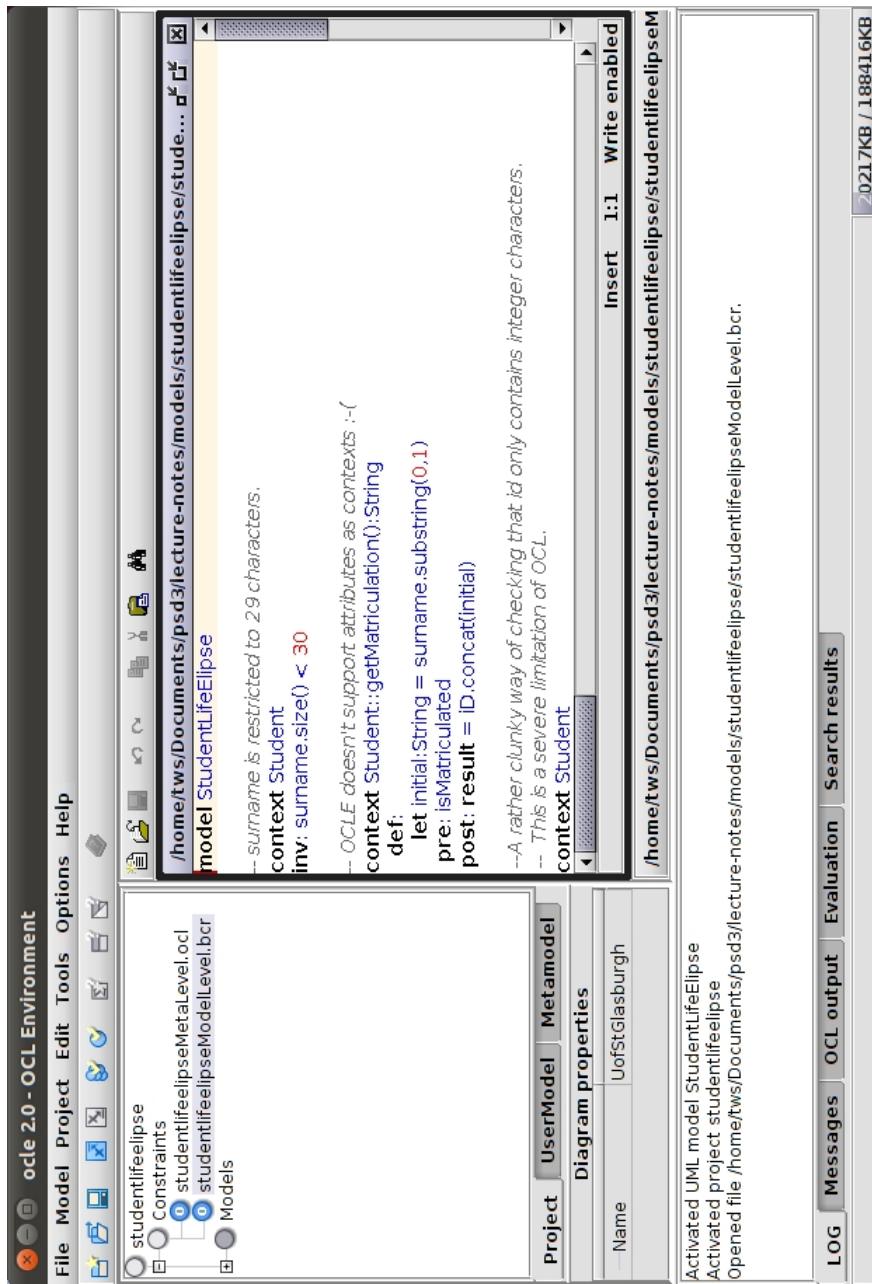


Figure 25.14: the StudentLifeElipse constraints file

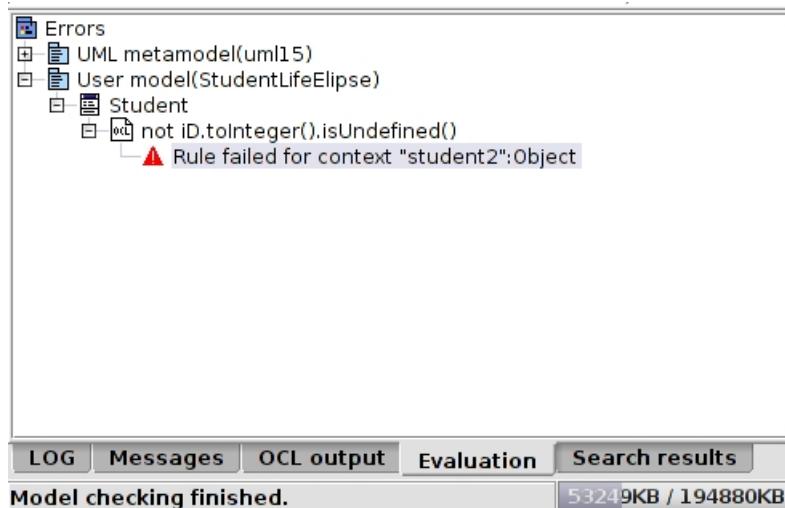


Figure 25.15: an error reported in the UoStGlasburgh model

Rule failed for `context "student2:Object"`

highlights the invariant that has been violated in the OCL file. The constraint is a (rather ugly) way of saying that an iD string must only contain digits. Looking at the instance student2 on the diagram shows that the student's matriculation string (0865566BT) has been used for their iD.

#### 25.9.4 Modifying the Snapshot

We need to correct the instance diagram for the University of St Glasburgh. To do this, go to the User Model tab, right click on the student2 instance and choose 'Edit Object'. A dialog will appear showing the current state of the object. Remove the last two characters from the iD attribute value and click 'OK'. Go back to the constraint file and re-compile it and re-evaluate the model, following the same steps as above.

You should now get the message in the LOG tab:

```
13 evaluations requested.
13 have been performed, 0 problem(s) found.
Model appears to be correct according to the selected
rules.
```

stating that the model now passes all the constraint evaluations.

#### 25.9.5 Adding More Constraints

Go back to the constraint file and add the following constraints:

1. A course title must be no more than 50 characters in length.

2. A student cannot have an empty surname if the forename is not empty.
3. A submission is not complete unless it has a mark
4. A submission cannot have a comment if no mark is assigned.
5. A student cannot make a submission for a course they are not registered on.
6. A student cannot register twice for the same course.

### 25.9.6 Generating Source Code

You can generate Java source code from the model which will include the OCL constraints as validation code. After compiling the constraints file, choose the:

Tools→ Generate code...

menu item. Set the output destination directory and then click ‘Next’ and ‘Ok’. Open one of the generated source code files. You will see that invariant constraints are contained in an inner class as methods that can be invoked by a checker.

### 25.9.7 Extensions

The OCLE archive contains a number of example projects in the:

`OCLE_HOME/help/examples`

directory. Try exploring and altering some of these projects. You can also try implementing the models from the exercises in OCLE.

**Part IV**

**Software Maintenance**

# Chapter 26

## Software Evolution and Maintenance

### Recommended Reading

PLACE HOLDER FOR sommerville10software

Recommended reading  
(806)

Software development as a process, rather than a product has been emphasised throughout this course. Although development efforts may be centered around a particular software product base (including source code, requirements specification, design documentation, user manuals, test suites, configuration files and deployment scripts) all these products *evolve* as the project progresses through a series of software releases, perhaps over many years. Chapter 4 discussed the environmental pressures which can cause software to evolve, such as rapid change in competing software applications, supporting technologies, or new requirements demanded from customers. Consequently, *software maintenance* activities often consume the majority of project resources over the lifetime of a software system, after an initial release.

Software evolution  
and maintenance  
(807)

### 26.1 Maintenance Costs in Software Engineering

Figure 26.1 illustrates a series of estimates as to the proportion of resources allocated to software maintenance activities in a software systems, between 1980 and 2006. Notice that there is an increasing trend in project resources devoted to maintenance of software, compared with other project activities, such as requirements specification and maintenance. More recent, survey based cost estimates of software maintenance are hard to come by, perhaps because there is a collective view amongst software engineering professionals that software maintenance now consumes more than 90% of resources in a project.

Resources allocated to  
maintenance in  
software projects  
(808)

This view reflects the growing characterisation of software projects as *brownfield* rather than *greenfield developments* (see Chapter 4). One way of viewing

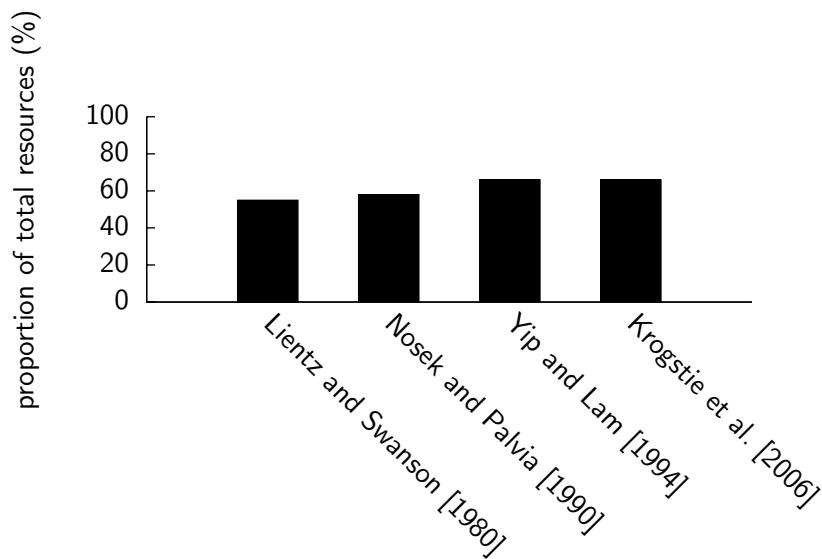


Figure 26.1: cost estimates of software maintenance between 1980 and 2006

a brownfield environment is as a larger scale software project to which new features are added and continual corrections and adaptations are made. From this perspective, almost all software project efforts can be categorised as maintenance, due to the work on or with pre-existing software artifacts. Other software development activities, such as requirements specification and design are conducted with respect to the constraints imposed by the brownfield environment. New artifacts introduced to a brownfield environment need to integrate with existing technology platforms and infrastructure, or large scale adaptation of the existing site needs to take place.

Lientz and Swanson [1980] characterised four different types of software activities which contribute towards these maintenance efforts:

- *adaptive* maintenance to respond to changes in the software system's environment, such as changes to the application programming interface of a system dependency, removal of support for a dependency by a vendor or the need to transfer the application to a software platform;
- *perfective* maintenance is concerned with the introduction of new features to the system at the request of a customer or user, or the improvement of the system's non-functional characteristics;
- *corrective* maintenance to remove defects from the system. Corrective maintenance occurs as part of the test-fix cycle (see the OOSE course notes); and
- *preventative* maintenance, which is concerned with introducing changes

Types of maintenance activity [Lientz and Swanson, 1980] (809)

to the system which will reduce the probability of failure.

Notice that these maintenance activities can be both a source of software evolution, as well as a response to it. For example, a perfective activity, such as the implementation of a new feature may introduce a software defect, which will require a corrective activity.

## 26.2 Lehman's Laws

Lehman [1980] proposed a series of 'laws' (hypotheses based on empirical observation of software projects) the evolution of software, which are useful in explaining the effects of software development activities on the structure of a software system.

Lehman's Laws  
[Lehman, 1980] (810)

- I    **continuing change:** changes in a system's environment causing pressure for change in the software system itself;
- II   **increasing complexity:** as software evolves, its structure becomes increasingly degraded, requiring on-going refactoring efforts;
- III   **self regulation of large-scale system evolution:** system attributes (defects reported, new features implemented) for mature, large-scale systems are constant between releases;
- IV   **conservation of organisational stability:** once established, software projects maintain a constant rate of progress, regardless of the resources committed; and
- V   **conservation of familiarity:** the rate of change between releases is broadly constant.

Other laws were introduced later, concerning software growth, declining quality and the software process [Lehman, 1996]. However, these appear to be in many ways adaptations of the original laws.

## Summary

Modern software engineering practice is dominated by maintenance activities, due to the continued evolution and adaptation of existing software systems. In addition, the development of new software can often be characterised as a software maintenance project, due to the existence of pre-existing platforms, libraries and infrastructures developed for previous projects.

Maintenance is an on-going activity, supported by well defined processes, such as comprehension, refactoring and platform migration. Wherever possible, software evolution should be managed using automated tools.



## Chapter 27

# Software Comprehension

### Recommended Reading

#### PLACE HOLDER FOR deimel90reading

Recommended reading  
(811)

The first step in many maintenance, development and re-engineering processes is often to gain an understanding of:

- what a software system does (specification); and
- how it does it (design and implementation).

Uses of software  
comprehension (812)

Consequently, *software comprehension* is an important skill in many software development tasks. Understanding a software system can be useful during:

- development, when integrating new components and libraries;
- reviews of unfamiliar code and/or documentation; or
- maintenance, when a software defect must be corrected, or a new feature implemented in an existing system. Nguyen et al. [2011] reported that about 50% of software maintenance efforts are expended just in understanding the existing source code base.

Software comprehension is concerned with gaining new knowledge about a software system based on existing or preliminary knowledge. There are different kinds of knowledge that can be gained about a software system:

Kinds of knowledge  
(813)

- *syntactic* knowledge is an understanding of the specific:
  - libraries,
  - commands,
  - methods,

- parameters and
- variables

that can be used to express instructions in a particular programming language or on a specific platform. For example:

- the `java.lang.Math` class has a method called `sin(double a)` that returns the sine of an angle, given in radians, and
- the `java.util.HashSet` class is a commonly used implementation of the features of a Set.
- the linux command `find ./ -name "*.java"` will find all java source files in the current directory tree.

When dealing with a particular software system, syntactic knowledge refers to an understanding of the specific names of the classes in the system and what their method and attribute identifiers and types are.

Syntactic knowledge must be used frequently to be retained at all, and will be gradually lost through a lack of use over time.

- *semantic* knowledge concerns an understanding of the meaning of a particular concept. Like syntactic knowledge, semantic knowledge can be either
  - general, such as an understanding of the operations that can be performed on an abstract data structure, or the key structural elements of a concrete data structure, or
  - task related, such as knowing the important domain information that must be captured in a student management system.

Semantic knowledge is less volatile than syntactic knowledge, because it concerns the principles which underpin a concept, rather than the specific implementation details on a particular platform.

There are also different strategies for gaining an understanding of a software system. These strategies have similar names to those adopted for software design and software testing:

- *top-down (plan based)* identifies known patterns in the target program. There are different levels of top-down comprehension of software source code:
  - strategic (global, language independent)
  - tactical (algorithms), and
  - implementation (operational);

- *bottom-up (step-wise abstraction)* searches for critical subroutines and determines their function, gradually working up the program hierarchy formulating abstractions from lower-level descriptions; and
- *hetarchic, middle out* is a hypothesis driven approach, in which questions concerning artifacts in the source code are formulated and then answered. For example:
  - This function failed because...
  - This variable is a...
  - This object is accessed by means of....

