Technology-Neutral Analysis and Design using UML and URDAD
## COLLABORATORS

<table>
<thead>
<tr>
<th>ACTION</th>
<th>NAME</th>
<th>DATE</th>
<th>SIGNATURE</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRITTEN BY</td>
<td>Fritz Solms</td>
<td>June 1, 2012</td>
<td></td>
</tr>
</tbody>
</table>

## REVISION HISTORY

<table>
<thead>
<tr>
<th>NUMBER</th>
<th>DATE</th>
<th>DESCRIPTION</th>
<th>NAME</th>
</tr>
</thead>
</table>

# Contents

1 Introduction to business analysis ........................................... 1

1.1 Introduction ........................................................................... 1

1.2 Definition of business analysis .............................................. 1

1.3 Sub-disciplines of business analysis ....................................... 1

1.4 Responsibilities of business analysis ...................................... 2

1.5 What is software design? ...................................................... 3

1.6 Architecture versus design ................................................... 3

1.6.1 Bibliography ..................................................................... 3

1.7 Model driven approach to software design .............................. 4

1.7.1 Automatic mappings ........................................................ 4

1.7.2 Independent business process and architecture optimization .. 5

1.8 Responsibilities of software design ......................................... 5

1.9 Challenges faced by software design ...................................... 5

1.9.1 Design challenges ............................................................ 5

1.9.1.1 Generating a "good design" ........................................ 5

1.9.1.2 Requirements tracing ............................................... 6

1.9.1.3 Validating a design .................................................. 6

1.9.1.4 Communicating a design .......................................... 6

1.9.2 Environmental challenges ............................................... 6

1.9.2.1 Requirements challenges ........................................ 6

1.9.2.2 Communication challenges ...................................... 6

1.10 Typical model-driven software development process ............. 7

1.11 Object-oriented concepts ..................................................... 8

1.11.1 Introduction ................................................................. 8

1.11.2 What is a use case? ....................................................... 8

1.11.3 What is an object? ....................................................... 8

1.11.4 What is a class? ........................................................... 8

1.11.4.1 Assigning process to services .................................. 9

1.11.4.2 Classes and substitutability ...................................... 9

1.11.5 Interfaces ..................................................................... 9

1.11.6 Components ............................................................... 9

1.11.7 Contracts in object oriented modeling ............................... 9
# The Unified Modeling Language (UML)

## Introduction

- What is UML
- History of UML
- UML does not provide a design methodology
- Applicability of UML
- Overview of the UML diagrams
  - Introduction
  - Use case diagrams
  - Sequence diagrams
  - Activity diagrams
  - Interaction overview diagrams
  - Class diagrams
  - Object diagrams
  - State charts
  - Timing diagrams
  - Communication diagrams
  - Component diagrams
  - Composite structure diagrams
  - Package diagrams
  - Deployment diagrams
- The UML model
- UML and MDA

## Use-case diagrams

- Introduction
- Simple use case diagrams
  - Introduction
  - Use cases
  - The subject
  - Communication links
  - User roles
- Stereotypes in use case diagrams
- Actors
  - Introduction
  - Actor types
- Example actors in system design
- Functional requirements
### 3.6 Technology-Neutral Analysis and Design using UML and URDAD

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.6.1</td>
<td>Introduction</td>
<td>22</td>
</tr>
<tr>
<td>3.6.2</td>
<td>Mandatory functional requirements via the include relationship</td>
<td>22</td>
</tr>
<tr>
<td>3.6.3</td>
<td>Conditional functional requirements via the extend relationship</td>
<td>22</td>
</tr>
<tr>
<td>3.6.3.1</td>
<td>Specifying the conditional</td>
<td>23</td>
</tr>
<tr>
<td>3.6.3.2</td>
<td>Specifying extension points</td>
<td>23</td>
</tr>
<tr>
<td>3.6.4</td>
<td>Identifying functional requirements</td>
<td>24</td>
</tr>
<tr>
<td>3.7</td>
<td>Use-case abstraction</td>
<td>24</td>
</tr>
<tr>
<td>3.7.1</td>
<td>Introduction</td>
<td>24</td>
</tr>
<tr>
<td>3.7.2</td>
<td>Scoping</td>
<td>24</td>
</tr>
<tr>
<td>3.7.2.1</td>
<td>Concrete use cases</td>
<td>25</td>
</tr>
<tr>
<td>3.7.3</td>
<td>Defining common requirements and common actors across use cases</td>
<td>25</td>
</tr>
<tr>
<td>3.8</td>
<td>Actor abstraction</td>
<td>26</td>
</tr>
<tr>
<td>3.9</td>
<td>Summary of the UML notation for use case diagrams</td>
<td>28</td>
</tr>
<tr>
<td>3.10</td>
<td>Exercises</td>
<td>28</td>
</tr>
</tbody>
</table>

#### 4 Class diagrams

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Introduction</td>
<td>29</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Objects and Classes</td>
<td>29</td>
</tr>
<tr>
<td>4.1.1.1</td>
<td>Objects</td>
<td>29</td>
</tr>
<tr>
<td>4.1.1.1.1</td>
<td>Common examples of objects</td>
<td>30</td>
</tr>
<tr>
<td>4.1.1.2</td>
<td>Identifying objects</td>
<td>30</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Classes</td>
<td>30</td>
</tr>
<tr>
<td>4.1.2.1</td>
<td>Identifying classes</td>
<td>31</td>
</tr>
<tr>
<td>4.2</td>
<td>Basic object and class diagrams</td>
<td>31</td>
</tr>
<tr>
<td>4.3</td>
<td>Mapping basic class and object diagrams onto Java</td>
<td>32</td>
</tr>
<tr>
<td>4.4</td>
<td>Mapping basic class and object diagrams onto XML</td>
<td>32</td>
</tr>
<tr>
<td>4.5</td>
<td>Attributes</td>
<td>33</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Collection attributes and multiplicity constraints</td>
<td>33</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Derived attributes</td>
<td>34</td>
</tr>
<tr>
<td>4.6</td>
<td>Mapping attributes onto Java</td>
<td>35</td>
</tr>
<tr>
<td>4.6.1</td>
<td>Mapping collection attributes onto Java</td>
<td>35</td>
</tr>
<tr>
<td>4.6.2</td>
<td>Mapping derived attributes onto Java</td>
<td>35</td>
</tr>
<tr>
<td>4.7</td>
<td>Mapping attributes onto XML</td>
<td>36</td>
</tr>
<tr>
<td>4.7.1</td>
<td>Mapping collection attributes onto XML</td>
<td>36</td>
</tr>
<tr>
<td>4.7.2</td>
<td>Mapping derived attributes onto XML</td>
<td>37</td>
</tr>
<tr>
<td>4.8</td>
<td>Services</td>
<td>37</td>
</tr>
<tr>
<td>4.8.1</td>
<td>Service inputs</td>
<td>37</td>
</tr>
<tr>
<td>4.8.1.1</td>
<td>Assigning a business process to a service</td>
<td>38</td>
</tr>
<tr>
<td>4.8.1.2</td>
<td>Assigning role names to the input objects</td>
<td>38</td>
</tr>
</tbody>
</table>
4.8.1.3 What if the service provider does not require any information from the client? .................................. 38
4.8.2 Return types .................................................................................................................................................. 39
  4.8.2.1 What if multiple deliverables? .................................................................................................................... 39
  4.8.2.2 What if nothing is returned? ........................................................................................................................ 39
4.8.3 Input, output and input/output parameters ..................................................................................................... 40
4.8.4 Multiple services with same service name (overloading) .................................................................................. 40
4.9 Mapping UML service declarations onto Java ...................................................................................................... 41
  4.9.1 Output and input/output parameters ................................................................................................................ 41
4.10 Access levels (visibility) ........................................................................................................................................ 42
4.11 Mapping UML access levels onto Java .................................................................................................................. 43
4.12 The camel naming convention .............................................................................................................................. 43
4.13 Interfaces .............................................................................................................................................................. 44
  4.13.1 Introduction .................................................................................................................................................... 44
  4.13.2 UML notation for interfaces ............................................................................................................................ 45
  4.13.3 Realizing/implementing interfaces ................................................................................................................ 45
  4.13.4 Decoupling from service providers ................................................................................................................ 45
    4.13.4.1 Required and provided interfaces ............................................................................................................. 46
  4.13.5 Extending interfaces ........................................................................................................................................ 47
    4.13.5.1 Join interfaces ........................................................................................................................................... 48
  4.13.6 Using a UML interface to specify a services contract ....................................................................................... 48
    4.13.6.1 Example: a services contract for a caterer ................................................................................................. 49
    4.13.6.2 Example: a message sender ...................................................................................................................... 49
  4.13.7 Guidelines for defining interfaces ................................................................................................................... 50
4.14 Mapping UML interfaces onto Java ...................................................................................................................... 50
4.15 Interfaces versus classes versus objects ............................................................................................................... 52
4.16 Class specialization ............................................................................................................................................... 52
  4.16.1 Introduction .................................................................................................................................................... 52
  4.16.2 UML notation for specialization ................................................................................................................... 52
  4.16.3 Substitutability ................................................................................................................................................ 53
  4.16.4 Inheritance .................................................................................................................................................... 54
    4.16.4.1 Inheriting and overriding business processes .......................................................................................... 55
  4.16.5 Polymorphism ................................................................................................................................................. 55
    4.16.5.1 Polymorphism on message recipient ......................................................................................................... 55
    4.16.5.2 Polymorphism on message parameters .................................................................................................. 55
  4.16.6 Abstract classes ............................................................................................................................................... 56
    4.16.6.1 Working with abstract concepts ................................................................................................................ 57
    4.16.6.2 Abstract services ....................................................................................................................................... 58
  4.16.7 Multiple inheritance ....................................................................................................................................... 59
  4.16.8 Completeness constraints ............................................................................................................................... 59
4.25 Aggregation ................................................................. 82
  4.25.1 Introduction ......................................................... 82
  4.25.2 UML notation for aggregation ................................. 82
  4.25.3 Difference between aggregation and composition .......... 82
4.26 Mapping UML aggregation relationships onto Java .......... 83
4.27 Mapping UML aggregation relationships onto XML .......... 85
4.28 Dependencies .......................................................... 87
4.29 Dependency relationships in Java .................................. 88
4.30 Meta classes ............................................................ 88
4.31 Mapping UML meta-classes onto Java .............................. 89
4.32 Containment ............................................................... 90
  4.33.1 Containment relationship between packages .................. 91
  4.33.2 Static nested classes ............................................. 91
  4.33.3 Inner classes ....................................................... 92
    4.33.3.1 Classes local to a method and anonymous inner classes 94
  4.33.4 Nesting interfaces ................................................. 94
  4.33.5 Nesting a class in an interface ................................. 95
4.34 Mapping UML containment relationships onto XML .......... 96
4.35 Summary of UML relationships ..................................... 96
  4.35.1 Dependency .......................................................... 97
  4.35.2 Association .......................................................... 97
  4.35.3 Aggregation .......................................................... 97
  4.35.4 Composition ........................................................ 98
  4.35.5 Realisation .......................................................... 98
  4.35.6 Specialisation ....................................................... 98
  4.35.7 Containment ........................................................ 98
  4.35.8 Shopping for relationships ....................................... 98
4.36 Templates ............................................................... 99
  4.36.1 Template classes .................................................. 99
4.37 Selective views ........................................................ 100