Soloway and Ehrlich [1984] conducted an experiment on the code comprehension approach of experts and novices. Participants were shown page of code for few seconds and asked to copy what they saw. Novices start from the top, get details right, but run out of information quickly. Conversely, experts get the structure of the software program right, such as major control structures, but miss out on all the details, such as the names of identifiers

Design studies that show people look for patterns at any level (i.e. they adopt a hetarchic approach). These are the hooks on which to build hypotheses about the system.

## 27.1 Source Code Comprehension

When reviewing software source code, program *structure* can be deriving from source code layout, if the code is reasonably well formed. Different types of indicators in the source code can be used as a guide to recovering the purpose and functionality of the code.

Code *chunks* are semantic abstractions over a fragment of source code. Dividing a program into chunks (and abstracting the details of each chunk) can ease comprehension of the overall system. Code chunks can be:

Source code chunks  
(815)

- identified at different levels of abstraction;
- labelled, using a method identifier, for example; and
- composed into higher level chunks.

Examples of code chunks include:

- control structures (**if**, **for** and so on),
- comments,
- groups of attribute or variable declarations,

- function declarations, and
- class definitions.

Beacons (816)

*Beacons* are indicators of a larger source code structure, for example:

- variable swapping is a beacon for a sorting algorithm,
- nested method calls for a recursive algorithm,
- operation identifiers which describe functional behaviour, for example `calculateIncomeTax(double salary)`.

When a beacon is found it can be used to recover the rest of the structure.

*Rules of discourse* describe conventions of programming in a particular language, for example:

- the use of capitalised nouns for class types in object oriented languages (`String`, `Date`, `BookCopy`) to differentiate from attribute, method and variable identifiers (`surname`, `length`, `toArray()`); and
- the convention for the operation signature of  
accessors (`public Type getAttribute()`) and  
mutators (`public void setAttribute(Type attribute)`)

for Java bean properties.

Studies [Wiedenbeck and Evans, 1986, Crosby et al., 2002] have suggested that expert programmers are more likely to detect beacons in program code than novices. However, violations of conventions associated with beacons cause more confusion for experts than for novices.

[Deimel and Naveda, 1990, pp18] provides a list of practical suggestions for conducting source code comprehension. In particular:

- Be sceptical and cautious, because source code, comments and other documentation may be inconsistent (1).
- Look for evidence of changes to the system such as code comments reporting change, version control system artifacts, log messages, maintenance documentation, defect tracking systems and feature request reports (10).
- Ask a colleague to review the code. It may be useful to review the code as a team (11).
- Be aware of the potential for attributes and variables with the same or similar identifier but different scopes, or types (12).

Tips for reading code  
[Deimel and Naveda,  
1990] (818)

- Look for variables that serve more than one function (17).
- Be aware that the system may contain redundant code that has not been retired (15).
- Use automated tools such as a debugger or profiler to trace the execution of code with test data (21).

You can find more tips and problem exercises in the original document [Deimel and Naveda, 1990].

## 27.2 Analysing Program Design

If a program is larger than can be recovered through source code inspection, it may be more appropriate to attempt to recover the larger scale features of the system using automated tools. Different aspects of the system can be recovered.

The *static structure* or architecture of a program can be recovered automatically from source code and presented as UML class, package or component diagrams. These tools can be used to identify artifacts such as:

- interfaces and operations;
- classes, attributes and methods;
- class associations and multiplicities; and
- inheritance hierarchies.

Automated analysis of program design (819)

Supplementary documentation concerning the system's application programming interface, can also be recovered from static analysis of source code, using generators such as JavaDoc.

System *structure*, including classes, attributes, association and inheritance relationships can be recovered from source code or compiled binaries using *reverse engineering* tools. Figure 27.1 shows a screenshot from the StarUML application, which can be used to reverse engineer class, component and deployment diagrams from Java source code.

The project being extracted is my third year team project, when I was an undergraduate on a similar course to PSD.

Structure Extraction using StarUML (820)

*Dynamic behaviour* can be observed using software profilers and debuggers, which record information about a program during execution, based on test data.

Useful information includes:

Observing dynamic behaviour (821)

- the number of invocations of each method;
- a *stack trace* of sequences of method calls based on sample data;
- the state of variables at different points in an application's execution;

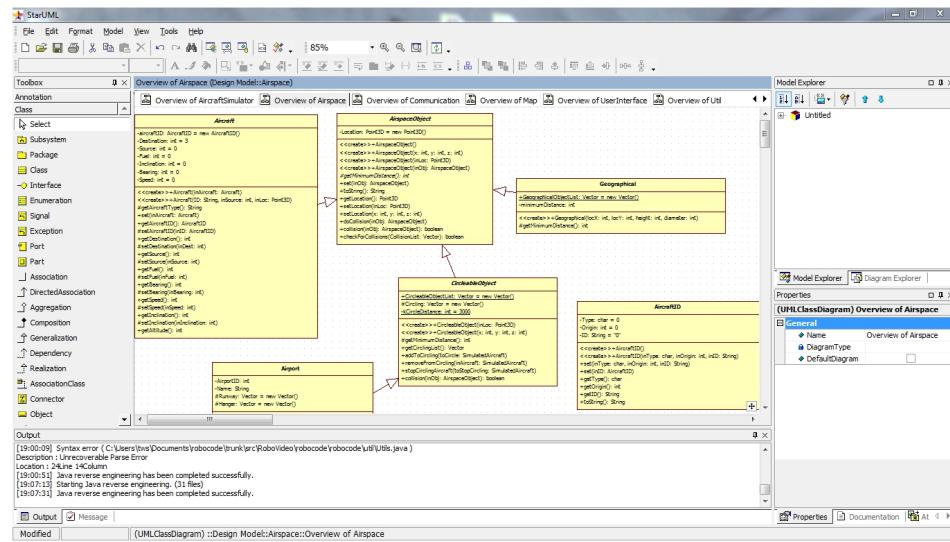


Figure 27.1: recovering structure using StarUML

- time in an object's scope;
- time in methods, including the duration of a method call and the proportion of total runtime; and
- call graph (visualised as a sequence diagram, for example).

The Eclipse Test and Performance Tools Platform (TPTP)<sup>1</sup> includes tools for profiling the runtime behaviour of Java applications. The profiler can be used to generate a variety of views on the behaviour of a software application. Figure 27.2 illustrates the summary view of the profiler.

### Eclipse TPTP Profiler (822)

The profiler was being used to examine the performance of a JUnit test class on a target class in a brute force password cracking application. The class is used to generate sequences of candidate passwords according to a specified rule (all strings of length four containing the character set [a-z] for example).

The view shows a hierarchy of packages, classes and operations that were called from within the application (notice that other operation calls within the Java SDK are filtered out by default). The view shows the number of calls made within each artifact, and the amount of time on calls.

Figure 27.3 illustrates the call tree view in the TPTP profiler. The view is similar to the summary view, except it only lists method calls in the left most column. Some of the method names are repeated, because the view is showing how much time was spent in each *call* of a method, rather than in the method itself.

### Eclipse TPTP Profiler – call tree (823)

<sup>1</sup><http://www.eclipse.org/tptp/>

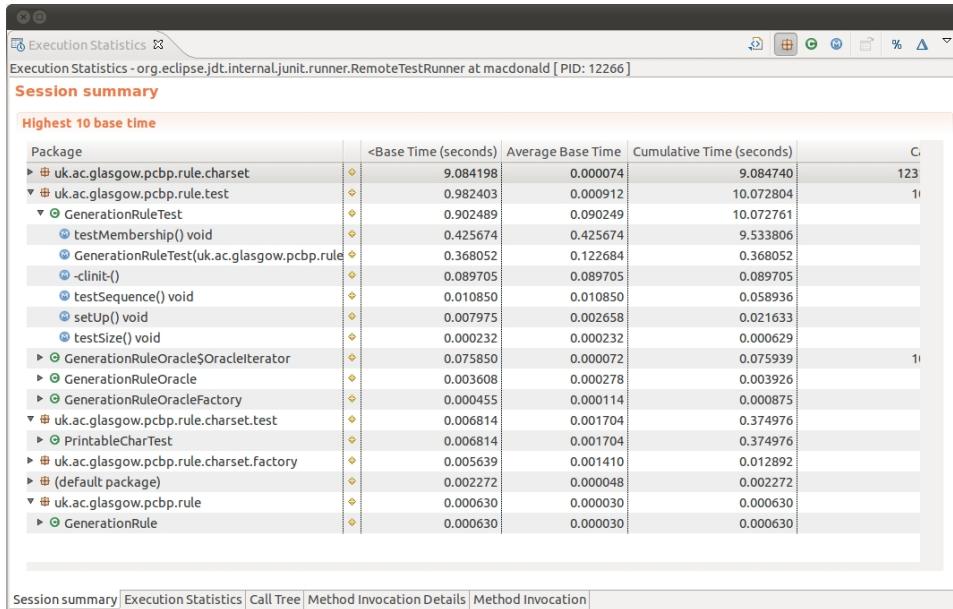


Figure 27.2: Eclipse TPTP Profiler summary view

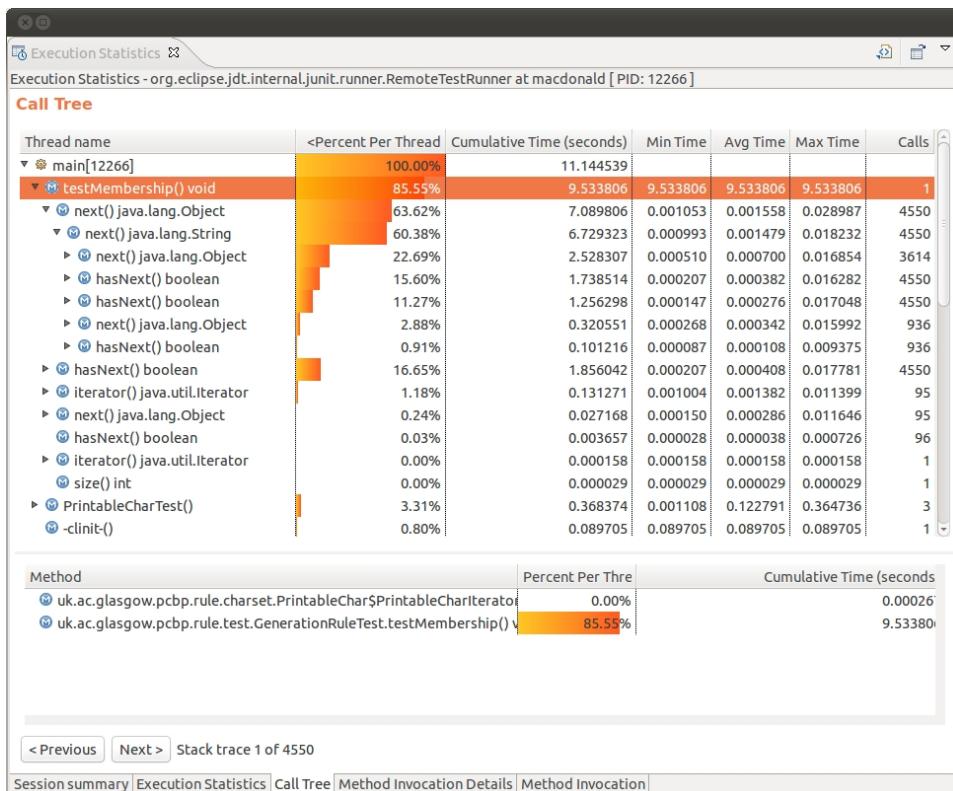


Figure 27.3: Eclipse TPTP Profiler call tree view

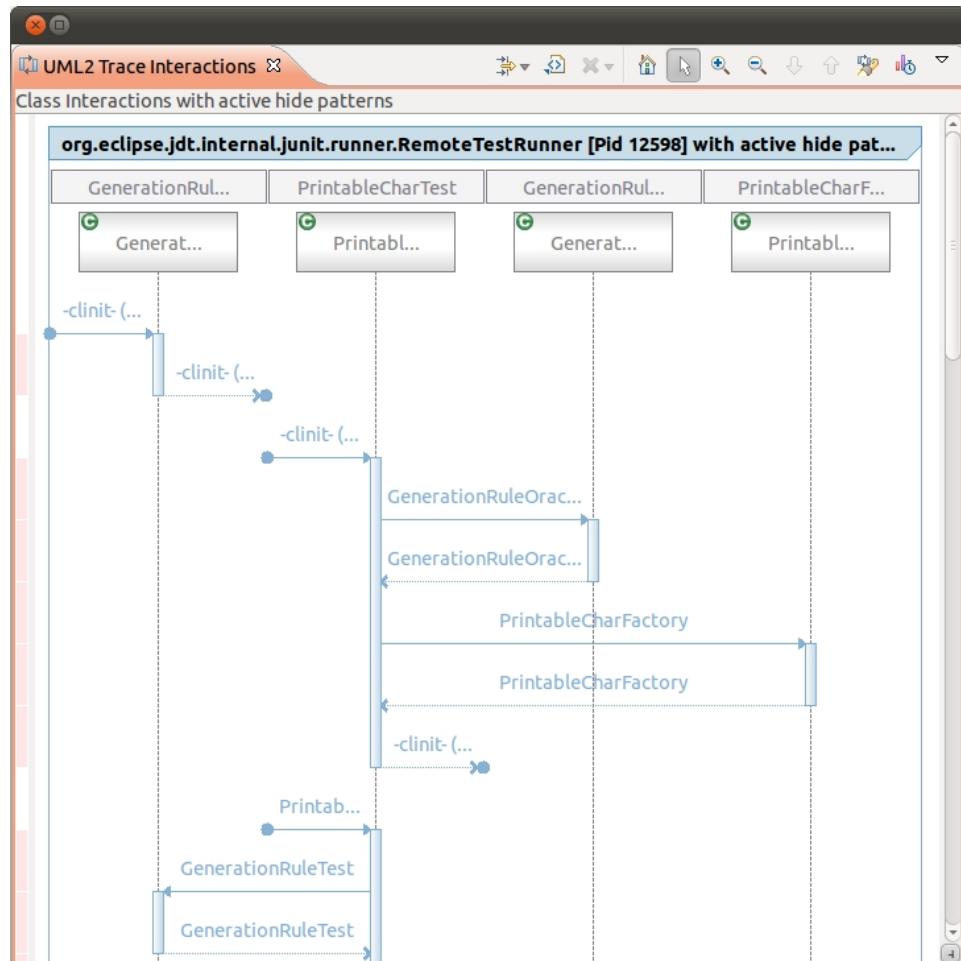


Figure 27.4: Eclipse TPTP Profiler call tree view

The next column shows the total proportion of time that was spent in each method call. In particular, the analysis in the column shows that testing done by the class is dominated by the `testMembership()` test method. In fact, this is unsurprising, since this method tests each and every candidate string to be generated.

Figure 27.4 illustrates a partial sequence diagram for the program when it was executed. The diagram shows the initial setup phase of the test class, in which the candidate password rule generator is constructed using a factory. The diagram shows an initial sequence of class initialisation calls, followed by instance constructor calls for the different objects under test.

Objects in the call sequence can be hidden or displayed as desired. When an object has been hidden, out-going and incoming calls from the sequence are still shown.

Eclipse TPTP Profiler  
– sequence diagram  
(824)

## 27.3 Source Code Obfuscation

Source code *obfuscation* is a deliberate technique to make source code harder to comprehend. In one sense, obfuscation is the opposite of source code comprehension and refactoring. The source code is changed so that it is more difficult for a human reader to follow, but is functionally equivalent to its previous behaviour.

Usually, this would seem like a bad idea. However, source code obfuscation has some legitimate uses, including:

Uses of source code obfuscation (825)

- protecting intellectual property rights;
- for entertainment and competition. The obfuscated C contest<sup>2</sup>, for example;
- to educate software developers in the need for clear code; and
- for providing test challenges to software reviewers.

There are several techniques for automatically obfuscating source code:

Obfuscation techniques (826)

- randomised identifier re-naming;
- operation re-ordering;
- layout corruption, removing white space, introducing unnecessary parentheses, for example;
- scope and visibility alteration, typically making everything global and public;
- insertion of extra redundant code; and
- control structure merging.

All of these techniques reduce the design quality of the target software, increasing the amount of coupling in the system and making it more difficult to isolate particular chunks of code for analysis and comprehension.

The source code shown below is credited to James Coplien, in the original version of these notes, and to Ian Phillipps by the obfuscated C contest as the winning entry in 1988. The program is written in the C programming language, and has been obfuscated to hide its true purpose.

Example code obfuscation (827)

---

<sup>2</sup><http://www.ioccc.org/>

```

#include <stdio.h>
main(t,_,a) char *a;{return !0<t?t<3?main(-79,-13,a+main(-87,1-_,
    main(-86,0,a+1)+a)):1,t<_?main(t+1,_,a):3,main(-94,-27+t,a)
    &&t==2?_<13? main(2,_+1,"%s %d %d\n"):9:16:t<0?t<-72?main(_,
    t, " @n' + ,#'*{*}w+/w#cdnr/+,{ }r/*de}+,*{*,/w{%,/w#q#n+,/#{
    l,+ ,/n{n+,/+#n+,/#\ ;#q#n+,/+k#;**+,/'r :d*3,{w+K w'K:+}e
    #' ;dq#'1 \ q#+d'K#!/+k#;q#'r}eKK#}w'r}eKK{nl}'/#;#q#n'){})#
    w'){}){nl}'/#n';d}rw' i;# \ ){nl}!/n{n#'; r{#w'r nc{nl}'/#{
    ,+'K {rw' iK{;[{nl}]/w#q#n'wk nw' \ iwk{KK{nl}}!/w{%'l##w#'
    ; :{nl}/*{q#'}ld;r'}{nlwb!/*de}'c \ ;:{nl}'-{}rw] '/+,}##'*#}
    nc,'#nw' /+kd'+e}+;#'rdq#w! nr' / ') }+}{rl#'{n' ')# \
    }'+}##(!/) :t<-50?_==a?putchar(31[a]):main(-65,_,a+1):
    main((*a=='/')+t,_,a+1) :0<t?main(2,2,"%s") :*a=='/' ||main(0,
    main(-61,*a, " !ek;dc i@bK'(q)-[w]*%n+r3#1,{}:\nuwloca-0;m .
    vpbks ,fxntdCeghiry"),a+1);}

```

The output from the program (the lyrics of the song '*The Twelve Days of Christmas*') is shown below.

```

On the first day of Christmas my true love gaave to me
a partridge in a peaar tree.

On the second day of Christmas my true love gaave to me
two turtle doves
and a partridge in a peaar tree.

On the third day of Christmas my true love gaave to me
three french hens, two turtle doves
and a partridge in a peaar tree.

On the fourth day of Christmas my true love gaave to me
four caalling birds, three french hens, two turtle doves
and a partridge in a peaar tree.

On the fifth day of Christmas my true love gaave to me
five gold rings;
four caalling birds, three french hens, two turtle doves
and a partridge in a peaar tree.
...

```

I've no idea how it works, however, it is a nice illustration of the difficulty of changing obfuscated, or poorly written code. In this case, I might want the program to print out the words 'Four colly birds' for the second verse ('colly' means common in colloquial English, and has been gradually been corrupted to 'calling' in modern versions of the song). However, I have little hope of finding where in the program I should make this change.

## Summary

Understanding a large, complex software base can take as much time as making any changes to it. There are numerous software engineering tools available to ease the software comprehension task, some as trivial as code formatting, some for identifying and removing redundant code.

## 27.4 Workshop: Software Comprehension with Eclipse

The purpose of this workshop is to practice applying the software comprehension tools, techniques and processes described in this chapter. The tools used are either integrated into the Eclipse Integrated Development Environment (IDE), or available as plugins.

The TPTP plugin may already be installed on the laboratory machines. If it is not, or you wish to use the plugin on your own machine, the installation instructions for the Eclipse Test and Performance Tools Platform (TPTP) plugin can be found on the Eclipse wiki:

[http://wiki.eclipse.org/Install\\_TPTP\\_with\\_Update\\_Manager](http://wiki.eclipse.org/Install_TPTP_with_Update_Manager)

Note that these instructions are for the Galileo release of Eclipse. If you rare using the Helios release, then change the release update URL to

<http://download.eclipse.org/releases/helios>

Once you have installed the TPTP, follow the tutorial at:

[www.eclipse.org/tptp/home/documents/tutorials/profilingtool/profilingexample\\_32.html](http://www.eclipse.org/tptp/home/documents/tutorials/profilingtool/profilingexample_32.html)

Finally, try running the profiling tool your own applications and coursework. In particular, try running the profiling tool against different versions of the software. Do you notice difference between the performance of the different application versions?

## 27.5 Exercises

1. Reconstruct the functionality of the following code.

```
package uk.ac.glasgow.psd.comprehension;

public class D {public static void main(String[] a)
    {int c= 042 ^ 0x42;while(c > 1) {int j= (c&~-2)
        ==0?c>>=1:(c += c << 1) + ++c; j= c+~-33+(c>=17?j
        ++<32?45:8:++j<5|12%c>0?~-1:12-~(-j++ +0100)); j
        += j==42?-c:j==043||c==(c&020)?111-j:j==2+j-c
        ?100-j:(j^c)&~-1;if(j % 8 == 5 & c % 5 < 3) j +=
        c+77;if((~c^010) == -1) j ^= (j<<1)^10;System.out
        .print((char)j);}}}
```

- (a) What are the beacons?
- (b) Do you recognise any keywords?
- (c) Do you recognise any chunks?

## Chapter 28

# Software Refactoring

### Recommended Reading

#### PLACE HOLDER FOR fowler00refactoring

Fowler's website on refactoring [www.refactoring.com](http://www.refactoring.com)

Software is easier to alter to respond to changes in the system environment, if the source code has been carefully maintained. Characteristics of well-maintained software source code include:

- effective use of a version control system;
- an automated test suite;
- a clear architecture and design;
- a simple and consistent coding style, enforced through automation where possible; and
- appropriate use of comments and other documentation.

A key discipline for maintaining the structure of a software system as it evolves is *refactoring*. Refactoring should be an-going process in a software development project, alongside the introduction of new features. Fowler [2000] defines a refactoring as:

*"Refactoring (noun): a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour."*

Fowler [2000]

and goes on to define the verb form (*to refactor*) as the application of a series of refactorings. This is a useful standard definition, but could be improved

Maintaining source code and system design (831)

Defining software refactoring (noun) [Fowler, 2000] (832)

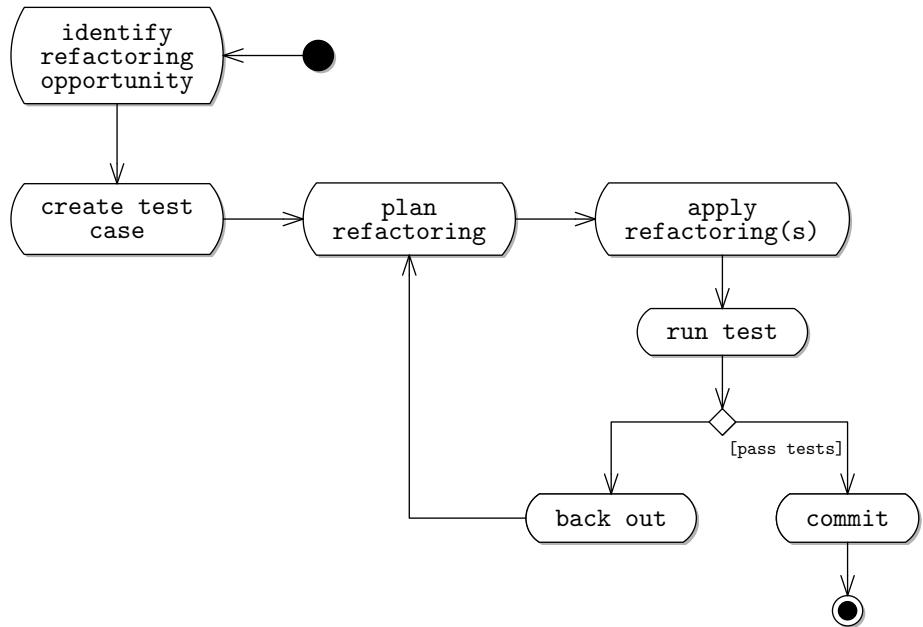


Figure 28.1: the refactoring process

to say “..while *minimising* changes in *functional* behaviour” (emphasis added). As we will see, refactoring often causes changes in the non-functional properties of a system, and indeed, can also cause changes in the functional behaviour or specification.

## 28.1 The Refactoring Process

Refactoring can be thought of as a form of source code cleanup, with the addition of a well defined process for performing the change. Figure 28.1 illustrates the process of refactoring. The process begins with the identification of opportunities to refactor. Fowler calls indicators of these opportunities ‘bad smells’ in the program source code. A ‘bad smell’ is a pattern in the source code indicating poor structure or inappropriate coupling between modules. More detail is given on types of bad smell and their implications for refactoring in Section 28.2.

Once an opportunity to refactor is identified, it is important to create tests for the affected classes, using an automated test harness framework, such as JUnit. The test cases are important because they will be used to show that the functional behaviour of the refactored code has not changed as a result of applying any refactorings. Of course, in well maintained code, a class or package will already have a well established suite of tests that can be used for this purpose.

Next the refactoring is planned, based on the guidance provided by the bad

smells. Once the revised design has been established, it can be applied, often using automatic refactoring tools that are provided as part of many Integrated Development Environments (IDEs) such as Eclipse.

At this point, the tests that were developed previously should be rerun. If the tests are successful, then it is appropriate to commit the new design and source code to the main project. This may involve merging the change into the main development trunk of the project source code repository. Having completed the refactoring, new opportunities for improving the design may also be identified. For example, if extract method is applied to a method with a large body, it may then also be appropriate to move that new method to another class.

If one or more of the tests fail, this indicates that the refactoring(s) have altered the functional behaviour of the altered class. It is necessary here to back out of (reverse) the changes. Additional planning and analysis can then take place to understand what unexpected effect the refactoring(s) had. Once the slip has been established, the revised refactoring(s) can be applied. This process continues iteratively, until a refactored system passes the pre-prepared tests.

## 28.2 Bad Smells

The first step in the refactoring process is to identify opportunities for refactoring. Refactoring should be treated as a parallel on-going process, that occurs alongside other software development activities. This means that you should be looking for opportunities to refactor when you are:

When to refactor  
(834)

- implementing new functionality;
- correcting a defect;
- doing a code review; and (most importantly)
- when you are trying to understand how a software artifact works.

While you undertake these tasks, you should also be observant for examples of poor software design in the source code. Fowler [2000] refers to these as *bad smells*. These are clues that the source code could be improved in some fashion.

The bad smells identified by Fowler are summarised below. The grouping of smells by category is not in the original book, but can be useful in recalling the different sources of stink. Consequently, bad smells can arise from:

Fowler [2000]'s 'Bad smells' (835)

- **cloning**

*duplicate code*: Source code that has been *cloned* rather than reused makes maintenance harder. If you find a bug in the original, you have to fix all the clones as well.

- **complex structures**

*long method*: The behaviour of long sequences of instructions is harder to understand, because all of the details of the implementation are presented to the reader at once. In general, methods that can't be read on a few lines in an editor are too long.

*large class*: Indicates that too many responsibilities have been allocated to a single class. It is also possible that the large class contains duplicate code.

- **variables and parameters**

*long parameter list*: Imposes *stamp coupling* on the design of the method, because each time the parameter list changes, every call of the method must also be changed. A long parameter list also indicates that a method that is used in different ways, depending on how it is called.

*feature envy*: Indicates that a method is in the wrong class, because it obtains most of its data from another class. If either the data or the accessing method change, then so will the other.

*data clumps*: Occurs when the same, independently sourced, data is used together in different locations in a system. This might be identifiable from duplicate parameter lists for methods, or from duplicate blocks of query method calls. Both of these examples are also indicators of cloning.

*primitive obsession*: Indicated by a preference for using primitive data types for representing more complex values with additional semantics. This is problematic because the semantics and supplementary properties of a data type is not explicit when it is represented as a primitive value, and has to be maintained independently of the data. A date in the Gregorian calendar could be represented by three integers, for example (one each for day, month and year). However, this means the relationships between these values has to be maintained elsewhere in the system.

*temporary field*: These are fields that are used as variables in method bodies, but aren't always needed. Fields should only be used to record information about an object's state that must persist between method calls.

- **making changes**

*divergent change*: Another indicator that a class has too many responsibilities, identifiable when a single class must be altered in different ways to respond to different changes in the system's environment. This suggests that the class has two or more different sets of responsibilities.

*shotgun surgery*: The opposite of divergent change, identifiable when a change in the environment necessitates a number of different changes in several different classes in a system. This makes maintenance more time consuming, and easier to get wrong.

*parallel inheritance hierarchies*: Identifiable by dependencies between two inheritance hierarchies, such that a change in one hierarchy causes a change in another, increasing coupling and maintenance costs.

- **control structures and polymorphism**

*switch statements*: Indicates a reluctance to exploit object oriented polymorphism and increases maintenance costs, because each time a new option is introduced into the system, a flag and additional line must be added to the switch.

*refused bequest*: Sometimes a sub-class doesn't need all of the operations provided by a super class. This means that all the methods of the super class are unnecessarily coupled to the sub-class.

*alternative classes with different interfaces*: This is a 'smell' because it means that the benefits of object-oriented polymorphism cannot be exploited. The two or more classes must be manually selected for use in the code.

- **design uncertainty**

*lazy class*: This can be a difficult smell to identify, given that many of the smells are about classes that do *too much*. However, lazy classes, that have insufficient *independent* responsibilities can be unnecessarily expensive to maintain.