5 Sequence diagrams ...................................................... 102
  5.1 Introduction ............................................................ 102
  5.2 Simple sequence diagrams ............................................ 102
    5.2.1 The time axis ..................................................... 103
    5.2.2 The objects ....................................................... 103
      5.2.2.1 The life line ................................................. 104
      5.2.2.2 The activation bar ......................................... 104
Technology-Neutral Analysis and Design using UML and URDAD

5.2.3 Service requests .................................................................................. 104
  5.2.3.1 Message to self ........................................................................... 104
5.2.4 Returns ............................................................................................... 104
5.2.5 Levels of granularity ........................................................................... 104
5.3 Message types ......................................................................................... 104
5.4 Timing constraints .................................................................................. 106
5.5 Interaction references ............................................................................ 107
5.6 Conditional flow (alt) ........................................................................... 108
5.7 Iteration in sequence diagrams (loop) ..................................................... 109
5.8 Concurrcencies in sequence diagrams (par) .......................................... 111

6 Activity diagrams .................................................................................. 112
  6.1 Introduction ........................................................................................... 112
  6.2 Basic activity diagrams ........................................................................ 112
    6.2.1 Activity and actions ....................................................................... 112
    6.2.2 Edges and events ......................................................................... 113
    6.2.3 Events and automatic transitions ................................................... 113
    6.2.4 Entry and exit activities .................................................................. 113
  6.3 Decision and merge nodes .................................................................... 113
    6.3.1 Formulating the conditionals .......................................................... 114
    6.3.2 Merge nodes .................................................................................. 114
  6.4 Activity partitions (swim lanes) ............................................................. 115
  6.5 Object flow in activity diagrams ............................................................ 116
  6.6 Structured and nested activities ............................................................. 117
  6.7 Concurrcencies in activity diagrams .................................................... 119
    6.7.1 Forking ......................................................................................... 119
    6.7.2 Flow final node ............................................................................ 120
    6.7.3 Synchronization .......................................................................... 120
  6.8 Sending and accepting signals ............................................................... 121
    6.8.1 Interruptible activities ................................................................... 122
  6.9 Object pins ............................................................................................. 123
  6.10 Expansion regions ............................................................................... 123
  6.11 Exception and error handlers ............................................................... 124
  6.12 Call operations, activity parameters and assigning behaviours/processes to services .................................................. 125
    6.12.1 Call operations ............................................................................ 125
    6.12.2 Assigning an activity/process to a service .................................... 126

7 Communication diagrams ........................................................................ 127
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.4.1</td>
<td>Introduction</td>
<td>145</td>
</tr>
<tr>
<td>10.4.2</td>
<td>The analysis phase</td>
<td>146</td>
</tr>
<tr>
<td>10.4.2.1</td>
<td>Introduction</td>
<td>146</td>
</tr>
<tr>
<td>10.4.2.2</td>
<td>Functional requirements</td>
<td>146</td>
</tr>
<tr>
<td>10.4.2.2.1</td>
<td>Pre-conditions, post-conditions and quality requirements</td>
<td>147</td>
</tr>
<tr>
<td>10.4.2.2.1.1</td>
<td>Bibliography</td>
<td>147</td>
</tr>
<tr>
<td>10.4.2.3</td>
<td>User work flow</td>
<td>148</td>
</tr>
<tr>
<td>10.4.2.4</td>
<td>The services contract</td>
<td>148</td>
</tr>
<tr>
<td>10.4.3</td>
<td>The design phase</td>
<td>149</td>
</tr>
<tr>
<td>10.4.3.1</td>
<td>Defining the use case contract</td>
<td>149</td>
</tr>
<tr>
<td>10.4.3.2</td>
<td>Responsibility identification and allocation</td>
<td>149</td>
</tr>
<tr>
<td>10.4.3.3</td>
<td>Business process specification</td>
<td>150</td>
</tr>
<tr>
<td>10.4.3.3.1</td>
<td>The activity diagram specifying the business process for the processClaim service</td>
<td>151</td>
</tr>
<tr>
<td>10.4.3.4</td>
<td>Projecting out the collaboration context</td>
<td>151</td>
</tr>
<tr>
<td>10.4.4</td>
<td>Transition to next level of granularity</td>
<td>152</td>
</tr>
<tr>
<td>10.4.4.1</td>
<td>Facilitating navigation across levels of granularity</td>
<td>153</td>
</tr>
<tr>
<td>10.4.4.2</td>
<td>Bibliography</td>
<td>153</td>
</tr>
<tr>
<td>10.5</td>
<td>URDAD views</td>
<td>153</td>
</tr>
<tr>
<td>10.6</td>
<td>How are the design activities realizing the desired design attributes embedded in URDAD?</td>
<td>154</td>
</tr>
<tr>
<td>10.7</td>
<td>Evaluating an URDAD based design</td>
<td>154</td>
</tr>
<tr>
<td>10.8</td>
<td>The URDAD profile</td>
<td>154</td>
</tr>
<tr>
<td>10.9</td>
<td>URDAD model organization</td>
<td>154</td>
</tr>
<tr>
<td>10.9.1</td>
<td>Why is the model organization important?</td>
<td>154</td>
</tr>
<tr>
<td>10.9.2</td>
<td>Guiding principles for model organization</td>
<td>155</td>
</tr>
<tr>
<td>10.9.3</td>
<td>URDAD rules for model organization</td>
<td>155</td>
</tr>
<tr>
<td>10.9.4</td>
<td>Example of the organization of an URDAD model</td>
<td>156</td>
</tr>
<tr>
<td>10.10</td>
<td>URDAD documentation generation</td>
<td>157</td>
</tr>
<tr>
<td>10.10.1</td>
<td>Introduction</td>
<td>157</td>
</tr>
<tr>
<td>10.10.2</td>
<td>Why documentation generation?</td>
<td>157</td>
</tr>
<tr>
<td>10.10.3</td>
<td>Types of documentation</td>
<td>158</td>
</tr>
<tr>
<td>10.10.3.1</td>
<td>Documentation required by business</td>
<td>158</td>
</tr>
<tr>
<td>10.10.3.2</td>
<td>Documentation required by business analysts</td>
<td>158</td>
</tr>
<tr>
<td>10.10.3.3</td>
<td>Documentation required by architecture</td>
<td>158</td>
</tr>
<tr>
<td>10.10.3.4</td>
<td>Documentation required by developers</td>
<td>159</td>
</tr>
<tr>
<td>10.10.3.5</td>
<td>Documentation required by quality assurance</td>
<td>159</td>
</tr>
<tr>
<td>10.10.3.6</td>
<td>Documentation required by service providers</td>
<td>159</td>
</tr>
<tr>
<td>10.10.3.7</td>
<td>Documentation required by operations</td>
<td>159</td>
</tr>
<tr>
<td>10.10.3.8</td>
<td>Documentation required by users</td>
<td>159</td>
</tr>
<tr>
<td>10.10.4</td>
<td>Documentation generation approaches</td>
<td>159</td>
</tr>
</tbody>
</table>
## List of Figures

1.1 MDA based approach ................................................. 4
1.2 An iterative, model driven software development process .............. 7

3.1 Simple use case diagrams .............................................. 17
3.2 A user can access multiple use cases .................................. 18
3.3 A vending machine provides different services to customers and maintenance operators ..................... 19
3.4 A switch which has been stereotyped as a control object ............... 20
3.5 Users, service providers and observers as the core actor types ........ 21
3.6 User, service provider and observer for a vending machine .......... 21
3.7 Extend relationships for conditional functional requirements ........... 23
3.8 Extension points ....................................................... 24
3.9 Using use case abstraction to define the scope of an ATM ................ 25
3.10 Using use case abstraction to specify commonalities across use cases ................................. 26
3.11 Specialized users have access to the services the more abstract/generic users can access ................. 27
3.12 Use case diagram ...................................................... 28

4.1 Simple UML class and object diagrams .................................. 31
4.2 Adding an attributes compartment to a class. .......................... 33
4.3 Multiplicity constraints on attributes ................................... 34
4.4 Derived attribute and the constraints specifying how their values are determined from the current value of the other attributes .................................................. 34
4.5 Services are shown in a separate services compartment .................. 37
4.6 Specifying the role names for the different input objects. .................. 38
4.7 A service without any input objects. .................................... 39
4.8 Compound return values ................................................. 39
4.9 Services need not have a return type. ................................... 40
4.10 Parameters may be specified as in, out and inout ......................... 40
4.11 Different business processes may be followed for different input parameters ................................. 41
4.12 A service for an account class ......................................... 41
4.13 The camel naming convention. ......................................... 44
4.14 An interface specifying the services required from an assessor. ............ 45
4.15 Estate agents and the property sales register both realize the services required from a property valuator.

4.16 Home loans is decoupled from any concrete realization of a property valuator.

4.17 The property valuator interface collapsed into its stereotype icon.

4.18 Showing the required and provided sides of an interface.

4.19 The property valuator as a required interface for home loans.

4.20 The function organizer extending the function venue provider interface.

4.21 The function agent as a join interface.

4.22 A simple services contract for a caterer.

4.23 A services contract for a message sender.

4.24 CreditCardAccount is a specialization of Account.

4.25 Both, credit card and electronic payments are special types of payments.

4.26 One may substitute a credit card account for an account when requesting to raise the subscription fee for a particular subscription number.

4.27 CreditCardAccount inherits all attributes and services of Account.

4.28 For all payments we capture the payment amount and date.

4.29 Different business processes will be followed for different types of loan applications.

4.30 A class hierarchy of various levels of abstract chargeables with concrete leaf chargeables.

4.31 Orders are processed against any current account.

4.32 An abstract PropertyValuator class with an abstract valuation service.

4.33 A personal card inherits all members of both, identity cards and driver’s licenses and is substitutable for both.

4.34 Complete constraint on a service prevents the service from being overridden.

4.35 Complete constraint on a class prevents specialization of that class.

4.36 A simple example of a UML specialization relationship.

4.37 An abstract graphics object class.

4.38 Mapping multiple inheritance relationships onto Java.

4.39 A class hierarchy for persons.

4.40 Using aggregation and interfaces for person roles and employee responsibilities.

4.41 A class hierarchy for accounts.

4.42 Using a delegation based model for accounts.

4.43 An accounts collection which may contain different concrete accounts.

4.44 The basic notation for a unary association.

4.45 Role names, association labels and cardinalities.

4.46 An association facilitating the location of associated information.

4.47 Decoupling from a service provider via interfaces.

4.48 A peer-to-peer relationship can be documented using bi-directional associations.

4.49 Decoupling peer-to-peer relationships via interfaces.

4.50 A credit card as an association class.

4.51 A property agent as an bi-directional association class.
6.6 Using a structured activity to show common transitions for a game .......................... 118
6.7 Using a structured activity to show transaction cancellation ........................................ 119
6.8 Sending messages in a background thread ................................................................. 120
6.9 An activity diagram for the processing of a claim ...................................................... 121
6.10 Sending and receiving signals within a business process .............................................. 122
6.11 Object pins showing the inputs and outputs across the activities for preparing a meal. .... 123
6.12 An expansion region showing the kitchen preparing the order items concurrently ......... 124
6.13 Specifying a transition to an error handler ............................................................... 125
6.14 Call operation ............................................................................................................ 126
6.15 The business process for processing a sale ................................................................. 126

7.1 Communication diagram for the buy product use case of a vending machine ................. 127
8.1 The objects participating in the buyProduct collaboration .............................................. 129
8.2 Ports through which the restaurant interfaces with the environment ............................. 130
8.3 The internal parts and the ports of a vending machine .................................................. 131

9.1 Components are pluggable classes .................................................................................. 132
9.2 Artifact documenting the manufacturing process for a vending machine .................... 133
9.3 High level component diagram for a vending machine .................................................. 134

10.1 High level view of URDAD in the context of a model driven approach ......................... 137
10.2 High level view of URDAD in the context of a model driven approach ......................... 139
10.3 High-level view of the URDAD methodology ............................................................. 145
10.4 More detailed outline of the URDAD methodology ...................................................... 146
10.5 Functional requirements for the process claim use case .............................................. 147
10.6 The user work flow for a success scenario of the use case ........................................... 148
10.7 The services contract for the process claim service ...................................................... 149
10.8 Responsibility identification and allocation for the process claim use case .................. 150
10.9 Activity diagram showing how the controller assembles the business process across services sourced from service providers ................................................................. 151
10.10 The collaboration context for the process claim use case ........................................... 151
10.11 Functional requirements for the provide settlement offer service ............................... 152
10.12 Responsibility allocation for the provide settlement offer service ............................. 153
10.13 Example of the organization of an URDAD model for an insurer ............................... 156
10.14 The stakeholder requirements diagram for the processClaim use case ......................... 162
10.15 The user work flow for the processClaim use case ..................................................... 164
10.16 Data structure (class) diagram for Claim ................................................................. 164
10.17 Data structure (class) diagram for ClaimSettlementReport ....................................... 165
10.18 The service contract for the processClaim use case .................................................. 165
10.19 The responsibility allocation diagram for the processClaim use case ......................... 165
10.20 The business process specification diagram for the processClaim use case ................. 166
10.21 The collaboration context diagram for the processClaim use case ............................ 166
List of Tables

4.1 Objects can be identified from nouns. ................................................................. 30
4.2 Classes as abstraction of specific objects. .......................................................... 31
10.1 Stake holders in the technology neutral (business) model and their quality requirements ......................................................... 141
10.2 Analysis and design principles supporting model qualities ................................. 143
Chapter 1

Introduction to business analysis

1.1 Introduction

Business analysis has established itself as a core responsibility through which organizations aim to better realize stake holder requirements. Business analysis helps an organization to improve how it conducts its functions and activities in order to generate improved stake holder value.

1.2 Definition of business analysis

Very widely speaking business analysis is the mediator between the stake holders of the organization who extract value from the organization and the architecture and development arms of the organization.

Architecture is responsible for ensuring that the organization has a suitable infrastructure within which it can deploy the business processes through which it realizes the stake holder services. Architecture includes organizational and systems architecture.

Development is responsible for implementing new business processes across the organizational and systems architecture and for implementing changes to existing business processes. This includes software development which implements business processes within IT systems.

Note

The International Institute of Business Analysis defines the role of a business analyst as follows:

A business analyst works as a liaison among stakeholders in order to elicit, analyze, communicate and validate requirements for changes to business processes, policies and information systems. The business analyst understands business problems and opportunities in the context of the requirements and recommends solutions that enable the organization to achieve its goals.

1.3 Sub-disciplines of business analysis

Business analysis is a wide field within which there are a number of specialization domains. According to the business analysis body of knowledge as maintained by the International Institute of Business Analysis there are six specialization areas for business analysis:

- Enterprise analysis Enterprise analysis “focuses on understanding the needs of the business as a whole, its strategic direction, and identifying initiatives that will allow a business to meet those strategic goals.” This domain includes the maintenance of the vision and mission of the organization and the responsibility of ensuring that the organizational architecture is aligned with this vision and mission, i.e. that it provides a suitable infrastructure within which the vision and mission can be realized.
• **Requirements planning and management** Requirements planning and management “involves planning the requirements development process, determining which requirements are the highest priority for implementation, and managing change.”

• **Requirements elicitation** Requirements elicitation “describes techniques for collecting requirements from stakeholders in a project.”

• **Requirements analysis and documentation** Requirements analysis and documentation “describes how to develop and specify requirements in enough detail to allow them to be successfully implemented by a project team.” This domain includes business process design -- the technology neutral business process design provides a complete view onto the functional requirements.

• **Requirements communication** Requirements communication “describes techniques for ensuring that stakeholders have a shared understanding of the requirements and how they will be implemented.”

**Note**

UML based business process design and requirements specification may assist in being able to effectively communicate requirements.

• **Solution assessment and validation** Solution assessment and validation “describes how the business analyst can verify the correctness of a proposed solution, how to support the implementation of a solution, and how to assess possible shortcomings in the implementation.”

**Note**

Having a contract driven approach where the requirements at any level of granularity are contained in a use case contract facilitates assessment and testing.

### 1.4 Responsibilities of business analysis

The responsibilities of the business analysis domain include

• **Assist in formulating a business case** Business analysis usually assists in formulating the business case for a new service or changes to an existing service.

• **Elicit and documenting stake holder requirements** Business analysis needs to capture the detailed requirements around any service or use case the organization aims to provide to any of its users (e.g. clients, investors, employees, ...). For each use case it needs to identify the stake holders which have an interest in that use case as well as the detailed requirements they have for that use case.

• **Design and document technology neutral business processes** Business analysis is responsible for designing business processes without concern about the implementation infrastructure and technologies. These business processes would then be mapped by development (including software development) onto the current choice of implementation technologies.

**Note**

System requirements can be extracted from the technology neutral business process design by projecting out those aspects of the business process which must be realized within the system.

• **Assist strategic management to define the organization’s vision and mission** Having an understanding of the stake holder requirements for the organization, business analysis can assist strategic management in defining the vision and mission of the organization.

• **Design and document the architecture of the organization** Business analysis can assist organizational architecture to define a conceptual organizational architecture. This architecture is then mapped by management and the technical team onto an implementation architecture.
1.5 What is software design?