*speculative generality*: Although abstracting over concrete implementation details is generally a good idea, it can be tempting to attempt to move all design decisions into the configuration for a software system. At its most extreme, this phenomenon is known as the *inner platform* anti-pattern, because the system is designed as a software platform that must be configured, on top of an existing software platform.

*incomplete library class*: This is less of a smell, than a potential cause of other smells. It occurs when it is discovered that a class from a library doesn't support all the operations needed by the client. In addition, the library class can't be altered, because it is used in many other projects.

- **delegation**

*message chains*: Occurs when one object accesses a data item through a series of intermediary objects. The requesting object is therefore bound to all the other objects in the chain.

*middle man*: Occurs when one object acts as an information broker to another object, but doesn't actually provide any information itself. There may be good reasons for this, if the broker is a proxy, for example. If the

object is just passing on information, however, it may be better for the communication to occur directly.

*inappropriate intimacy*: Also known as the *object-orgy* anti-pattern. This smell is the opposite of accessing information via middlemen or chains. Instead, all objects freely interact with the properties of other objects, causing an ‘orgy’ of couplings between them.

*data class*: Classes should generally have behavioural responsibilities as well as data items. If a class lacks behaviour, this is often because the behaviour has been implemented somewhere else.

Many of the bad smells and refactoring remedies are illustrative of the principles of good design described in the OOSE lecture notes, and in the course text [Lethbridge and Laganière, 2005]. These principles embody the adage “good design has low coupling and high cohesion”. Unsurprisingly, refactoring is concerned with reducing the coupling and increasing the cohesion in a software application.

A final bad smell described by Fowler [2000] is the excessive use of comments to explain unnecessarily complex source code. The comments are indicative of poor structure that can be improved through the application of refactoring(s). This process helps to create *self documenting* code. Once the code has been refactored, its purpose and functionality become much more self-evident, reducing the need for supplementary documentation. This excessive documentation can be largely removed when the refactoring process is complete.

Figure 28.2 illustrates an example of what I would consider excessive documentation given by Brooks [1995]. The documentation is arguably necessary due to the limits of the programming languages available at the time. However, the excessive documentation makes understanding program flow harder, not easier.

Excessive comments  
vs. self documenting  
code (836)

Types of refactorings  
(837)

## 28.3 Applying Refactorings

Once a ‘bad smell’ has been identified, it is necessary to identify a suitable refactoring to remedy the problem. There are a number of different categories of refactoring.

- **fixing methods**

- extract method ↔ inline method
  - replace method with method object

- **moving functionality**

- move method, move field
  - extract class ↔ inline class

```

//QLT4 JOB ...

QLTSRT7: PROCEDURE (V);

/*********************************************************************
/*A SORT SUBROUTINE FOR 2500 6-BYTE FIELDS, PASSED AS THE VECTOR V. A      */
/*SEPARATELY COMPILED NOT MAIN PROCEDURE, WHICH "UST USE AUTOMATIC CORE      */
/*ALLOCATION.                                                               */
/*
/*THE SORT ALGORITHM FOLLOWS BROOKS AND IVERSON, AUTOMATIC DATA PROCESSING ,*/
/*PROGRAM 7.23 , P. 350. THAT ALGORITHM IS REVISED AS FOLLOWS:          */
/* STEPS 2-12 ARE SIMPLIFIED FOR M=2.                                     */
/* STEP 18 IS EXPANDED TO HANDLE EXPLICIT INDEXING OF THE OUTPUT VECTOR.   */
/* THE WHOLE FIELD IS USED AS THE SORT KEY.                                */
/* MINUS INFINITY IS REPRESENTED BY ZEROS.                                 */
/* PLUS INFINITY IS REPRESENTED BT ONES.                                    */
/* THE STATEMENT NUMBERS IN PROG. 7.23 ARE REFLECTED IN THE STATEMENT     */
/* LABELS OF THIS PROGRAM.                                                 */
/* AN IF-THEN-ELSE CONSTRUCTION REQUIRES REPETITION Of A FEW LINES.       */
/*
/*TO CHANGE THE DIMENSION OF THE VECTOR TO BE SORTED, ALWAYS CHANGE THE    */
/*INITIALIZATION OF T. IF THE SIZE EXCEEDS 4096, CHANGE THE SIZE OF T, TOO.*/
/*A MORE GENERAL VERSION WOULD PARAMETERIZE THE DIMENSION OF V           */
/*
/*THE PASSED INPUT VECTOR IS REPLACED BY THE REORDERED OUTPUT VECTOR.      */
/********************************************************************

/* LEGEND (ZERO-ORIGIN INDEXING) */

DECLARE
  (H,                      /*INDEX FOR INITIALIZING          */
   I,                      /*INDEX OF ITEM TO BE REPLACED    */
   J,                      /*INITIAL INDEX OF BRANCHES FRON NODE I */
   K) BINARY FIXED,        /*INDEX IN OUTPUT VECTOR          */

  (MINF,                  /*MINUS INFINITY                 */
   PINF) BIT (48),         /*PLUS INFINITY                  */

  V (*)      BIT (*),     /*PASSED VECTOR TO BE SORTED AND RETURNED */

  T (0:8190) BIT (48): /*WORKSPACE CONSISTING OF VECTOR TO BE SORTED, FILLED*/
                        /*OUT WITH INFINITES, PRECEDED BY LOWER LEVELS */
                        /*FILLED UP WITH MINUS INFINITES */

/*
 * NOW INITIALIZATION TO FILL DUMMY LEVELS, TOP LEVEL, AND UNUSED PART OF TOP*/
/* LEVEL AS REQUIRED. */

INIT: MINF= (48) '0'B;
      PINF= (48) '1'B;

      DO L= 0 TO 4094; T (L) = MINP; END;
      DO L= 0 TO 2499; T (L+4095) = V(L); END;
      DO L=6595 TO 8190; T(L) PINP; END;

K0: K = -1;
K1: I = 0;                /*
K3: J = 2*I+1;             /*SET J TO SCAN BRANCHES FROM NODE I.      */
K7: IF T(J) <= T(J+1)    /*PICK SMALLER BRANCH               */
   THEN                   /*
   DO:                   /*
K11:   T(I) = T(J); /*REPLACE
K13:   IF T(I) = PINF THEN GO TO K16; /*IF INFINITY, REPLACEMENT-----|| */
                           /* IS FINISHED
K12:   I = J;           /*SET INDEX FOR HIGHER LEVEL
   END;                  /*
   ELSE                   /*
   DO:                   /*
K11A:   T(I) = T(J+1); /**
K13A:   IF T(I) = PINF THEN GO TO K16; /**
K12A:   I = J+1;        /*
   END;                  /*
K14: IF 2*I < 8191 THEN GO TO K3; /*GO BACK IF NOT ON TOP LEVEL -----+|| */
K15: T(I) = PINF;          /*IF TOP LEVEL, FILL WITH INFINITY      || */
K16: IF T(0) = PINF THEN RETURN; /*TEST END OF SORT <-|- | */
K17: IF T(0) = MINF THEN GO TO K1; /*FLUSH OUT INITIAL DUMMIES -----+| */
K18: K = K+1;              /*STEP STORAGE INDEX      | */
   V(K) = T(0);           /*STORE OUTPUT ITEM      -----+| */
END QLTSRT7;

```

Figure 28.2: Brooks on self-documenting code, adapted from [Brooks, 1995, pp173]

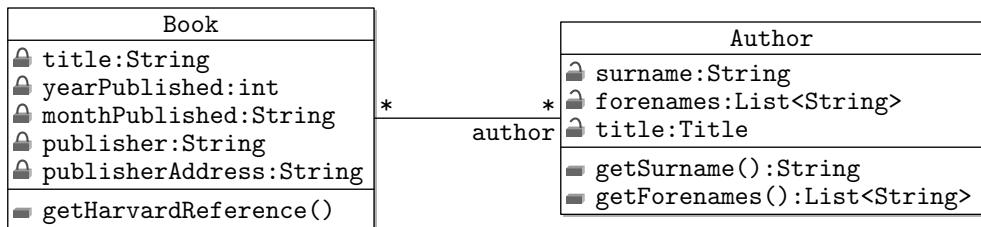


Figure 28.3: the Book and Author classes

- **organising data**

- encapsulate field
- replace data value with object
- replace magic number with symbolic constant

- **simplifying method calls**

- parameterise method ↔ remove parameter
- use parameter object

- **simplifying conditions**

- decompose conditional
- consolidate duplicate conditional fragments
- replace conditional with polymorphism

- **reorganising classes**

- pull up method → push down method
- pull up field → push down method
- extract superclass
- extract subclass
- collapse hierarchy

Table 28.1 summarises the bad smells described by Fowler [2000] and the proposed remedies. We will use this mapping investigate the application of refactorings as an illustrative example of the general technique.

We will now consider applying the refactoring process to the branch library system. Figure 28.3 illustrates a partial class diagram for the branch library management system, concerning the relationship between books and their authors.

The source code for the attributes of the Book class are shown below.

smell	strategies
duplicated code	extract method → (pull up method, form template method)   substitute algorithm
long method	extract method → (introduce parameter object, replace temp with query)   replace method with method object → extract method
large class	extract class, extract subclass
long parameter list	replace parameter with method, preserve whole object
divergent change	extract class
shotgun surgery	move method, move field, inline class
feature envy	move method, extract method
data clumps	extract class → (introduce parameter object, preserve whole object)
primitive obsession	replace data value with object, replace type code with class, extract class, introduce parameter object
switch statements	replace type code with subclasses → replace conditional with polymorphism
parallel inheritance hierarchies	move method, move field
lazy class	collapse hierarchy, inline class
speculative generality	collapse hierarchy, inline class, remove parameter, rename method
temporary field	extract class, introduce null object
message chains	hide delegate, extract method → move method
middle man	remove middle man, inline method
inappropriate intimacy	move method, move field, extract class, hide delegate, change bidirectional association to unidirectional, replace inheritance with delegation
alternative classes with different interfaces	rename method, move method, extract superclass
incomplete library class	introduce foreign method, introduce local extension
data class	encapsulate field, extract method, move method, hide method
refused bequest	push down method, push down field, replace inheritance with delegation

Table 28.1: bad smells and strategies for refactoring [Fowler, 2000]

```

private List<Author> authors = new ArrayList<Author>();
private int yearPublished;
private String monthPublished;
private String title;
private String publisher;
private String publisherAddress;

```

The source code shown below is for the getHarvardReference() method in the Book class in the branch library management system. The purpose of the method is to format the details of the book in the Harvard referencing style.

```

String result = "";

for (Author author: authors){
    String surname = author.getSurname();
    result+=surname + ", ";

    List<String> forenames = author.getForenames();

    for (String forename: forenames)
        result += forename.charAt(0)+" . , ";
}
result+=" ("+yearPublished+") ";

result+=title+" . ";

result+=publisherAddress+": "+publisher+".";
return result;
}

```

Example – Book.  
getHarvardReference  
(  
(840)  
Some bad smells in  
the  
getHarvardReference  
(  
) method  
(841)

Example – BookTest  
(842)

Looking at the method, we can find several examples of bad smells:

- it is a *long method*, containing a mixture of queries, processing and string composition;
- the method has *primitive obsession*, particularly in the yearPublished and monthPublished attributes; and
- there is a faint whiff of *data clumping*, since it is likely that the publisher's name and address will be used a lot together.

The next stage is to construct a test case that checks whether any changes made alter the correctness of the method. The code for the BookTest class, which contains a method for testing getHarvardReference() is shown below.

```

private Book book;

@Before
public void setUp() throws Exception {

```

```

List<String> forenames = new ArrayList<String>();
forenames.add("Ian");
Author author =
    new Author("Sommerville", forenames, Title.PROFESSOR);

List<Author> authors = new ArrayList<Author>();
authors.add(author);

book = new Book(authors, 2008, "September",
    "Software Engineering",
    "Addison Wesley",
    "London");

}

@Test
public void testGetHarvardReference() {
    Assert.assertEquals(
        "Sommerville, I., (2008) Software Engineering." +
        " London: Addison Wesley.",
        book.getHarvardReference());
}

```

Now we can start planning some refactorings to the Book class, based on the ‘bad smells’ that have been identified:

Refactoring plan (843)

- apply *extract method* to the code for formatting the author’s name and *move method* to the resulting method, so that it is placed in the Author class, to deal with the *long method* smell;
- apply *replace data value with object* to the combined yearPublished and monthPublished attributes, to deal with the *primitive obsession* smell; and
- apply *extract class* to the attributes concerning the book publisher. Also, apply *introduce parameter object* to the relevant String arguments to the constructor. Note that this may introduce a *lazy class*, so it might be better to use *extract inline class*.

While performing the refactoring, several new lines of code are introduced to the getHarvardReference() method, for dealing with extracting the year of publication from the published attribute. These are also refactored using *extract method*. Similarly, the *extract method* and *move method* refactorings can be applied to the code that now accesses and formats the publisher’s details.

The refactoring also causes several changes to the BookTest setup method, because the constructor call for the Book class must be altered. This is okay, because the method under test, getHarvardReference(), has not been altered.

Figure 28.4 illustrates the revised class diagram for the Book class and its companions. The revised code for the Book class is shown below.

Revised Book class (844)

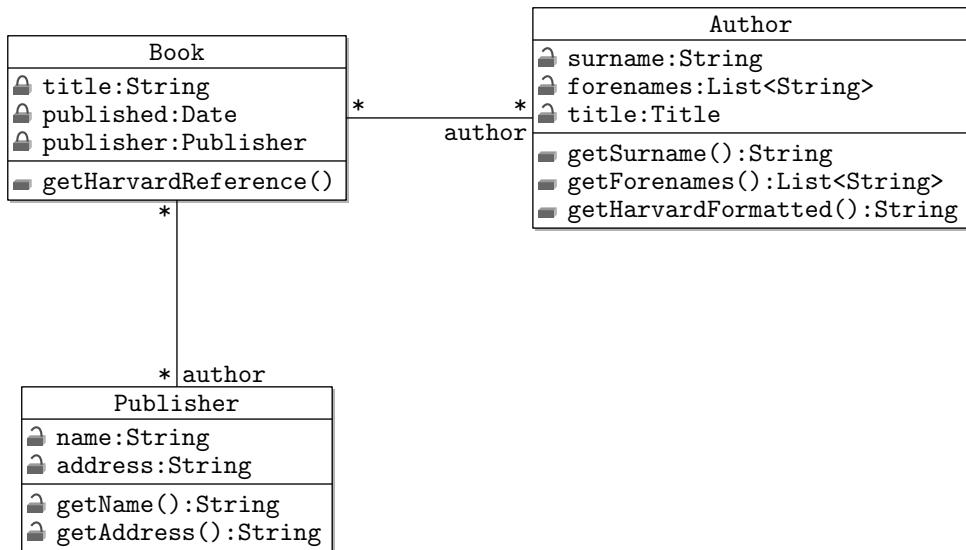


Figure 28.4: the revised book class and companions

```

private List<Author> authors = new ArrayList<Author>();
private Date published;
private String title;
private Publisher publisher;

public String getHarvardReference(){
    String authors = getHarvardFormattedAuthors();
    int yearPublished = getYearPublished();
    String publisherDetails = publisher.getHarvardFormatted
        ();

    String result = authors+" ("+
    yearPublished+") "+title+". "+publisherDetails;
    return result;
}

private String getHarvardFormattedAuthors(){
    String result = "";
    for (Author author: authors)
        result += author.getHarvardFormatted();
    return result;
}

private int getYearPublished() {
    Calendar c = Calendar.getInstance();
    c.setTime(published);
    int yearPublished = c.get(Calendar.YEAR);
    return yearPublished;
}

```

Many software tools, such as IDEs have integrated tool support for automatic refactoring. Eclipse, for example, supports (contextualised) operations for automatically:

Revised Book class methods (845)  
Automated refactoring (846)

- Rename
- Move
- Extract Interface
- Extract Superclass
- Pull Up
- Push Down
- Extract Class
- Change Method Signature
- Inline
- Introduce Parameter Object
- Introduce Indirection
- Infer Generic Type Arguments
- Generalize Declared Type
- Encapsulate Field

There is a workshop on using Eclipse for refactoring in Section 28.5.

## 28.4 Limits of Refactoring

To summarise the previous section, the refactoring process is a powerful mechanism for managing design quality as a software application evolves. However, refactorings do themselves cause the software evolve and consequently can and do alter the observable behaviour of software systems. Refactorings can cause observable changes to a software system in either the:

Limits of refactoring (847)

- non-functional properties; or
- the application programming interface (API) of a system.

## Refactorings that alter APIs (848)

Refactorings which causes changes to non-functional properties may be either beneficial or detrimental. A refactoring might decrease a system's overall memory footprint, whilst simultaneously reducing the system's response time. This may happen because the refactoring reduces the amount of cloning in a software system. Unfortunately, this may increase the number of method calls that must be made, increasing execution time for a transaction. These considerations are particularly important for mobile, embedded or realtime applications in which computing resources may be limited. However, a clear design is usually easier to *tune*, so that over the long term, the required non-functional properties of a system are easier to achieve.

In addition to changing non-functional behaviour, some refactorings require that an application programming interface (API) to a software module be altered. The API is the set of public operations and attributes of one module that can be accessed by other modules in a system at runtime. In object-oriented systems, every object has an API defined by the public members (operations and attributes) of its class. Changes to an API happens when:

- a class member is moved;
- a class member is renamed;
- the visibility of a class member is changed;
- the list of parameters to an operation is changed;
- the return type of an operation is changed; or
- the exceptions that may be raised are changed.

This may not be a significant problem if the API is for a class that is only accessed within a software system. However, some APIs must be exposed to users of the system. Fowler [2000] refers to these as *published* interfaces, because they have been exposed and documented for use by external clients of the entire software system. Consequently, every user of the software system is coupled to the API specification. The version of the system used is said to be a *dependency* for all its users.

If the API specification is changed as a result of the refactoring process, then the part of the old API that has changed becomes *deprecated* and must either be discarded, or retained in the new version of the system while the dependent systems are updated. Typically, the development team for a software system that has dependents issues a *migration plan* explaining how users can adapt their own systems to the new API.

## Summary

Refactoring is a key practice that should occur alongside other software activities. Refactoring is a formalised form of code cleanup, in which changes are made

according to a well specified process and with respect to a pre-defined plan. Refactoring opportunities are detected through the occurrence of 'bad smells' in code. Each bad smell can be remedied through the application of one or more refactorings. Applying a refactoring may lead to further opportunities to refactor.

## 28.5 Workshop: Refactoring with Eclipse

Consider the source code shown below provides of a poorly structured software application, written in Java. The software has been placed on Moodle in a zip archive under 'Lectures'. Two versions are available. One version is the unformatted, obfuscated original. The other is the end product of comprehension and refactoring.

Recovering structure –  
example (849)

```
import java.awt.*; import java.util.*; public class F  
extends Frame{Date D=new Date();void T(Date d){D=d;  
repaint();}double P=Math.PI,A=P/2,a,c,U=.05;int W,H,m,R;  
double E(int a,int u){return(3*P/2+2*P*a/u)%(2*P);}void N  
(Graphics g,double q,double s){g.fillPolygon(new int []{H  
s,q},H(U,q+A),H(U,q+3*A)},new int []{J(s,q),J(U,q+A),J(U,q  
+3*A)},3);}public void paint(Graphics g){Color C=  
SystemColor.control;g.setColor(C);g.fillRect(0,0,W=size()  
.width,H=size().height);W-=52;H-=52; R=Math.min(W/2,H/2);  
g.translate(W/2+25,H/2+36);g.setColor(C.darker());for(m  
=0;m<12;++m){a=E(m,12);g.drawLine(H(.8),J(.8),H(.9),J(.9)  
);}m=D.getMinutes();N(g,E(D.getHours())*60+m,720),.5);N(g,  
E(m,60),.8);N(g,E(D.getSeconds(),60),.9);}int H(double y)  
{return(int)(R*y*Math.cos(a));}int H(double y,double q){a  
=q;return H(y);}int J(double y){return(int)(R*y*Math.sin(a));}  
int J(double y,double q){a=q;return J(y);}public  
static void main(String [] _){throws Exception{F c=new F();c  
.resize(200,200);c.show();for();}{c.T(new Date());Thread.  
sleep(200);}}}
```

Fortunately the code may be amenable to comprehension and restructuring, so we may be able to recover its functionality relatively easily.

Contextualised refactoring tools are integrated into the Eclipse IDE. You can access any refactoring for a fragment of source code by highlighting the fragment, right clicking the mouse button. This will cause a context menu to appear. From there, navigate to 'Refactor' and then choose the refactoring you wish to apply.

For example, to rename the `Clock` class to `ClockFace` select the `Clock` identifier in the class declaration. Then right click the mouse button, choose 'Refactor', then 'Rename...'. A tooltip will appear in the editor next to the `Clock` identifier with the description 'Enter new name, press Enter to refactor'. Type the new name `ClockFace` directly into the editor and press return.

You may get a warning dialogue explaining that changing the name of the class may break any external dependencies such as scripts. You can ignore this warning for now and click 'Continue' (What type of problem associated with refactoring is the dialogue warning you about?). All references to the `Clock` class are updated to the new name in the source file.

Eclipse can also be used to automatically format source code. To do this, open a source file and choose 'Source → Format' from the window menu.

Alternatively, use the keyboard shortcut 'Control, Shift and F'.

For the tutorial follow the steps taken below to refactor the F.java source file into the Clock.java file, using the automated refactoring tools wherever possible.

The first step is to apply some formatting to the source code structure, using the features of an automated tool such as Eclipse. The code shown below shows the result of this step.

```
import java.awt.*;
import java.util.*;

public class F extends Frame {
    Date D = new Date();

    void T(Date d) {
        D = d;
        repaint();
    }

    double P = Math.PI, A = P / 2, a, c, U = .05;
    int W, H, m, R;

    double E(int a, int u) {
        return (3 * P / 2 + 2 * P * a / u) % (2 * P);
    }

    void N(Graphics g, double q, double s) {
        g.fillPolygon(new int[] { H(s, q), H(U, q + A), H(U, q
            + 3 * A) },
            new int[] { J(s, q), J(U, q + A), J(U, q + 3 * A) },
            3);
    }

    public void paint(Graphics g) {
        Color C = SystemColor.control;
        g.setColor(C);
        g.fillRect(0, 0, W = size().width, H = size().height);
        W -= 52;
        H -= 52;
        R = Math.min(W / 2, H / 2);
        g.translate(W / 2 + 25, H / 2 + 36);
        g.setColor(C.darker());
        for (m = 0; m < 12; ++m) {
            a = E(m, 12);
            g.drawLine(H(.8), J(.8), H(.9), J(.9));
        }
        m = D.getMinutes();
        N(g, E(D.getHours() * 60 + m, 720), .5);
        N(g, E(m, 60), .8);
```

```

    N(g, E(D.getSeconds(), 60), .9);
}

int H(double y) {
    return (int) (R * y * Math.cos(a));
}

int H(double y, double q) {
    a = q;
    return H(y);
}

int J(double y) {
    return (int) (R * y * Math.sin(a));
}

int J(double y, double q) {
    a = q;
    return J(y);
}

public static void main(String[] _) throws Exception {
    F c = new F();
    c.resize(200, 200);
    c.show();
    for (;;) {
        c.T(new Date());
        Thread.sleep(200);
    }
}
}

```

This step reveals immediately that:

- the program has a graphical user interface because it extends the class `Frame` from the `java.awt` package;
- there is an unused attribute `c` in the `F` class that can be removed; and
- the program has a `main` method that constructs a new instance of type `F` and then passes the instance a new `Date` instance every 200 milliseconds, using the `T(Date)` method.

The  
`main(String[] _)`  
method (850)

The `T(Date d)`  
method (851)

Following the method call, we can see that the `Date` instance is assigned to an instance attribute (`d`), so we could rename this method to  `setDate()` and the attribute and parameter to `date`. Automatic refactoring tools in IDEs such as Eclipse can be helpful here.

The next step is to investigate the `paint(Graphics g)` method, which is called when `repaint()` is called. Looking at the attributes `W` and `H`, we can see

that they are used to store the width and height of the application frame, so we can also rename these. Also notice that the renamed attributes are only used in the method where they are declared, so it is probably better to declare them as method variables.

The same applies to the `m` attribute, which is being used to store a `minutes` attribute from the `Date` instance, and the `R` attribute which is being used to store a maximum radius, if a circle were to be drawn in the frame. There is a slight problem here, that `m` is used for a different purpose, as an index in the for loop. This index needs to be renamed, by convention to `i`.

Next lets consider the `E` method, which, given the modulo operation, must return a proportion of  $2\pi$ , so an angle in radians. The first thing that can be done is to rename the attribute `a`, to make it clear that is represents an angle. The method can also be renamed to make its purpose clear.

Given that `Math.PI` is a standard constant in the Java language, it is a probably unnecessary to define a separate attribute to store this value in the `F` class. If the `Math.` part of the static variable is too cumbersome, we could use a static import instead. Although this does couple the class to the definition of `pi` in the Java SDK, this probably isn't going to change anytime soon.

Now consider the four methods `H(double)`, `H(double, double)`, `J(double)`, and `J(double, double)`, which have similar structures. Each method is used to calculate the length of an opposite or adjacent side of a triangle when given a length for a hypotenuse proportionate to the radius of the circle, based on the `angle` attribute. Another purpose for this calculation is to obtain the `x` and `y` coordinates of a 2D vector represented as a length and an angle. So, we can rename these two pairs of methods appropriately.

In addition, note that all the methods are coupled to the value of `angle`, which can be set externally to the method, or from within the method. This is unfortunate, because the flow of control is confused, and the value of `angle` may change outwith the methods that depend on it. It would be better to pass the `angle` attribute to each method and perform the calculation directly. The same is true of the `radius` attribute, however, this is used in combination with a constant, so must be left alone for the time being. This now makes the function of the for loop in the main `paint()` method much clearer. The loop is used to draw twelve lines on the `Graphics` object for the frame at twelve different angles.

The final method to examine is the `N()` method, whose only purpose is to draw a filled polygon. Initially, we can rename the input parameters, since we know how they will be used in the `calculateX()` and `calculateY()` methods.

Next, we can see that the polygon has three points, one at the end of the specified vector and two very close to the origin (parameterised by the constant `U` and angle offset by `A`, so `N` is used to draw triangles of different lengths and different angles. Regarding `A`, it can be observed that it is set to  $\pi/2$ , or a quarter of circle. So, the offset expression for the final coordinate can be simplified to `angle - A`.

The  
paint(Graphics g)  
method (852)

The E(int a, int u)  
method (853)

The H(double y,  
double q) methods  
(854)

The N(Graphics g,  
double q, double s)  
method (855)

The

paint(Graphics g)  
method (857)

Returning to the paint() method reveals the purpose of the N() method. The method is used to draw three triangles at the appropriate angles for seconds, minutes and hours of the current date. The N() method and class can now both be renamed.

Finally, the method order can be reorganised and most made private. Some of the method calls used in the original application have been deprecated, so these are replaced as well. The revised version of the class is shown below.

```
import java.awt.*;
import java.util.*;

import static java.lang.Math.PI;

public class Clock extends Frame {

    private double offset = PI / 2, U = .05;

    private int radius;

    private Date date = new Date();

    public void setDate(Date date) {
        this.date = date;
        repaint();
    }

    public void paint(Graphics g) {
        Color C = SystemColor.control;
        g.setColor(C);

        int width = getSize().width;
        int height = getSize().height;

        g.fillRect(0, 0, width, height);
        width -= 52;
        height -= 52;
        radius = Math.min(width / 2, height / 2);

        g.translate(width / 2 + 25, height / 2 + 36);
        g.setColor(C.darker());

        for (int i = 0; i < 12; ++i) {
            double angle = calculateAngle(i, 12);
            g.drawLine(
                calculateX(.8, angle),
                calculateY(.8, angle),
                calculateX(.9, angle),
                calculateY(.9, angle));
        }
    }
}
```

```

Calendar cal = Calendar.getInstance();
cal.setTime(date);

int hours = cal.get(Calendar.HOUR);
int minutes = cal.get(Calendar.MINUTE);
int seconds = cal.get(Calendar.SECOND);

drawHand(g, calculateAngle(hours*60 + minutes, 720), .5)
;
drawHand(g, calculateAngle(minutes, 60), .8);
drawHand(g, calculateAngle(seconds, 60), .9);
}

private double calculateAngle(int numer, int denom) {
    return (3 * PI / 2 + 2 * PI * numer / denom) % (2 * PI)
;
}

private void drawHand(
    Graphics g, double angle, double length) {

    g.fillPolygon(
        new int[] {
            calculateX(length, angle),
            calculateX(U, angle + offset),
            calculateX(U, angle - offset)},
        new int[] {
            calculateY(length, angle),
            calculateY(U, angle + offset),
            calculateY(U, angle - offset)},
        3
    );
}

private int calculateX(double length, double angle) {
    return (int) (radius * length * Math.cos(angle));
}

private int calculateY(double length, double angle) {
    return (int) (radius * length * Math.sin(angle));
}

public static void main(String[] _) throws Exception {
    Clock c = new Clock();
    c.resize(200, 200);
    c.show();
}

```

```
while(true) {  
    c.setDate(new Date());  
    Thread.sleep(200);  
}  
}  
}
```

Following the comprehension and refactoring process, we discover that the application is a clock! When you have finished, consider if any other refactorings might be appropriate. Prepare a refactoring plan for these and apply them to the file.