Software design involves designing a solution which realizes a set of use services (use cases) satisfying the functional requirements for these use cases. The design may be mapped onto a software implementation.

There are significant benefits in designing in a technology neutral way, generating a design which can be mapped onto different implementations in different technologies and programming languages or even onto a manual realization.

1.6 Architecture versus design

At times architecture is seen simply as high level design. Such a differentiation would, however, be quite arbitrary and would also have the consequence that the skills requirements and tasks for architecture would not differ significantly from those of design.

Amnon Eden and Rick Kazman [EK_2003_ADI] formalized the distinction using the locality criterion stating that architectural statements describe properties of the entire system (non-local) whereas design specifications pertain only to a limited part thereof.

In particular, architecture provides the infrastructure within which the components realizing the functionality are deployed. It provides a framework within which the work product realizes its quality requirements, ensuring that these are addressed across the various usage scenarios (services or use cases) of the work product.

Design, on the other hand, is driven by the functional requirements for the work product, generating a solution realizing its various usage requirements. This includes

- functional responsibility distribution across functional components (lower level service providers),
- interface specifications, and
- the specification of work flows and algorithms.

A further difference between architecture and design elements is that the latter typically change at a much higher frequency than the former. The architecture or infrastructure of a work product typically only changes when

- the vision for the work product changes due to changes in either
  - the scope of services to be deployed in the architecture changes, or
  - the quality requirements for the work product,
- or when stake holders feel that the architecture has not succeeded to address their requirements.

Both of these happen much less frequently than changes to functional or usage requirements of the work product. Users will continuously change the functional demands made on a work product.

For example, one family may use a house for a holiday home, while another may use the same house as their prime residence. The functional requirements may be quite different (one family may use a room as a study while the next may use it as another bedroom). Furthermore, the design of the kitchen or bathroom may change quite regularly without touching the architecture of the house.

1.6.1 Bibliography

1.7 Model driven approach to software design

The Object Management Group (OMG) released the Model Driven Architecture (MDA) in 2001. The aim of MDA is to promote model driven development with a separation of the technology neutral business process design from the implementation architecture and technologies.

MDA requires that the design is done in an architecture and technology neutral way resulting in a Platform Independent Model (PIM). This PIM is then mapped onto an implementation in the target architectures and technologies resulting in a Platform Specific Model (PSM).

The PSM contains all the information about the business process specification as well as how the business process is to be mapped onto the target architecture and technologies. The last step specified by the MDA is then the actual implementation mapping. The output of this phase is the Enterprise Deployment Model (EDM). The EDM contains the implemented code as well as any other system artifacts required for the deployment and running of the system. The EDM can be deployed into production.

1.7.1 Automatic mappings

MDA envisages that both, the mapping of the PIM onto the PSM as well as the mapping of the PSM onto the EDM can be automated. This is aimed to

- significantly reduce the cost of introducing changes to the technology and architecture, and
- enable organizations to more rapidly extract value from emerging technologies.
1.7.2 Independent business process and architecture optimization

One of the core benefits of a model driven approach is that it enables one to independently vary the business process specification and the architecture into which the business process is to be deployed.

1.8 Responsibilities of software design

The responsibilities of software design include

- designing a software solution satisfying the requirements,
- effectively communicating the design to the team or system responsible for performing the implementation mappings onto the implementation architecture and technologies,
- validating that an implementation does indeed comply to the design specification, and
- assessing the impact of requirements changes on the design.

1.9 Challanges faced by software design

Effective software design has proven to be difficult due to not only the intrinsic challenges of software design itself, but also due to a wide range of environment challenges.

1.9.1 Design challenges

Software design is intrinsically difficult. One needs to generate a "good design" which requires that one needs to understand what differentiates a "good design" from one which we would regard as a design of lower quality.

Then we need to be able to validate a design in order to establish that the design does indeed provide a solution to the requirements.

Finally we need to be able to effectively communicate the design to the team or system responsible for mapping the design onto a software implementation in the specified target architecture and technologies.

1.9.1.1 Generating a "good design"

Before we can claim to have generated a "good design" we need to understand what makes one design better than another. Certainly simplicity is core. If there are two design solutions which are otherwise equal, than the simpler solution is preferable. Complexity adds risk and cost/ This is not only for the initial design and implementation, but also over the entire life span of the system - i.e. while the system needs to be maintained.

Another widely accepted criterion for a good design is that it should exhibit good responsibility localization, i.e. a high level of cohesion. Preferably each object should only have a single responsibility and all its services should fall within its responsibility focus.

A good design also yields a high level of re-usability, i.e. should generate reusable components. This will reduce the long-term development and maintenance costs and also the risk (both, the risk associated with the additional development required if one does not achieve a high level of reuse and inconsistency risk in that different system elements may perform the same processing step in different ways.

A good design should preferably be such that the architecture and implementation technologies can vary independently, i.e. that one can change the architecture and implementation technologies without having to change the design and vice versa. This ensures that a design is implementable within different architectures and technologies and that technology and architecture changes do not require that the design needs to be redone.
1.9.1.2 Requirements tracing

One would like to be in a position where any design element can be traced back to an aspect of the requirements. This enables one to

• determine which aspects of a system are impacted on a change in the requirements,
• the aspects of a system which have become obsolete due to changing requirements, and
• the potential causes of a particular requirement not being realized correctly (bug tracing).

1.9.1.3 Validating a design

It is beneficial if one can validate whether a design does indeed provide a solution to the requirements - i.e. before the design is actually implemented. Otherwise one is only able to do validation and testing once the implementation mapping has been completed.

1.9.1.4 Communicating a design

The design needs to be communicated to the team or system responsible for the implementation. Using a semantically solid design language like the Unified Modeling Language (UML) makes communication of the design simpler and less error prone. Most developers are these days at least reasonably proficient in interpreting UML diagrams.

1.9.2 Environmental challenges

Software design operates between

• business analysis providing the use case requirements,
• architecture which provides the core software infrastructure, specifies the technologies to be used and the deployment environment for the software,
• implementation which includes the developers which map the software design onto an implementation within the specifies architecture and technologies.

Many of the challenges faced by software development originate on these interfaces. These are often communication problems.

1.9.2.1 Requirements challenges

Studies have shown that the majority of bugs in software originate from incorrect, incomplete or misunderstood requirements. It can be expected that the skills increase in the business analysis domain and the increased use of UML for requirements specification will slowly reduce these problems. Nevertheless, it can be expected that requirements risk will remain for some time the core risk for software design.

In addition to the static requirements risks, there are challenges around the requirements changing. Clients typically need to operate in a continuously changing environment. Software design must be able to effectively operate in an environment where requirements change very frequently.

1.9.2.2 Communication challenges

Ultimately a design needs to be effectively communicated to the team responsible for its implementation. This may be a team of programmers who map the design onto an implementation in the chosen architecture and implementation technologies. As the model driven tools (MDA tools) mature, one can expect that more and more of the mapping can be automated.
1.10 Typical model-driven software development process

Moderns software development approaches are virtually exclusively iterative, with each iteration realizing or modifying a particular use case realized by the system. As such, most modern software development approaches are use case driven.

![Diagram of an iterative, model-driven software development process]

The input for an iteration should thus be a use case contract which specifies the functional and non-functional stake holder requirements around the use case. The use case contract is provided to the design team, the architecture team and to quality assurance.

Given the use case contract, quality assurance can go ahead to define the test cases for the use case.

In the mean time architecture can assess whether the current systems architecture can effectively address the quality requirements for the use case. If not, it is responsible for making the appropriate architectural adjustments.

The design team design also receives the use case contract in order to design a business process realizing the functional requirements. The design process yields a technology neutral business process design, the PIM.

Once architecture and design have completed their work, implementation can go ahead and map the technology neutral business process design onto the architecture and technologies specified by architecture.

The resultant implementation is tested by quality assurance before handing the use case over to operations for deployment. If the system does not provide the required functionality, the design team is requested to apply the required design changes. Having done so, implementation will have to update the implementation mapping before handing the use case over for testing again.

If the system does not deliver the required qualities, architecture is requested to insure that the required architectural optimizations are done. At times architecture may identify certain design decisions or even certain implementation mappings causing inferior system quality. If this is the case, architecture will raise these issues with the design and implementation team respectively. Often, however, the increased quality may be achieved through architectural adjustments. Once these have been applied, the implementation team will have to update the implementation mapping before handing the system back for testing.
1.11 Object-oriented concepts

1.11.1 Introduction

An understanding of the core object oriented concepts is required for any form of object oriented modeling. This is irrespective of whether one uses an object oriented approach to organizational modeling, business process design, system design or software development.

1.11.2 What is a use case?

A use case is a service of value, i.e. a service offered to a client which provides some value to the client.

There may be significant amount of interaction between the client and the service provider around realizing a use case. Furthermore, there may be multiple scenarios, some of which may be success scenarios where the client does receive the value from the use case and others which may lead to the service being refused for some or other reason.

1.11.3 What is an object?

An object is a discrete conceptual or physical entity with a well defined boundary which encapsulates state and behaviour.

In particular, an object

- has identity,
- may have certain properties,
- may have internal state,
- may offer services, and
- may have relationships with other objects.

Examples of objects include an apple, an organization, a person, a system, a system component, a loan application and an abstract concept.

Generally one can state that whenever one uses a noun in natural language, one refers to an object.

1.11.4 What is a class?

Classes represent concepts and can be used to specify static aspects around these concepts. They are abstraction of objects and encapsulate commonalities across all objects of that class, i.e. across all instances of the class.