# Chapter 29

# Re-engineering Legacy Systems

## Recommended Reading

PLACE HOLDER FOR sommerville10software

PLACE HOLDER FOR brodie93darwin

### 29.1 Legacy Systems

Software evolution will occur at different rates for different parts of the software system. Additionally, different software systems will also evolve at different rates, as requirements change, or new platforms and technologies become available. Some software systems, once implemented and deployed will become *mission critical* for an organisation. This means that the continued survival of the organisation is entirely dependent on the continued availability of the system. The system implementation may encapsulate important data, for example, or realise parts of key business processes.

Eventually, such systems become known as *legacy systems*, because they have been bequeathed by a previous generation of software engineers be to maintained and preserved. Legacy systems are characterised by:

- size, consisting of millions of lines of source code, or thousands of function points. Larger systems are generally more problematic because they are harder to comprehend and change;
- age of typically more than ten years, although a system does not need to be used for this long to become a legacy;
- non-replaceable dependencies, particularly hardware components that are no longer manufactured NASA apparently used eBay to acquire spare parts for the shuttle programme<sup>1</sup>;

Characteristics of legacy systems (859)

---

<sup>1</sup><http://www.nytimes.com/2002/05/12/us/for-parts-nasa-boldly-goes-on-ebay.html>

- unsupported dependencies such as software platforms and frameworks that have been deprecated. This may include deprecated operating systems, database management systems, middleware frameworks, graphics libraries or other software components;
- poor or absent documentation due to previous evolutions of the software;
- a lack of expert knowledge regarding the system within the owning organisation as the original developers have moved on in their careers, or even retired;
- poor integration with other systems; and
- the presence of supplementary systems or manual ‘work arounds’ to implement desired new functionality.

The high value placed on the system’s continued operation mean that there are few opportunities to undertake maintenance or update the system. Consequently, future maintenance activities become increasingly expensive and difficult and therefore increasingly unattractive as investment options.

Full replacement of the system may also be considered high risk, because the survival of the organisation may be dependent on the success of the replacement project. In addition, it may be necessary to implement a migration to the new system which does not permit any down-time during the transfer.

A significant factor in the cost of managing legacy systems is the loss of knowledge about the software system within the organisation. The knowledge may be formally stored by the organisation, or held informally by the organisation’s employees.

The loss of knowledge can be due to:

- lack of documentation (the knowledge is *tacit*), or documentation is inaccurate;
- loss of staff due to redundancy or retirement, or transfer to new projects; and
- shortage of new staff with appropriate skills or training opportunities.

Knowledge about general systems (automated or manual) used or developed within an organisation is often referred to as *organisational memory*. Some organisations create specific business units to manage organisational knowledge and minimise losses.

Sometimes, some change to a legacy system becomes inevitable. Figure 29.1 illustrates the different forms of change that may be contemplated for a legacy system, based on its value to the organisation and on the ease with which the system can be changed.

Organisational  
memory (860)

When to re-engineer  
(861)

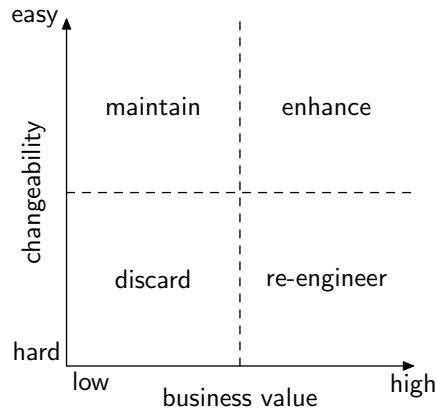


Figure 29.1: Deciding how to change a legacy system

In general systems that have little business value should attract few resources for maintenance or other changes. In particular, such systems that have also become very hard to change should probably just be discarded. The continued investment in the software system (however small) is probably unjustified. This may be because the legacy system was successfully replaced by a new system and is now largely redundant. Equally, if a system is still relatively easy to change it may be appropriate to continue *maintaining* the system, even though the business value is low. This is because the lower investment required to keep the system functional is justified for the small benefit obtained.

Conversely, substantial investment may be required for systems that have high business value. There are two options in this situation. If a system is relatively easy to change, then enhancements can be made directly to the existing system. However, if the system has become hard to change, i.e. it has the characteristics of a legacy system, then *system re-engineering* or complete replacement may be necessary.

Figure 29.2 illustrates the compromise between refactoring, reengineering and replacement in terms of benefit and risk. Normal maintenance activities permit continued use of a system, but may offer limited opportunity for the introduction of new features that enhance system value. Conversely, replacement provides considerable freedom to re-implement the software on a well-supported platform, with the opportunity to introduce desired new features.

However, as has already been discussed, complete replacement of a legacy system entails considerable risk:

- Replacement projects often commit an organisation to a new system before the system has been deployed. Consequently, there may not be an option to roll back to the existing legacy system if the new development project fails.
- There may not be the option of introducing the new system in parallel to

Risk vs value in software maintenance (862)

Causes of risk for

software replacement (863)

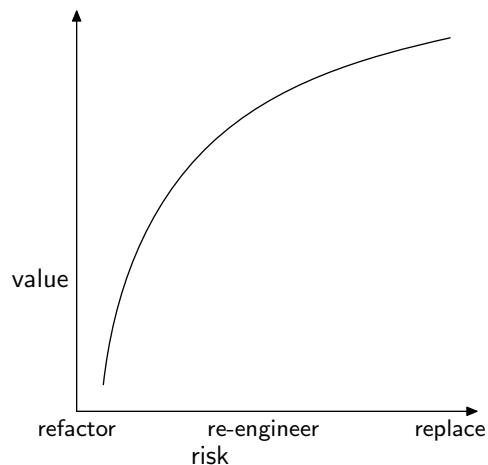


Figure 29.2: risk vs value spectrum in software maintenance

the existing system. This may be due to resource constraints on personnel. Training on the new system may also be incomplete as a consequence.

- There may be risks associated with the use of the new software platform, particularly if it has not stabilised as a software project. Frequent changes in the software platform may cause disruption to the replacement project.

Consequently, re-engineering is a compromise between these two extreme possibilities, allowing the existing architecture and functionality of a software system to be gradually *ported* to a new platform, reducing the risk of a one-off effort to deploy a complete replacement.

In general, it is preferable to avoid large-scale re-design of a software system during re-engineering. Smaller scale refactorings will largely be appropriate to assist in source code comprehension and platform migration. However, it is sometimes necessary to undertake some re-design of a software system to ease the migration process, particularly if a software system is poorly or inappropriately structured.

Techniques in software re-engineering include:

- reverse engineering;
- software comprehension;
- source code translation;
- software refactoring;
- platform migration; and
- data re-engineering or cleaning.

These tasks may all be performed in parallel.

## 29.2 Reverse Engineering

In software engineering, reverse engineering usually refers to the translation of a machine code artefact into a higher level language, using:

Reverse Engineering (865)

- a combination of automated tools; and
- and manual inspection.

This is then followed by a software comprehension and refactoring process to recover what the software does; and how it does it. Refactoring is typically limited to replacing automatically generated labels with more meaningful identifiers that were lost during the compilation process. However, comprehension of decompiled code can be challenging because:

- the original source code may have been substantially optimised by the compiler during the compilation process; and
- the developer may have deliberately used obfuscation techniques.

The Java source code shown below is for a simple `HelloWorld` class, that can be reverse engineered using standard software tools.

Example `HelloWorld` application (866)

```
package uk.ac.glasgow.psd.decomp;

public class HelloWorld {
    public String toString(){
        return "Hello, world!";
    }

    public static void main (String [] args){
        System.out.println(new HelloWorld());
    }
}
```

The code can be *disassembled* using the standard `javap` command line application. The code below shows the result of dissassembly.

Disassembly with `javap` (867)

```
Compiled from "HelloWorld.java"
public class uk.ac.glasgow.psd.decomp.HelloWorld extends
    java.lang.Object{
    public uk.ac.glasgow.psd.decomp.HelloWorld();
    public java.lang.String toString();
    public static void main(java.lang.String []);
}
```

Information is included on the class package name, its hierarchy, and the specification of method signatures and attributes.

Java class files can also be de-compiled. The code shown below illustrates the decompilation of the `HelloWorld` class using the Jdec Java decompiler.

Decompilation with Jdec (868)

```

import java.io.PrintStream;
import java.lang.Object;
import java.lang.String;
import java.lang.System;

//End of Import

public class HelloWorld

{

//CLASS: uk.ac.glasgow.psd.decomp.HelloWorld:
public      HelloWorld( )
{
super();
return;

}

//CLASS: uk.ac.glasgow.psd.decomp.HelloWorld:
public      String toString( )
{
return "Hello, world!";
}

//CLASS: uk.ac.glasgow.psd.decomp.HelloWorld:
public static void main( String [] args)
{
HelloWorld JdecGenerated5 = new HelloWorld();
System.out.println(JdecGenerated5);
return;

}
}

```

### 29.3 Source Code Translation

Source code  
translation (869)

Once a source code base has been obtained it can be translated from the legacy language to the new target language. This task may vary in difficulty, depending on:

- the similarity in concepts, principles and features between the two languages;

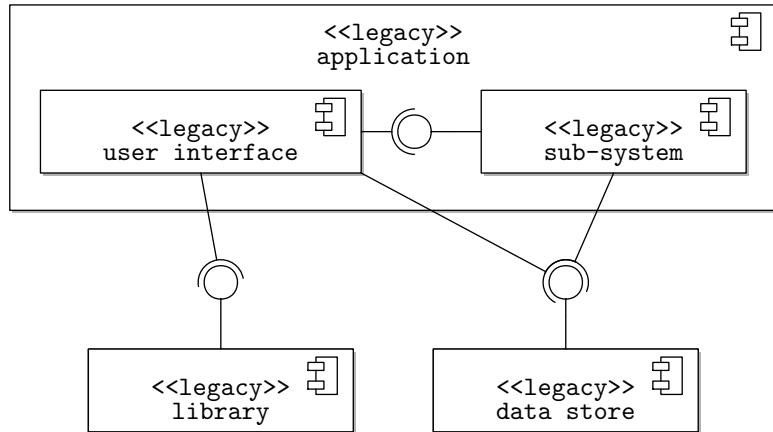


Figure 29.3: A poorly structured legacy system architecture.

- the availability of comparable libraries; and
- the availability of automated translation tools.

For example, translation between an object oriented language, such as C to another object oriented language such as Java, will likely be relatively straightforward, as both languages embody similar concepts. There may be some difficulty with dealing with issues such as multiple inheritance in C. However, this is much simpler than translating from a declarative or functional language into an imperative programming language.

## 29.4 Architectural Re-structuring

Consider the system architecture illustrated in the component diagram in figure 29.3. The software application is tightly coupled and dependent on two external legacy components, a library and a data store.

Ideally, it would be preferable to migrate the system to a new platform. However, the extent of coupling between the components means that the system will need to be transferred to the new platform as a whole: a high risk proposition. Instead, it may be possible to use two types of software components to migrate changes in the software system gradually. To do this, the architecture needs to be re-structured before individual components can be migrated.

Architectural  
restructuring (870)

Figure 29.4 illustrates the same legacy application, but with a re-structured to ease platform migration. Specifically, a new sub-system has been introduced between the user interface component and its dependencies. This sub-system will contain the business logic of the application that had become merged into the user interface. As a consequence, the overall coupling in the system has been reduced.

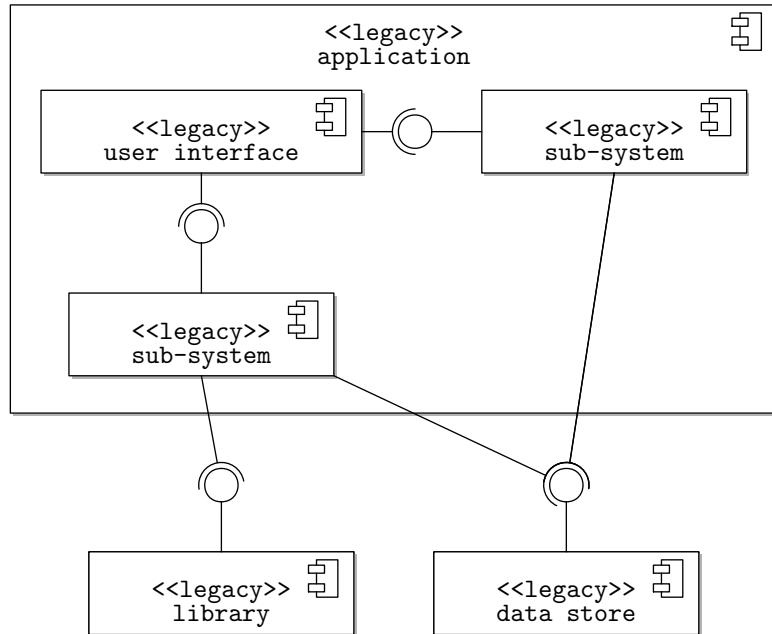


Figure 29.4: A revised structure for the legacy system architecture.

## 29.5 Platform Migration

Cleanly de-coupled legacy architectures can be gradually migrated from the legacy platform to the target platform. Individual components should be identified in the order in which they should be migrated as part of a *migration plan*. The end goal of this plan may only be a partial migration, leaving some components behind on the legacy platform. This may be necessary if the resources for the migration project do not permit a complete migration. Future projects may be able to complete this work.

Platform migration  
(872)

Two types of artifact are useful when undertaking a gradual migration of a legacy system to a new software platform.

- *Software wrappers* (sometimes called *adapters*) are superficial software layers that are added to legacy components. A wrapper maps directly from API calls in the target language to the API of a legacy component. A wrapper is essentially *passive* and only responds to requests from a client in the target environment.
- *Software gateways* are components which mediate interactions between multiple legacy components. Unlike a wrapper, a gateway coordinates interactions between multiple components, both receiving and initiating activity.

Figures 29.5 and Figure 29.6 illustrate the related but distinct uses of soft-

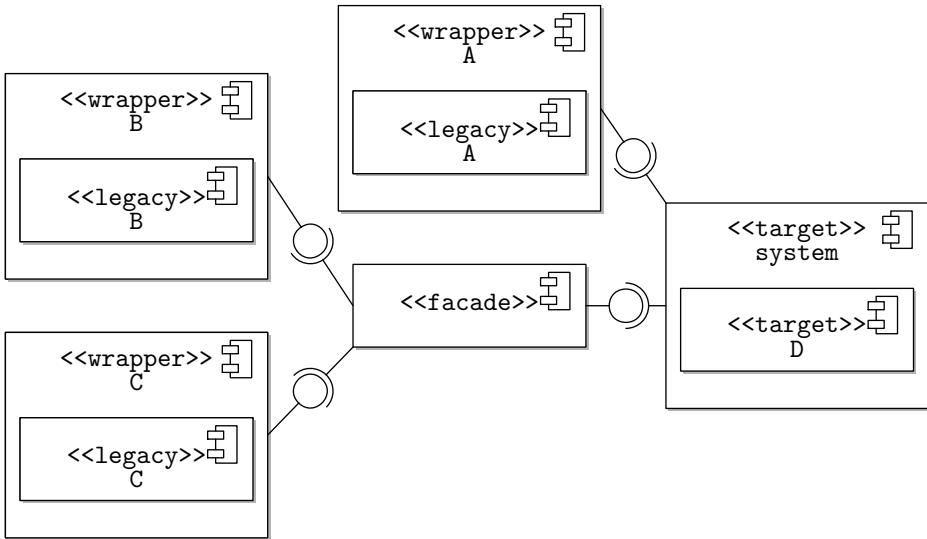


Figure 29.5: software wrappers during migration

ware wrappers and gateways during software migration.

Software wrappers  
(873)

Software wrappers are useful when passive legacy components need to be integrated into a re-engineered or replacement system. The diagram shows a target system implemented on a new target platform. The system is dependent on a number of legacy components, one of which has been re-engineered and incorporated into the new system on the target platform.

Several other legacy components have not yet been implemented on the target platform. Instead, the target system interacts with these legacy components via a wrapper. A software provides an API to the target system in the same language as the target platform. Any calls to the API from the target system are delegated to the legacy component using one or more operations of the legacy component's API. The implementation of the API in the legacy component is separated from its use in the new target system by the use of the wrapper. The wrapper can eventually be replaced, once the legacy component has been reengineered for the target platform.

In some cases, it may be useful to provide an intermediate *façade* over several wrappers which provide access to individual legacy components. Façades are used when it is useful to simplify a series of complex interactions with several components for a client. The use of a façade is shown in Figure 29.5.

Figure 29.6 illustrates the use of a gateway during the migration of a legacy system. The system is in an intermediate state, having successfully migrated one component (D) to the new target platform. The system is midway through the migration of component C, so both the legacy and target versions of this component are shown.

Gateways are used to mediate interactions between legacy and target compo-

Software gateways  
(874)

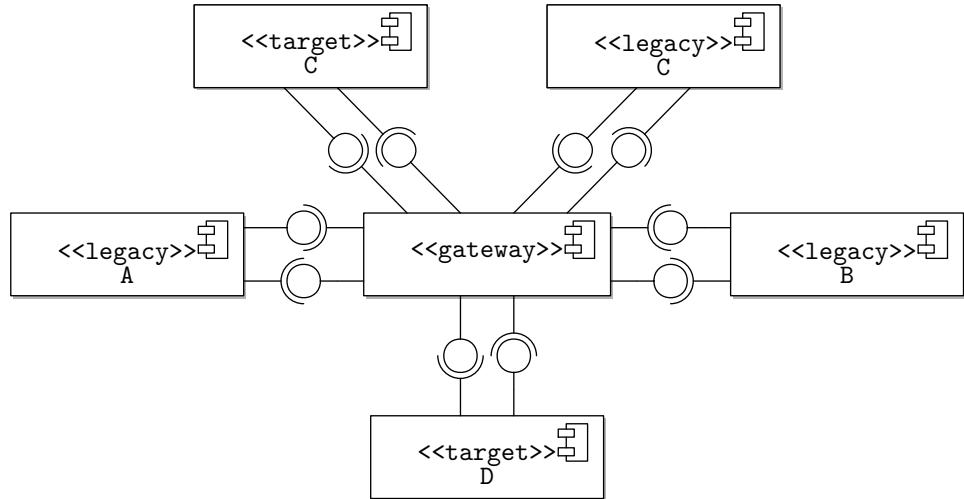


Figure 29.6: software gateways during migration

nents. These interactions occur transparently, such that the legacy component does not need to be adapted to the functionality of the gateway. The gateway depends on APIs exported by legacy components, in order to utilise their functionality. This connection may be done directly, or through external wrappers for the legacy components. In the former case, the gateway act as a wrapper to the legacy component.

In addition, the gateway provides interfaces for components to *initiate* interactions with other components. The gateway maintains a routing table between the system components, so that requests can be directed to the appropriate target to legacy component. As more components are implemented on the new platform, the routing table is updated to reflect the change.

Legacy versions of reengineered components do not need to be immediately removed from the system. By keeping the legacy components, it is possible to back out of the migration step, if the step proves unsuccessful.

Eventually, all components should be transferred to the new platform. If a decision is made to keep some legacy components (because there are insufficient resources to reengineer them), then these should be provided with a wrapper.

Figure 29.7 illustrates the use of two gateway components in a layered architecture. This arrangement is common when migrating legacy information systems, which typically have a classic user interface, business logic, data layer layered architecture.

The figure illustrates an information system which originally consisted of a command line user interface, a business logic component and data storage in one or more files on a file system. There is a desire to re-engineer the system using a modern software platform. Data in the new system will be managed in a relational database management system, probably managing multiple tables for

Multiple gateways in a layered architecture  
(875)

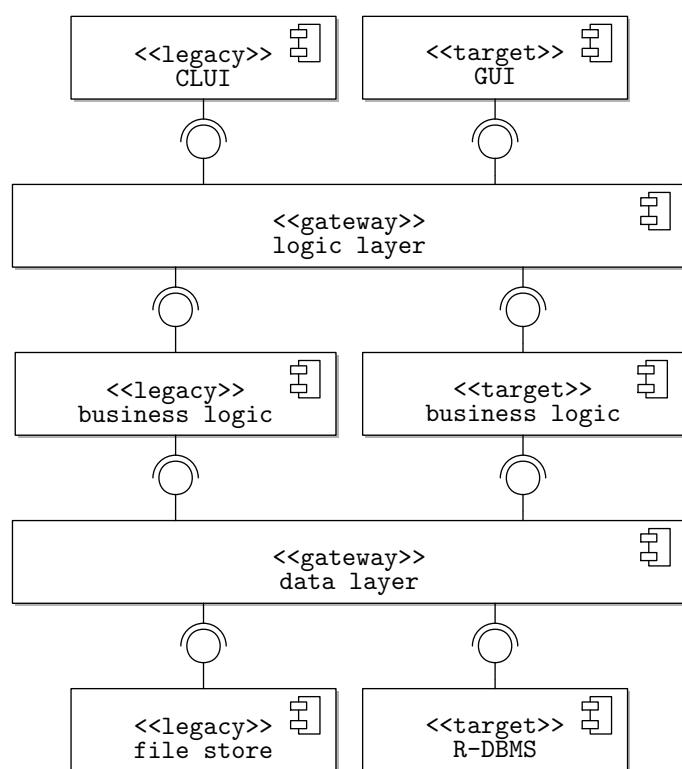


Figure 29.7: multiple gateways in a layered architecture

the multiple files stored in the file store. A new graphical user interface will also eventually be provided and the business logic also needs to be re-implemented on the new software platform.

To manage this process, gateways are installed between the user-interface and business logic, and between the business logic and data layers. In addition, new components implemented on the target platform are added to the gateways, mirroring the functionality of the legacy components. Initially these components will not contain any functionality and all requests from higher layers will be routed to legacy components in the lower layers.

As new functionality is added to the target platform components, the gateway routing tables are updated so that requests begin to be made to the parts of the target platform components that have been implemented. At the data layer, the same information may be managed in parallel in the two data storage components temporarily, so that a migration is not successful, it can be rolled back.

Brodie and Stonebraker [1993] provides a more comprehensive discussion of the issues involved in gradually migrating information systems. They refer to the gradual migration approach as 'Chicken Little' (i.e. very cautious!).

## Summary

Re-engineering can be a high risk activity, particularly if a large number of changes are attempted simultaneously. Small, incremental steps help reduce that risk.

## 29.6 Workshop: Using Logging Frameworks (log4j)

Logging, the act of recording or reporting program behaviour separately from the normal user interface has a number of uses:

Why use logging?  
(876)

- separate runtime behaviour reports from output for user;
- auditing and record keeping, e.g. security applications;
- performance monitoring;
- tracking intermediate results;
- reporting defects;
- because debugging is not available/inappropriate, if the application is distributed, for example; and
- as a form of documentation.

A logging policy is a description of the decisions made by a developer or organisation regarding the types of actions that should be reported in logs, the amount of detail to be included in a log message and the format for a log message. Implementing a logging policy in an application is problematic because:

Problems with logging  
(877)

- the amount of information required for a particular task can vary; and
- it is hard to ensure consistency of logging style between different developers, including:
  - message content and format,
  - storage location; and
- logging imposes extra runtime load on the application
  - compute time, and
  - storage access time.

The Java code below illustrates an example of this problem:

Logging without a framework (878)

```
public void updateWithPlainLogging(
    String filePath, String key, Serializable value)
    throws PersistenceException{

    System.out.println("Opening resource ["+filePath+"].");
    Map<String,Serializable> values = getPersistenceMap(
        filePath);
```

```

System.out.println("inserting "+key+":"+value+" pair.");
        ;
values.put(key, value);

System.out.println("LoggingPersistent." +
    "updateWithPlainLogging() committing file.");
writePersistenceMap(filePath,values);
System.out.println("::Done::");
}

```

The logging statements have been hard code to print to the standard output stream, potentially becoming mixed with actual output information for the user. If the code is used as part of another application, the logging messages will probably have to be commented out when debugging has been completed. The log messages also have varying formats and take variable levels of detail.

Goal of a logging framework (879)

A *logging framework* centralise control of:

- appenders, directing where log messages should be stored;
- verbosity, allowing the amount of detail included in a log to be varied according to need without altering the application source code; and
- log message format, enforcing consistency on the detail included in each log message.

log4j (880)

The log4j<sup>2</sup> framework is part of the Apache framework collection, which also includes:

- the apache web server;
- Tomcat application container;
- Axis SOAP implementation;
- Apache commons;
- Xerces XML Parser; and
- the Ant deployment scripting tool.

more informationon log4j can be found on the project website:

<http://logging.apache.org/log4j/1.2/index.html>

Setting up loggers in log4j (881)

The Java code below is taken from the Programmatic class, showing how to setup and use the log4j framework from within an application.

```

private static Logger logger =
Logger.getLogger(Programmatic.class);

```

---

<sup>2</sup>many Java libraries, tools are called <thing>4j

The class contains a static `Logger` attribute, which is initialized when the class is loaded using the static `Logger.getLogger()` method. The Programmatic `.class` class object is passed as an argument to this method in order to specify that we want the logger for this class. Every class in an application can have its own logger.

The first step is to setup the logging framework with an appender.

```
Layout layout = new PatternLayout();  
  
Appender appender = new ConsoleAppender(layout);  
  
logger.addAppender(appender);  
logger.setLevel(Level.DEBUG);
```

Setting up loggers in log4j (882)

The first statement of the `Programmatic()` constructor creates a `Layout` for specifying how messages should be formatted. The default `PatternLayout` just reports the logging message itself.

The second line creates a `ConsoleAppender` appender for reporting log messages to the console, and the next two statements after this attach the `Appender` instance to the class' `Logger`, and sets the logging level to `DEBUG`. There is a hierarchy of logging levels available in log4j:

- OFF no messages
- FATAL just before `System.exit`
- ERROR requested action could not be completed
- WARN bad, but not terrible stuff
- INFO progress notification
- DEBUG verbose for debugging
- TRACE messages that trace method invocations and exits.

Setting the log level to debug means that all messages except TRACE level messages will be sent to the appender. The next statement in the code logs the debug message "`Oops`", which appears as the first line in the log, as shown below.

```
logger.debug("Oops!");
```

```
Oops!
```

Setting up loggers in log4j (884)

The next two statements alter the format of log messages by specifying a new `PatternLayout`.

## Setting up loggers in log4j (885)

```
Layout layout2 = new PatternLayout(  
    "%p %d{ISO8601} %t %c - %m%n");  
  
appender.setLayout(layout2);  
  
logger.debug("Oh no");  
  
DEBUG 2011-05-25 15:33:56,929 main uk.ac.glasgow.oose2.  
logging.Programmatic - Oh no
```

The layout is configured with a string which includes a number of characters preceded by the % escape character. These escaped characters allow extra information to be included in the log, including the log level (%p), date and time the message was logged (%d) and %t), the name of the logger (%c) and the message itself (%m), followed by a new line (%n).

The final statement in the block causes another debug log message to be added to the log, but this time in the new message format specified.

The next three statements illustrate the logging level hierarchy further. By setting the log level to FATAL, only the most important log messages will be added to the log. Consequently, the debug log message does not appear in the log, but the fatal log message does.

## Setting up loggers in log4j (886)

```
logger.setLevel(Level.FATAL);  
logger.debug("oh, that shouldn't have happened!");  
  
logger.fatal("oh, that didn't get printed out!");  
  
FATAL 2011-05-25 15:33:56,940 main uk.ac.glasgow.oose2.  
logging.Programmatic - oh, that didn't get printed out  
!
```

Perhaps the most common use for log4j is for logging exceptions. The block of Java code shown below illustrates how to report exceptions in logs as errors.

## Setting up loggers in log4j (887)

```
logger.setLevel(Level.ERROR);  
  
String fileName = "no-such-file.txt";  
try {  
    File f = new File(fileName);  
    InputStream fis = new FileInputStream(f);  
} catch (FileNotFoundException e) {  
    logger.error("Couldn't find file ["+fileName+"].", e);  
}
```

The error() method call is used inside a **catch** block to record the occurrence of the exception. Notice that the error() method has an extra argument - the exception (or in fact Throwable that caused the error. The result in the log file is shown below. Notice that the method stack trace is included in the log after the logging message.

```

ERROR 2011-05-25 15:33:56,941 main uk.ac.glasgow.oose2.
    logging.Programmatic - Couldn't find file [no-such-
    file.txt].
java.io.FileNotFoundException: no-such-file.txt (No such
    file or directory)
at java.io.FileInputStream.open(Native Method)
at java.io.FileInputStream.<init>(FileInputStream.java
    :137)
at uk.ac.glasgow.oose2.logging.Programmatic.<init>(
    Programmatic.java:62)
at uk.ac.glasgow.oose2.logging.Programmatic.main(
    Programmatic.java:27)

```

Notice also, that log4j cannot be used to enforce format consistency within messages, or the particular values that will be reported. This detail is still left to the developer to decide manually.

The log4j framework would not be much use if the logging configuration had to be set programmatically. Fortunately, log4j incorporates mechanisms for specifying configuration files that can be read into the application on start up.

The code below shows how to configure log4j using a properties file:

```

PropertyConfigurator.configure("config/l4j.properties")
;

logger.debug("How about this?");

Logger.getRootLogger().trace("So why didn't this work?"
);

```

The contents of the log file are shown below:

```

## Root logger
log4j.rootLogger=DEBUG, R

log4j.appender.R=org.apache.log4j.ConsoleAppender
log4j.appender.R.layout=org.apache.log4j.PatternLayout
log4j.appender.R.layout.ConversionPattern=%p %d{ISO8601}
    %t %c : %m%n

log4j.logger.uk.ac.glasgow.oose2=logging=FATAL
log4j.logger.uk.ac.glasgow.oose2.logging.Programmatic=
    INFO

```

The properties file uses a key=value format to configure the log4j format. The first line sets the root logger to the debug level and attaches the appender named R. The next three lines configure the R appender as a ConsoleAppender , with the same layout as above. There are many other types of appenders available in the framework including (for example) logging to:

- a log file;

[Setting up loggers in log4j \(889\)](#)

[Setting up loggers in log4j \(890\)](#)

[Appenders \(891\)](#)

## Logger Name Hierarchy (892)

- any other PrintWriter;
- a remote logging server; and
- an SMTP server.