Commonalities which may be defined for a class include

- common user properties,
- common internal state,
- common services offering,
- common processes followed when realizing a particular type of service request, and
- common relationships to other objects.

Any object which is an instance of the class will thus have the properties and behaviour as specified for the class.
1.11.4.1 Assigning process to services

Introducing classes, enables one to specify the processes which should be followed for its services. These could be business or system processes. When a user requests a service from an instance of the class, the instance will follow the process for that service as defined for the class.

1.11.4.2 Classes and substitutability

Since a class is an abstraction of an object, it can be used to decouple from any particular object, i.e. in cases where the dependency is on a class, one can substitute one instance of a class with another.

For example, one may make the following statement:

_Hire a car and drive to your parents._

This is a statement made at the class level. When executing the request one may substitute any object which is a car and hence has the properties and behaviour of a car.

1.11.5 Interfaces

An interface defines a coherent set of public features which must be provided by any object realizing or implementing the interface. An interface is often used as the core of a services contract.

The required public features include public properties and services, i.e. externally visible properties and services which are accessible to the users of an object realizing the interface.

1.11.6 Components

A component is a special type of class that represents a replacable unit for some subject (e.g. organization or system).

A component thus specifies a modular element of some higher level object which is pluggable. It can thus be exchanged with another component fulfilling the same requirements.

The requirements are usually encapsulated within a services contract. A component is a pluggable class which realizes that contract. In order to realize the contract, the component must

- provide an implementation of the required services (typically specified in an interface),
- and realize the functional and non-functional requirements around these services.

1.11.7 Contracts in object oriented modeling

Interfaces alone do not guarantee pluggability. A service provider may realize all services specified in an interface, but may, for example, not provide all deliverables (satisfy not all post-conditions) or may not be able to provide the scalability, performance or some other quality requirement.

Contracts are introduced to fully specify the client requirements. This then facilitates true pluggability of different service providers, i.e. that service providers realizing the contract can be exchanged without affecting any of the business or system processes.

Using a contracts based approach, one can define the system or business process without referring to any specific service provider. The service provider selection is typically made either when the process is implemented or when it is executed. One may even fully define an organizational or business process without knowing whether the process will be realized manually or by a system. Such a process would then be mapped onto manual and system processes deployed across organizational and systems architecture.

In order to expand an interface into a services contract, one needs to add the pre- and post-conditions, quality requirements, communication channels and, if relevant, the message exchange patterns for each use case.
Part I

The Unified Modeling Language (UML)
Chapter 2

Introduction

The Unified Modeling Language (UML) has been widely adopted to document business processes, system requirements and system design. It is widely used by business analysts, architects, software designers and programmers.

2.1 What is UML

The Unified Modeling Language (UML) is a non-proprietary object oriented graphical modeling language maintained by the Object Management Group. UML uses a range of complementing diagrams to specify an object model. This object model may document, for example,

• a system, including
  – the services it offers to its users,
  – the system processes through which these services are realized,
  – the system structure supporting these processes, and
  – the environment within which the system is to be deployed,

• or an organization including
  – the services it offers to its clients and other users,
  – the business processes through which these services are realized,
  – the organizational structure hosting these business processes, and
  – the environment into which the organization is to be deployed.

2.2 History of UML

Prior to UML a wide range of object oriented modeling languages were used. UML grew out of the desire to introduce a standard, methodology neutral modeling language. In the mid 1990’s Rational Software Cooperation had acquired some of the most well known object oriented methodologists, Jim Rumbaugh, Grady Booch and Ivar Jacobson. Rumbaugh was the core developer of the Object Modeling Technique (OMT), Grady Booch had developed the Booch method and Ivar Jacobson had developed the Object Oriented Software Engineering method, OOSE. Together with a range of organizations called the UML partners, Rumbaugh, Booch and Jacobson developed the Unified Modeling Language (UML) which was released in January 1997.

Over time the semantic consistency of the UML was improved resulting in UML 2.0 which was released towards the end of 2005. UML also acquired support for specifying rules via the Object Constraint Language (OCL). The OCL contribution made by IBM enables UML to specify formal contracts for service providers.
2.3 UML does not provide a design methodology

UML is a modeling language. It does not provide guidelines around how to analyze stake holder requirements or how to design a solution satisfying these stake holder requirements. Analysis and design methodologies like URDAD (Use Case, Responsibility Driven Analysis and Design) use UML to document the resultant analysis and design models.

2.4 Applicability of UML

UML is typically used to either

- capture the requirements for a subject, or to
- specify a design solution realizing these requirements.

The subject could be an organization or a system. If the subject is an organization, then UML can be used to specify, for example, the client requirements around the services an organization provides to the client as well as the business process which realizes these requirements.

The subject could also be a hardware or software system or a system containing hardware and software elements. In either case, UML can be used to capture the user requirements as well as the system design.

2.5 Overview of the UML diagrams

2.5.1 Introduction

UML is a diagrammatic language. The requirements for a subject and design solution realizing these requirements are thus communicated through a collection of complementing diagrams. Each diagram provides an additional view through which additional information is fed into the requirements or design model.

UML supports in total 13 types of diagrams. These diagrams can be grouped into different categories, each category focusing on a different aspect of the model. For example, class and composite structure diagrams are used to specify the structural aspects of the subject as well as the data structures of the information components exchanged between the role players.

Similarly activity diagrams, sequence diagrams, state charts, communication and timing diagrams are all used to specify the dynamics of a process and how the organizational or system components collaborate to realize the business or system process. Two of these, the sequence and communication diagrams, focus on the messages exchanged in the process of the role players (e.g. system or organizational components) collaborating to realize a use case. Activity diagrams and state charts, on the other hand, focus on the actual activities performed by these role players and the affect these activities have on their state. Timing diagrams put the message interchange patterns and the state in a formal timing framework.

2.5.2 Use case diagrams

Use case diagrams are used to provide an overview of the functional requirements for a subject. They are used to specify the scope of the subjects operations, the external objects the subject interfaces with, and an overview of the actual functional requirements around the services the subject provides to its users.

2.5.3 Sequence diagrams

Sequence diagrams were originally developed to provide an intuitive way to show the sequence of messages exchanged in a specific usage scenario. UML has extended the notation to support alternative flows, reuse of interaction patterns and concurrencies. Nevertheless, the core value of sequence diagrams lies in providing a very simple and intuitive way of showing an example of how objects collaborate to realize a specific usage scenario. Furthermore, they provide a natural notation for specifying time and duration constraints on processes.
Sequence diagrams are commonly used to validate business or system process scenarios. They also show the information or objects provided when a service is requested from a service provider, as well as what service providers return upon completion of a service.

### 2.5.4 Activity diagrams

Activity diagrams provide a very intuitive notation to capture work flow, i.e. the activities performed in order to realize a use case. The activities could be performed by a single object. However, often different role players collaborate to realize a use case. Activity diagrams can be used to show activities across objects, i.e. how the role players collaborate to realize the use case.

Unlike sequence and communication diagrams which are largely used to document specific examples of a system or business process, activity diagrams are used to specify the process in general. They have support for decision points, object flow, concurrencies and synchronization points in a process.

### 2.5.5 Interaction overview diagrams

Interaction overview diagrams shows for a collaboration realizing some use case the higher level flow across lower level interactions. Each interaction typically achieves some lower level goal relevant to the realization of the higher level use case.

**Note**

An interaction is a sequence of messages exchanged between a selected set of participants over a finite time period.

The interaction overview diagram is similar to an activity diagram in that it shows alternate and concurrent flow. The elements are, however, not activities, but interactions specified using sequence, communication or timing diagrams.

### 2.5.6 Class diagrams

Class diagrams use classes and interfaces to specify the structure of object types and the static relationships between them. One of the strengths of class diagrams is that they are able to effectively communicate even complex structure.

They can be used to specify or document the structure of organizational or system components. In addition to the above, class diagrams are also commonly used to specify the core of a services contract.

### 2.5.7 Object diagrams

Object diagrams are used to refer to or document specific objects, i.e. specific instances of classes. In particular, they can show a snapshot of the state of an object (at some time instant).

Object diagrams are also commonly used when referring to an object participating in an example scenario of a business or system process.

### 2.5.8 State charts

State charts are used to track the state of an object through a business or system process. They expose the states of the object as well as the state transitions and the events which cause them.

UML state charts extend Harel state charts by adding

- nested states with shared state and common transitions, as well as
- composite states with concurrencies.
2.5.9 Timing diagrams

Time diagrams are special types of sequence diagrams which show not only the sequence of messages exchanged between objects, but also show the various states the objects reside in, the state transitions and the events which cause them. Like sequence diagrams, they also support time and duration constraints.

2.5.10 Communication diagrams

Communication diagrams provide an alternative view on for the information provided in sequence diagrams. Like the former they show the messages exchanged within usage scenarios. They do, however, expose more clearly the communication paths required between the objects participating in the collaboration.

2.5.11 Component diagrams

A component is pluggable class which implements some services contract or interface. It represents a particular implementation of an internal or external service provider required by the subject (e.g. system or organization).

Component diagrams are then used to show a particular implementation of a subject, exposing the selected service provider types. Typically component diagrams will show the interfaces they implement as well as the dependency between the components.

Component diagrams are often used by architects to specify a high level decomposition of a system or an organization.

2.5.12 Composite structure diagrams

Composite structure diagrams are used to show the internal structure of cohesive objects and the collaborations that this structure supports. These collaborations realize the use cases or services offered by the object.

Composite structure diagrams expose the internal parts of an object, the ports through which these parts interact with one another or with external objects, as well as the connectors between these parts.

2.5.13 Package diagrams

UML models can become quite elaborate, containing a lot of information. One needs mechanisms which enable one to rapidly find certain UML diagrams and model elements. One such mechanism is a hierarchical packaging structure similar to the file system hierarchy one traverses using a file manager.

Any model element would ultimately be packaged in some or other package.

Package diagrams can be used to show

- the nesting of packages (i.e. elements of the package hierarchy),
- the content of a package (classes, interfaces, diagrams, other packages, ...), and
- dependencies between packages.

2.5.14 Deployment diagrams

The deployment diagrams document at how the subject is to be deployed within a particular environment. It shows high level components, the contracts they realize, the nodes onto which these components are deployed and the communication paths between them.
2.6 The UML model

Even though UML is a diagrammatic language, the core information is ultimately contained within a single UML model. The various diagrams feed information into the model and publish selected information from the model. One can thus see the diagrams as a user interface into the model.

The UML model contains the full semantic information across the various diagrams. It ensures that the information captured across the various diagrams is consistent.

There is thus no need for any particular diagram to show all aspects of the model. One can expose only that subset of information which is relevant to what one would like to communicate through that particular view.

Note
A UML tool will allow a user to delete aspects from a diagram without removing those aspects from the model.

2.7 UML and MDA

MDA, the Model Driven Architecture, provides a framework for model driven development which separates architecture and technology choices from the design solution. The core design in MDA is thus a technology and architecture neutral design.

The technology neutral design solution would then be mapped onto one’s choice of implementation technologies and architecture (infrastructure). The mapping would involve assigning responsibilities to either organizational components, system components or external service providers. Commonly organizational processes are deployed across all three.

In a model driven approach, changing the environment into which the organizational process is to be deployed (i.e. changing the organizational infrastructure, the technologies used or the outsourcing decisions) would not require changing the design. It would only require mapping the design onto the new realization infrastructure.
Chapter 3

Use-case diagrams

3.1 Introduction

Use case diagrams are used to look at some subject from the perspective of its stake holders. They are used to specify

- the functional/services requirements of the stake holders of the subject, and to
- the scope of a subject’s operations.

The subject could be an organization, a business unit, a system or a system component.

A use case diagram exposes the services required by users of the subject without looking at how the subject realizes these services. In addition it can be used to specify any mandatory or optional functionality required by any of the stake holders for that use case as well as any external objects (e.g. service providers) the subject should interface with when realizing the service.

Furthermore, use case diagram provide the ideal platform for defining the scope of a subject. Having understood and specified the scope enables one to assess whether a requirement for a new service is within or outside the scope of the subject.

3.2 Simple use case diagrams

3.2.1 Introduction

Simple use case diagrams the user roles and the use cases they make use of. In addition they may show the subject which is responsible for realizing these use cases.
3.2.2 Use cases

A use case is a service which provides the user some value. It represents a complete user service for which there may be significant interaction between the role players (e.g., the user and the subject).

A user may access multiple services. For example, the user of a watch may use the following services of a watch:

- Display time of day.
- Display current date.
- Measure elapsed time.
3.2.3 The subject

The subject is the object whose requirements are being explored and documented. It represents a domain with a well defined boundary. The boundary visually shown by drawing a bounding rectangle. The subject is responsible for realizing the use cases which fall within its domain, i.e. which are contained within its bounding rectangle.

The subject could be

- an abstract entity for which we are defining a services contract,
- an organization or a component of an organization, or
- a system/system component.

3.2.4 Communication links

When a user makes use of a service or use case offered by the subject, information or objects are exchanged between the user and the subject. The fact that a user participates in a use case is shown by drawing a solid line (an association) between the user and the use case.

3.2.5 User roles

A user role is a role played by an external object which makes use of one or more services/use cases offered by the subject. Different user roles may have access to different use cases.
Figure 3.3: A vending machine provides different services to customers and maintenance operators

For example, the vending machine in Figure 3.3 offers the buy product use case to customers. On the other hand, maintenance operators use the vending machine to set product prices, insert or remove products and to maintain the cash in the vending machine.

Note that the same object may, at different times, play different user roles. Should the maintenance operator no longer be able to resist the chocolates, (s)he may decide to play the role of a customer in order to be able to use the buy product use case.

### 3.3 Stereotypes in use case diagrams

A stereotype can be used to extend an existing model element by adding additional semantics to it. Applying a stereotype to a model element introduces a conceptual distinction of the basic model element. Normally this distinction implies a different type of usage.

For a stereotype one defines its application domain. Some stereotypes may be applied to only classes or use cases, others may apply across a range of model elements.

For example, one may want to introduce the concept of a control object which is different from any other object in the sense that it can be used to control other objects. The application domain of the control stereotype could be constrained to classes and interfaces. The control stereotype is, in fact, a standard stereotype available in UML.

A stereotype is shown in UML using french quotation marks or guillemets, <<< ... >>>. Alternatively or additionally a stereotype icon may be assigned to the stereotype. The standard stereotype icon for the <<<control>>> stereotype.
Figure 3.4: A switch which has been stereotyped as a control object

Figure 3.4 shows various notation for a switch to which the <<control>> stereotype has been applied. In either case the stereotype icon is shown.

3.4 Actors

3.4.1 Introduction

An actor is a role played by an external object in the context of a use case offered by some subject. Actors participate in the use case by exchanging information or objects with the subject.

The default stereotype icon for an actor is a stick man. This does not, however, imply that the actor of a system must always be a person. The actor could be another organization, a business unit which is not part of the business unit which is the subject of the use case diagram, an external system or system component or a person. The stick man representation can be used in either case.

Since an actor is a role played by an external object which is relevant for a use case, it does not prevent

- different objects playing the same role, or
- the same object playing multiple roles.

3.4.2 Actor types

One can distinguish between three core types of actors:

- **Users** The user is the role which makes use of a use case or service offered by the subject. It extracts the primary value from the use case, typically initiates the use case and, at times, may drive it.

- **Service providers** Service providers provide services to the subject, assisting it to realize its services or use cases to its users (e.g. clients or system users). They do not make use of the use case. The services provided by these service providers usually fall outside the scope of the subject.

---

Note

When mapping a business or system process onto an implementation one often introduces additional service providers.

- **Observers** An observer of a use case does not make use of the use case. Nor does it provide lower level services to the subject to assist it to realize the use case for its users. Instead, an observer receives certain deliverables from the subject in the context of it realizing the use case for its user. While users extract the core value from the use case, observers extract auxillary value from it.
Either of these roles, user, service provider or observer, can be played by external organizations, systems or people.

### 3.5 Example actors in system design

The standard actor types, users, service providers and observers, are directly relevant for system design. Consider, for example, the *buy product* use case of a vending machine which also makes teas and coffees.

The user of the *buy product* use case is the customer. The customer triggers the use case and obtains the primary value from the use case.

In order to make tea or coffee, the vending machine requires water. This is provided by an external service provider, a water supply valve. The water supply valve does not make use of the services of a vending machine. Instead it provides services to the vending machine -- it is a service provider.
Assume that for each purchase made, the vending machine needs to send a message to an accounting system. The accounting system does not use the services offered by the vending machine. Nor does it provide any services to it. Instead it simply receives information in order to update the general ledger -- it is an observer.

### 3.6 Functional requirements

#### 3.6.1 Introduction

Use case diagrams provide a notation through which one can specify functional requirements around a use case. Each of these functional requirements generates some value which is required by some stake holder. For this reason these functional requirements are modeled as lower level use cases.

#### 3.6.2 Mandatory functional requirements via the include relationship

A mandatory functional requirement or work flow step resembles a requirement which must be addressed in any success scenario of the use case, i.e. in any scenario where the user obtains the primary value from the use case.

In UML one can specify a mandatory functional requirement by inserting an "<<include>>" relationship which points from the use case to the functional requirement.

For example, for any success scenario of the withdraw cash use case offered by an automatic teller machine (ATM), the ATM will have to

- read the ATM card,
- capture the pin number from the user,
- validate the card and pin number with the bank,
- capture the withdrawal amount from the user,
- request the withdrawal from the bank,
- show a transaction confirmation,
- issue the cash, and
- return the card.

If any of the steps are not done for any scenario where the client obtains the primary value (the cash), then the use case does not fulfill the functional requirements as specified by the stake holders.

---

**Note**

The functional requirements are not true use cases, i.e. complete user services which provide value on their own value to the user. For example, the ATM does not offer to clients a service validate cards and pin numbers with the bank - a card holder would not decide that s(he) would like to use the ATM today to validate a pack of cards and their pin numbers with the bank. The validation of a card and the associated pin number is not a user service - it is solely a functional requirement around the use cases offered by the ATM.

#### 3.6.3 Conditional functional requirements via the extend relationship

Some functional requirements need to be addressed only under certain conditions. These can be documented by introducing an "<<extend>>" relationship pointing from the conditional functional requirement to the use case. The conditional is specified in square brackets.

Consider, for example, the following functional requirements around the withdraw cash use case:
1. If the client requests to select the types of notes, provide him the facility to do so.

2. Print a transaction confirmation if the printer is working.

Both of these are conditional (not mandatory) as we can have success scenarios (for which the user receives the cash) where these functional requirements are not met. If the user did not select the notes (s)he would like to receive, the ATM will use whatever is available. Similarly,

The requirement to show a transaction confirmation is mandatory, i.e. we may not have a success scenario where the client receives the cash for which the transaction confirmation is not shown.

3.6.3.1 Specifying the conditional

Conditionals are specified in UML by enclosing them within square brackets. A conditional represents the test of an if statement.