The final two statements configure the logger for logging package and Programmatic class respectively. This shows that the log levels of individual loggers can be configured independently. Loggers are organised into a hierarchy which mirrors the package hierarchy for an application:

- uk.ac.glasgow.oose2
- uk.ac.glasgow.oose2.logging
- uk.ac.glasgow.oose2.logging.Programmatic

with the root logger at the top. If a log level is not explicitly set for a particular logger, then the log4j framework searches up the logger hierarchy until a log level is found. Consequently the debug() call on the Programmatic class logger is not recorded, because the logger has been restricted to the INFO level.

# Bibliography

Kent Beck and Cynthia Andres. *Extreme Programming Explained*. XP Series. Addison Wesley/Pearson Education, second edition, February 2005.

Kent Beck and Ward Cunningham. A laboratory for teaching object oriented thinking. *ACM SIGPLAN Notices*, 24(10):1–6, October 1989.

Herbert D. Benington. Production of large computer programs. *Annals of the History of Computing*, 5(4):350–361, October 1983.

Simon Bennett, Steve McRobb, and Ray Farmer. *Object Oriented Systems Analysis and Design Using UML*. McGraw Hill, Shoppenhangers Road, Maidenhead, Berkshire, SL6 2QL, third edition, 2006.

Barry W. Boehm. A spiral model of software development and enhancement. *ACM Sigsoft Software Engineering Notes*, 11(4):14–24, August 1988.

Barry W. Boehm. Software risk management: Principles and practices. *IEEE Software*, 8(1):32–41, January 1991.

Grady Booch. Object oriented design. *ACM SIGAda Ada Letters*, 1(3):64–76, March/April 1982.

Pierre Bourque and Robert Dupuis, editors. *Guide to the Software Engineering Body of Knowledge, 2004 Edition*. IEEE Computer Society, Los Alamitos, California, March 2005.

Michael L. Brodie and Michael Stonebraker. DARWIN: on the incremental migration of legacy information systems. Technical Report TR-0222-10-92-165, University of California, Berkeley, March 1993.

Frederick P. Brooks, Jr. *The Mythical Man-Month*. Addison Wesley, ninth edition, 1995. ISBN 0-201-83595-9.

Jordi Cabot and Ernest Teniente. Constraint support in MDA tools: a survey. In Jan P. Nytn, Andreas Prinz, and Merete S. Tveit, editors, *Automatic Generation of Modelling Tools. European Conference on Model-Driven Architecture 2006*, volume 4066 of *Lecture Notes in Computer Science*, pages 256–267. Springer Verlag, 2006.

- Dominique Cansel, J Paul Gibson, and Dominique Méry. Refinement: a constructive approach to formal software design for a secure e-voting interface. In *1st International Workshop on Formal Methods for Interactive Systems*, Macau SAR China, October 2006.
- Narciso Cerpa and June M. Verner. Why did your project fail? *Communications of the ACM*, 52(12):130–134, December 2009.
- Brad J. Cox. Planning the software industrial revolution. *IEEE Software*, 7(6): 25–33, November/December 1990.
- Martha E. Crosby, Jean Scholtz, and Susan Wiedenbeck. The roles beacons play in comprehension for novice and expert programmers. In J. Kuljis, L. Baldwin, and R. Scoble, editors, *14th Annual Workshop of the Psychology of Programming Interest Group*, pages 58–73, Brunel University, UK, June 2002.
- Darren Dalcher. Disaster in london the LAS case study. In *Proceedings of the IEEE Conference and Workshop on Engineering of Computer-Based Systems, ECBS '99*, pages 41–52, Nashville, TN, USA, March 1999. IEEE Computer Society Press.
- Lionel E. Deimel and J. Fernando Naveda. Reading computer programs: Instructor's guide and exercises. Educational Materials CMU/SEI-90-EM-3, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213, August 1990.
- Edsger W. Dijkstra. Go-to statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968.
- Khaled El Emam and A. Günes Küçük. A replicated survey of IT software project failures. *IEEE Software*, 25(5):84–90, September/October 2008.
- Anthony Finkelstein and John Dowell. A comedy of errors: the london ambulance service case study. In *Proceedings of the Eighth International Workshop on Software Specification and Design*, pages 2–4, Washington, DC, USA, 1996. IEEE Computer Society Press.
- Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Object Technology Series. Addison Wesley, Pearson Education Inc, One Lake Street, Upper Saddle River, NJ 07458, USA, 2000.
- Bernd Freimut, Susanne Hartkopf, Peter Kaiser, Jyrki Kontio, and Werner Kowitzsch. An industrial case study of implementing software risk management. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering, ESEC/FSE-9*, pages 277–287, Vienna, Austria, 2001. ACM Press.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns : elements of reusable object-oriented software*. Professional Computing Series. Addison Wesley, first edition, October 1994.

David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch or why itâŽs hard to build systems out of existing parts. In , editor, – *International Conference on Software Engineering*, pages 179–185, Seattle, WA, USA., – 1995. ACM Press.

Robert L. Glass. *Software Runaways: Lessons Learned from Massive Software Project Failures*. Prentice Hall, first edition, September 1997a.

Robert L. Glass. Software runaways some surprising findings. *The DATA BASE for Advances in Information Systems*, 28(3):16–19, Summer 1997b.

Ron Gould. Independent review of the Scottish Parliamentary and local government elections 3 may 2007. The Electoral Commission, Trevelyan House, Great Peter Street, London, SW1P, October 2007.

David Harel. On visual formalisms. *Communications of the ACM*, 31(3):514–530, May 1988.

Richard Hopkins and Kevin Jenkins. *Eating the IT Elephant: Moving from Greenfield Development to Brownfield*. IBM Press, 2008.

William E. Howden. Validation of scientific programs. *Computing Surveys*, 14 (2):193–227, June 1982.

Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Object Technology Series. Addison Wesley, One Jacob Way, Reading, Massachusetts 01867, 1999.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammadika Elkaduve, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In , editor, – *To appear 22nd ACM Symposium on Principles of Operating Systems (SOSP)*, pages –, Big Sky, MT, USA, October 2009.

Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, and Stefan Savage. Experimental security analysis of a modern automobile. In *31st IEEE Symposium on Security and Privacy*, pages 447–462, Berkeley/Oakland California, USA, May 2010. IEEE Computer Society.

John Krogstie, Arthur Jahr, and Dag I.K. Sjøberg. A longitudinal study of development and maintenance in Norway: Report from the 2003 investigation. *Information and Software Technology*, 48(11):993–1005, 2006.

Philippe Kruchten. *Rational Objectory Process*. Rational Software Corporation, 4.1 edition, September 1997.

Meir M. Lehman. On understanding laws, evolution and conservation in the large-program life cycle. *Journal of Systems and Software*, 1:213–221, 1980.

Meir M. Lehman. Laws of software evolution revisited. In Carlo Montangero, editor, *Proceedings of the 5th European Workshop on Software Process Technology*, volume 1149 of *Lecture Notes In Computer Science*, pages 108–124, Nancy, France, October 1996. Springer-Verlag.

Timothy C. Lethbridge and Robert Laganière. *Object-Oriented Software Engineering*. McGraw Hill, Shoppenhangers Road, Maidenhead, Berkshire, SL62QL, second edition, 2005. Practical Software Development using UML and Java.

Bennet P. Lientz and E. Burton Swanson. *Software Maintenance Management. A study of the Maintenance of the Computer Application Software in 487 Data Processing Organisations*. Addison-Wesley, 1980.

Russell Lock, Tim Storer, Natalie Harvey, Conrad Hughes, and Ian Sommerville. Observations of the Scottish elections 2007. *Transforming Government: People, Process and Policy*, 2(2):104–118, 2008.

James Martin. *Rapid Application Development*. Macmillan, 886 Third Avenue, New York, New York 10022, 1991.

Greg Miller. Scientific publishing: A scientist's nightmare: Software problem leads to five retractions. *Science*, 314(5807):1856–1857, 2006.

Chris Morris. Some lessons learned reviewing scientific code. In , editor, *SECSE'08*, volume – of –, pages –, Leipzig Germany, May 2008. –, –.

B. T. Mynatt. *Software Engineering with Student Project Guidance*. Prentice-Hall, 1990.

Peter Naur and Brian Randell, editors. *Report on a conference sponsored by the NATO Science Committee*, Garmisch, Germany, 1968 1968.

Vu Nguyen, Barry Boehma, and Phongphan Danphitsanuphan. A controlled experiment in assessing and estimating software maintenance tasks. *nformation and Software Technology*, 52(6):682–691, June 2011.

John T. Nosek and Prashant Palvia. Software maintenance management: Changes in the last decade. *Sorware Maintenance: Research and Practice*, 2 (3):157–174, 1990.

Emil Obreshkov. Software release build process and components in ATLAS offline. In *Conference on Computing in High Energy and Nuclear Physics 2010*, Taipei, Taiwan, October 2010.

Roger S. Pressman. *Software Engineering A Practitioner's Approach*. McGraw Hill, fourth edition, 1997.

Rational. The rational unified process. best practices for software development teams. white paper TP026B, The Rational Corporation, 2003.

Eric S. Raymond. The cathedral and the bazaar: musings on linux and open source by an accidental revolutionary. *Information Research*, 6(4), 2001. URL <http://www.tuxedo.org/~esr/writings/cathedral-bazaar/>.

Bruce Robinson. Limited horizons, limited influence: Information technology the london ambulance service. In *Proceedings of the International Symposium on Technology and Society Technical Expertise and Public Decisions*, pages 506–514. IEEE Computer Society Press, June 1996.

Scott Rosenberg. *Dreaming in Code*. Crown, 2007.

James Rumbaugh, Michael R. Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object Oriented Modelling and Design*. Prentice-Hall International, Englewood Cliffs, New Jersey, first edition, October 1991.

Rebecca Sanders and Diane Kelly. Dealing with risk in scientific software development. *IEEE Computer Software*, 25, 2008.

Eliot Soloway and Kate Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, 10(5):595–609, September 1984.

Ian Sommerville. Software construction by configuration: Challenges for software engineering research. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, Budapest, Hungary, September 2005. IEEE Computer Society.

Ian Sommerville. *Software Engineering*. International computer science. Addison-Wesley, eighth edition, 2007.

Ian Sommerville. *Software Engineering*. International computer science. Addison-Wesley, ninth edition, 2010.

South West Thames Regional Health Authority. Report of the inquiry into the london ambulance service. South West Thames Regional Health Authority, February 1993.

Tim Storer and Russell Lock. Accuracy: The fundamental requirement for voting systems. In *Proceedings of the The Third International Conference on*

*Availability, Reliability and Security, ARES 2008*, pages 374–379, Fukuoka, Japan, March 2009. IEEE Computer Society.

U.S. CFTC/SEC. Findings regarding the market events of May 6, 2010. report of the staffs of the CFTC and SEC to the joint advisory committee on emerging regulatory issues. U.S. Commodity Futures Trading Commission. Three Lafayette Centre, 1155 21st Street, NW Washington, D.C. 20581. U.S. Securities & Exchange Commission 100 F Street, NE Washington, D.C. 20549, September 2010.

Stephen Viller and Ian Sommerville. Ethnographically informed analysis for software engineers. *International Journal of Human-Computer Studies*, 53(1):169–196, 2000.

Susan Wiedenbeck and Nancy J. Evans. Beacons in program comprehension. *SIGCHI Bulletin*, 18(2):56–57, 1986.

Jim Woodcock and Ana Cavalcanti. A concurrent language for refinement. In Andrew Butterfield, Glenn Strong, and Claus Pahl, editors, *5th Irish Workshop on Formal Methods, IWFM 2001*, Workshops in Computing, Ireland, Dublin, July 2001. British Computer Society.

Stephen W.L. Yip and Tom Lam. A software maintenance survey. In *Proceedings of the First Asia-Pacific Software Engineering Conference*, pages 70–79, Tokyo, Japan, December 1994. IEEE Computer Society.

## Appendix A

# Use Case Description Template

A collection of LaTeX macros have been defined for a use case description template in the latex style package `usecasedescription`. The source file for the macros is available at.

<http://fims.moodle.ac.uk/course/view.php?id=128...>

The basic template is an environment that can be included in a `LATEX` document as shown below:

```
%...
\usepackage{usecadedescription}
%...
\begin{document}
%...
\begin{UseCaseTemplate}
\UseCaseLabel{}
\UseCaseDescription{}
\UseCaseRationale{}
\UseCasePriority{}
\UseCaseStatus{}
\UseCaseActors{}
\UseCaseExtensions{}
\UseCaseIncludes{}
\UseCaseConditions{}
\UseCaseNonFunctionalRequirements{}
\UseCaseScenarios{}
\UseCaseRisks{}
\UseCaseUserInterface{}
\end{UseCaseTemplate}
```

The template appearance and usage is illustrated on the next page:

<b>Use case</b>	the use case label as appears on the use case diagram
<b>Description</b>	a brief textual description of the activity that occurs during the use case, accompanied by an activity diagram or pseudo code as appropriate
<b>Rationale</b>	a justification of the use case based on evidence gathered during requirements elicitation
<b>Priority</b>	MoSCoW categorisation of the use case
<b>Status</b>	the status of the use case (e.g. implemented, not elaborated) and any changes that have been made to it
<b>Actors</b>	<p>a list of the actors associated with this use case as shown on the use case diagram</p> <ul style="list-style-type: none"> <li>• actor 1</li> <li>• actor 2</li> <li>• ...</li> </ul>
<b>Extensions</b>	<p>a list of the use cases that extend this use case as shown on the use case diagram</p> <ul style="list-style-type: none"> <li>• use case 1</li> <li>• use case 2</li> <li>• ...</li> </ul>
<b>Includes</b>	<p>a list of use cases included by this use case, as shown on the use case diagram</p> <ul style="list-style-type: none"> <li>• use case 3</li> <li>• use case 4</li> <li>• ...</li> </ul>
<b>Conditions</b>	<p>a list of pre and post conditions</p> <p><b>pre</b> condition 1</p> <p><b>pre</b> condition 2</p> <p><b>post</b> condition 1</p> <p>...</p>

<b>Non-Functional Requirements</b>	a list of categorised, use case specific non-functional requirements  <b>security</b>  <b>user interface</b>  ...
<b>Scenarios</b>	a list of all the scenarios produced for this use case. Include key scenarios in the requirements document appendix  <b>primary:</b>  <b>alternative 1:</b>  <b>alternative 2:</b>  ...
<b>Risks</b>	a list of use case specific risks, referencing more detailed risk descriptions in the risk management plan.  <ul style="list-style-type: none"> <li>● risk 1.1</li> <li>● risk 2.5</li> <li>● risk 3.1.1</li> <li>● ...</li> </ul>
<b>User Interface</b>	user interface requirements relevant to this use case