3.6.3.2 Specifying extension points

An extension point is a point (or a collection of points) in the work flow where extensions (the extra conditional functional requirements) become available. For example, if we introduce the following two extension points for the withdraw cash use case

1. User confirmation The point in the work flow where the user confirms the transaction

2. Bank confirmation The point in the work flow where the bank confirms the transaction.

then we can specify that the let user select notes extension should become available at the point where the user confirms the withdrawal and that the print transaction confirmation extension should be conditionally executed when the bank confirms the transaction. The UML notation for this is shown in Figure 3.8

Figure 3.7: Extend relationships for conditional functional requirements
3.6.4 Identifying functional requirements

Every functional requirement for a use case should fulfill some stake holder need or requirement. One would typically start by identifying for a use case all stake holders. Then one would identify for each stake holder the functional requirements for that use case. This can be documented in a use case diagram by inserting the <<requires>> dependencies between the stake holder and the functional requirements they introduce.

Ultimately one would like to have full traceability of any feature of a business or system process to the requirement it fulfills as well as to the stake holder who requires that functionality.

3.7 Use-case abstraction

3.7.1 Introduction

The ability to abstract is one of the primary intellectual tools through which we can simplify the conceptualization of complex domain. UML provides a notation for specifying more abstract use cases. This is used for

* defining the scope of the subject’s operations through abstract, high-level use cases,
* specifying common functional requirements and common actors across concrete use cases.

3.7.2 Scoping

Use case diagrams provide a useful mechanism to specify and manage the scope of a subject, e.g. to decide which services fall within the scope of an organization or system and which services should rather be assigned to external objects. These would be out-sourced to external service providers or assigned to external systems.

To specify a use case abstraction or generalization relationship in UML, one draws a solid line with a triangular head (a generalization relationship) pointing from the more concrete (specialized) use case to the more abstract or generic use case.
Figure 3.9: Using use case abstraction to define the scope of an ATM

Figure 3.9 shows that the high level use cases are the client and maintenance services. The scope of the client services is defined by the next level use cases, electronic banking and token vending. A request for a new client service which is not a specialization of either electronic banking or token vending would be declared out of scope (alternatively one could revisit the scope of the ATM). An example would be that of vending a product (like chocolates). Such a service could not be seen as a specialization of either of the use cases defining the client scope and hence would be out of scope.

3.7.2.1 Concrete use cases

The leaf use cases in the specialization tree (e.g. withdraw cash, buy bus ticket, ...) are the concrete use cases the users actually use. It is for these concrete use cases that business and system processes need to be defined. They also from the basis for iterative realization of client and/or system services.

3.7.3 Defining common requirements and common actors across use cases

Use case generalization or abstraction enables one to specify functional requirements and actors at various levels of abstraction. For example, in Figure 3.10 the mandatory functional requirement of verifying the card and pin with the bank applies to all client
services including the electronic banking services (e.g. withdraw cash, ...) and the token vending services (e.g. buy bus ticket, ...).

Similarly, the printing of a token must be done for all token vending use cases, but not for the electronic banking use cases. Furthermore, for any of the financial transaction related use cases, the user may choose to have a transaction confirmation printed.

A client can make use of any of the client services (all specializations of client service), all client services interface with the bank and all token vending services interface with a vendor.

An added benefit of use case generalization, beyond scoping and specifying common functional requirements and actors, is that it cleans up the use case diagrams considerably. Instead of having to draw an association from the client use case to each concrete use case, we just draw a single association to the generic client services use case. Similarly, we do not need to draw an <<include>> relationship from each concrete token vending use case to the mandatory functional requirement of having to print a token.

### 3.8 Actor abstraction

UML supports the concept of actor abstraction. This is enables one to introduce specialized roles which are substitutable for more basic roles. A specialized user can access all services which the more abstract/general user can access.
Figure 3.11: Specialized users have access to the services the more abstract/generic users can access

Consider, for example different user roles for an accounting system as shown in Figure 3.11. A financial clerk could issue invoices and credit notes, make and receive payments and capture vendor invoices. A financial manager could generate income statements and balance sheets and add accounts and users. However, since the financial manager has been modeled as a specialization of a financial clerk, the role is fully substitutable for that of the clerk. Consequently a financial manager can make use of the use cases available to the clerk like that of issuing an invoice.

Of course we could have added the links between the financial manager and the use cases available to financial clerks. However, this would not only make the diagram less readable, it would also have a different semantic meaning. In particular, if we add a new use case for a financial clerk, it would not mean that this service would also be available to financial managers. However, if we specify that a financial manager is a special type of financial clerk, it would mean that a financial manager would always have access to the use cases available for financial clerks.

Note
Actor specialization should be used with caution. In some cases it may be better to require financial managers to actively assume the role of a financial clerk in order to make use of any of the services meant for clerks.
3.9 Summary of the UML notation for use case diagrams

Figure 3.12 summarizes the UML notation for use case diagrams.

![Use case diagram](image)

- A use case is a service accessible to a user which provides value to the user.
- The subject is the actor which offers the use case and is responsible for realizing it.
- A generalized use case is an abstraction of concrete use cases used for abstracting and encapsulating common functional requirements and common actors.
- An <<inclusion>> relationship is used to specify a mandatory functional requirement which needs to be addressed for any success scenario of the use case.
- An <<extension>> relationship specifies a conditional functional requirement which needs to be addressed when some or other condition is met.
- An extension point is a point in the workflow where the extension becomes available or needs to be conditionally executed.
- A user is a primary actor who enacts use or the service (use case) and obtains the primary value of the use case.
- A specialized user is substitutable for a more generic user, obtaining access to all use cases which the general user has access to and may access additional use cases.
- A service provider is a role played by an external object which provides lower level services, assisting the subject to realize its use cases for its users.
- An observer receives auxiliary value from the use case by receiving messages or objects from the subject in the context of the subject realizing a use case for its user.
- A stakeholder has an interest in a use case and may place functional requirements around a use case.

Figure 3.12: Use case diagram

3.10 Exercises

1. Consider a system you have been/are going to be working on.

   (a) Provide a one or two paragraph description of the purpose and responsibilities of the system.

   (b) Identify the different user roles and the services (use cases) they require.

   (c) Perform use case abstraction in order to specify the scope of the system.

   (d) Select one concrete use case and do the following:

       • Identify the stakeholders who have an interest in (and hence may provide requirements for) the use case.

       • For each stakeholder, identify the mandatory and conditional functional requirements.

       • Identify any secondary actors (service providers and observers) for the use case.
Chapter 4

Class diagrams

4.1 Introduction

Class diagrams are used to model the static structure. This may include the static structure of

- an organization or a component of an organization,
- a system or a system component, or
- some domain object (like an account or an invoice).

In particular, class diagrams can be used to specify

- the services offered by the instances of a particular class,
- the attributes which each instance of a particular class will have, and
- the relationships between one instance of a class and instances of other classes.

These class diagrams are used during both, the analysis phase where the requirements around business or system services are established, as well as during the design phase where a solution for the requirements is designed.

The class diagrams do not contain any information about the dynamics of an object. However, one can later assign activity, state and interaction diagrams to the services in order to specify the system or business process which should be followed when realizing the service.

4.1.1 Objects and Classes

Objects and classes are the central concepts in object oriented modeling. They represent concrete entities and their abstractions.

4.1.1.1 Objects

The objects are specific discrete conceptual or physical entities from our modeling domain. Each object has a well defined boundary and may have a state which could change over time. Furthermore, an object may have behaviour and could offer services to other objects.
4.1.1.1 Common examples of objects

- The *organization* as a whole is an object with a well defined boundary and identity which encapsulates state and behaviour.
- A particular business unit like the *Finance department* is an object.
- An *external service provider* to whom the organization out-sources some functionality is an object.
- A *vehicle* is an object which has both, state and behaviour.
- An *invoice* and a *home loan application* are both objects which have state, but no behaviour.
- A system component which is busy processing some request is an object.

4.1.1.2 Identifying objects

A very simple, but effective way to identify objects for your modeling domain is to go through a natural language description of the domain and identify the nouns. You will have used a noun because for those concepts which represent objects (or classes). Often you would want to map the objects from your conceptual understanding of a domain onto objects used in your model of that domain.

Consider, for example, the following description of some domain:

Sam and Jill are two of our clients who have accounts with us. Sam has an account in Australian dollars while Jill has an account in South-African Rand. Both accounts record transactions with their corresponding transaction date in a statement. Clients can request statements over any period defined by a start date and an end date.

These nouns can be directly mapped onto the objects, yielding the following collection of objects:

<table>
<thead>
<tr>
<th>Sam</th>
<th>Jill</th>
<th>client</th>
<th>account</th>
<th>ausDollar</th>
<th>zaRand</th>
</tr>
</thead>
<tbody>
<tr>
<td>transaction</td>
<td>transactionDate</td>
<td>statement</td>
<td>period</td>
<td>startDate</td>
<td>endDate</td>
</tr>
</tbody>
</table>

Table 4.1: Objects can be identified from nouns.

4.1.1.2 Classes

In most cases one prefers to model at a more abstract level where the model elements are not specific objects, but abstractions of specific.

The first level of abstraction is called a class. It encapsulates the commonalities across a class of objects, i.e. across all instances of the same class. The commonalities may include common

- state aspects (variables),
- services, and
- common relationships to instances of other classes.

In addition, one can use behaviour and interaction diagrams to specify how instances of a class need to realize a service, i.e. to define the business or system process which needs to be executed when a service is requested.

Classes thus represent concepts or types and class diagrams are used to define the static aspects of these concepts.
4.1.1.2.1 Identifying classes

Even in natural language we often communicate using classes instead of objects. For example, you may say

*We issue invoices for purchases.*

or

*We will use a caterer to provide the refreshments for this conference.*

In either case you are not referring to a specific invoice or caterer with a specific identity but to a more abstract concept for which there are multiple concrete instances. We are talking about classes, not objects.

Note
Some of the nouns identified in Section 4.1.1.2 like client and account were actually referring to classes. Their instances would be objects.

In cases where the natural concepts are not classes, you can look for objects which have the same state and behaviour. Such objects may be instances of classes themselves.

Let us revisit the example discussed in Section 4.1.1.2. We could abstract the objects identified to the classes shown below:

<table>
<thead>
<tr>
<th>Client</th>
<th>Account</th>
<th>Currency</th>
<th>Transaction</th>
<th>Date</th>
<th>Period</th>
<th>Statement</th>
</tr>
</thead>
</table>

Table 4.2: Classes as abstraction of specific objects.

4.2 Basic object and class diagrams

The simplest form of a class diagram is a rectangle with the name of the class drawn in the rectangle. The name of the class should be chosen as to cleanly and precisely define the concept represented by the class.

An object diagram which specifies a specific instance of a class is shown as a rectangle with the object and/or class name separated by a colon. The colon represents an *instance of* operator. The object name and its class name are underlined in an object diagram to further differentiate an object diagram from a class diagram.

![Simple UML class and object diagrams](image)

Figure 4.1: Simple UML class and object diagrams

In Figure 4.1 the top row (Client and Account) represents classes, while the bottom row specifies instances of these classes. We are specifying that sam is an instance of the Client class.
Object diagrams are commonly used to document example scenarios of a state of a business process execution. In either case one often omits the object name. One thus simply specifies that one is working with some instance of a class without giving that instance a name. In such cases one still uses the instance of operator (the colon) to specify an instance of a client (:Client) or an account (:Account).

Less frequently one would refer to an object without specifying or showing its class. In such cases one provides only the object name and underlines it to differentiate the object diagram from a class diagram. For example, in Figure 4.1 we introduce jill as an object without specifying that she is an instance of a client.

### 4.3 Mapping basic class and object diagrams onto Java

A basic class diagram is mapped onto a basic class definition in Java. If the access level in UML is public, then so is the Java class:

```java
public class Account {}
```

The Java compiler will not leave the class quite as empty as that. The resultant class after the compiler insertions is as follows:

```java
public class Account extends Object {
    public Account() {}
    private Account this;
}
```

An object diagram refers to an instance with an instance name. This maps onto a reference variable name in Java:

```java
Account myAccount;
```

The instance itself must have been created at some stage via the new operator:

```java
Account myAccount = new Account();
```

### 4.4 Mapping basic class and object diagrams onto XML

In W3C XML Schema, a UML class is mapped to a complex type:

```xml
<complexType name="Account">
</complexType>
```

In XML the object name would map either onto

- a unique identifier, which would typically be a relative or absolute URI,

  ```xml
  <element name="myAccount" type="Account"/>
  ```

- A unique identifier within an object specified within an XML schema via an ID or preferably a key declaration.

- or, at a more abstract level, a role name identifying an object which plays a particular role with respect to another object. For example, below we give the client’s account the role name clientAccount:
The object or instances of classes would be the XML objects in some other XML data stream. For example, an XML object complying to the data type specification given in the above XML schema would look something like this:

```xml
<account>
  ...
</account>
```

Its object identity would typically be relative to the resource within which it is defined. Alternatively one could use a key definition in an XML schema to, for example, specify that the account number is a unique identifier for an account:

```xml
<account accNo="a111">
  ...
</account>
```

### 4.5 Attributes

Attributes are commonly used when modeling domain objects. An attribute represents a component of the class which may have state. The state of the component is part of the state of the owner. For example, an account has a balance. If the balance of the account changes, then the state of the account changes.

Attributes are shown in UML in a separate attributes compartment which is added below the compartment which hosts the class name.

![Figure 4.2: Adding an attributes compartment to a class.](image)

- One can add an attributes compartment to a class diagram to specify the attributes of instances of the class.
- The left diagram shows the balance and account number attributes of an account without specifying their types.
- The second diagram specifies that the account number must be an integer and that the balance is of type money.
- The third diagram introduces the concept of money and shows that money has an amount and a currency.

#### 4.5.1 Collection attributes and multiplicity constraints

At times an object may have a collection attribute which contains multiple instances. In such cases one can the required cardinality or cardinality range on the attribute type (the class). This is done using square brackets with the cardinality constraint specified within the bracket. Thus

```xml
telephoneNumbers:TelephoneNumber[3]
```

specifies that the `telephoneNumbers` attribute is a collection of 3 telephone numbers. Similarly
telephoneNumbers:TelephoneNumber[1..3]

specifies that between 1 and 3 telephone numbers are required.

If one wants to specify an open ended cardinality, then one can use a single star, *. For example, to specify that a claim has one or more claim items one uses

claimItems:ClaimItem[1..*]

If a star is used without specifying a lower bound for the cardinality range then zero is assumed for the lower bound. For example, to specify that an account has a transaction history with zero or more transactions, one can use transactions:Transaction[*]

As a more complete example, consider the specification of an order shown in Figure 4.3. The class diagram specifies that a claim has one claim number, one or more claim items, and either one or two contact details.

The contact details itself has between one and 3 telephone numbers, an optional email address, a postal address and a physical address.

4.5.2 Derived attributes

At times attributes are not independent of one another. The value of one attribute may depend on the value of the other attributes of the object. Such an attribute is called a derived attribute.

For example, whether an account is in an overdrawn state or not would depend on its current balance and overdraft limit.

As a more complete example, consider the specification of an order shown in Figure 4.3. The class diagram specifies that a claim has one claim number, one or more claim items, and either one or two contact details.

The contact details itself has between one and 3 telephone numbers, an optional email address, a postal address and a physical address.

As a more complete example, consider the specification of an order shown in Figure 4.3. The class diagram specifies that a claim has one claim number, one or more claim items, and either one or two contact details.

The contact details itself has between one and 3 telephone numbers, an optional email address, a postal address and a physical address.
4.6 Mapping attributes onto Java

A public attribute maps onto a query method, and if write access is permitted, also a set method. In addition one may also define an underlying private field. For example, a product with price and code attributes could map onto:

```java
public class Product {
    ...
    public int getCode() {return code;}
    public Money getPrice() {return price;}
    public void setPrice(Money newPrice) {price = newPrice;}
    private int code;
    private Money price;
}
```

4.6.1 Mapping collection attributes onto Java

A collection attribute will map onto a collection variable in Java. One may use an array or an instance of one of the collection classes for the collection variable. For example, a claim with 2 contact details and one or more claim items could be mapped onto Java as follows:

```java
public class Claim {
    public Collection<ClaimItem> getClaimItems() {return claimItems;}
    public ContactDetail[] getContactDetails() {return contactDetails;}
    private ContactDetails[] contactDetails;
    private Collection<ClaimItem> claimItems;
}
```

The multiplicity constraints often need to be enforced by the code.

**Note**
If there was an ordered constraint in the model, we would change the `Collection<LineItem>` to `List<LineItem>`.

4.6.2 Mapping derived attributes onto Java

Derived attributes are implemented using a query method which returns the value of the derived attribute as calculated from the values of the other attributes.

For example, an account with `availableFunds` and `overdrawn` as derived attributes could be mapped onto Java as follows:

```java
public class Account {
    ...
    public double getAvailableFunds() {return balance - minimumBalance;}
    public boolean isOverdrawn() {return balance < minimumBalance;}
```
...  
  private double balance, minimumBalance;
}

4.7 Mapping attributes onto XML

If the UML attribute type represents a simple XML type, the attribute can be mapped onto either an attribute or a sub-element. On the other hand, if the attribute type corresponds to a complex XML type, then the attribute needs to be mapped onto a sub-element. For example, a product may be defined in an XML schema document as follows:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<schema targetNamespace="http://www.solms.co.za/retail/products"
    xmlns:prod="http://www.solms.co.za/retail/products"
    xmlns="http://www.w3.org/2001/XMLSchema">
  <complexType name="Product">
    <sequence>
      <element name="price" type="prod:Money"/>
    </sequence>
    <attribute name="code" type="integer"/>
  </complexType>

  <complexType name="Money">
    <sequence>
      <element name="amount" type="decimal"/>
      <element name="currency" type="string"/>
    </sequence>
  </complexType>
</schema>
```

An XML instance complying to this specification of a product might look as follows:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<product code="1234">
  <price>
    <amount>100</amount>
    <currency>EURO</currency>
  </price>
</product>
```

4.7.1 Mapping collection attributes onto XML

The multiplicity constraints in UML map onto more limited multiplicity constraints in XML schema:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<schema targetNamespace="http://www.solms.co.za/insurance/claim"
    xmlns:claim="http://www.solms.co.za/insurance/claim"
    xmlns="http://www.w3.org/2001/XMLSchema">
  <complexType name="Claim">
    <sequence>
      <element name="clientReference", type="string"/>
      <element name="claimItem" type="claim:ClaimItem" maxOccurs="unbounded"/>
    </sequence>
  </complexType>
</schema>
```
4.7.2 Mapping derived attributes onto XML

Derived attributes should preferably not feature in an XML-schema based data type specification as XML parsers would not be able to validate the derived data. Instead the reporting of derived attributes should be left to reporting tools or to system representations of the XML objects (for example, to the Java representation obtained from an XML-Java mapping of the XML data).

4.8 Services

One can use a UML class diagram to show the services provided by any instance of the class.

For example, Figure 4.5 specifies two services provided by an insurer, that of providing a quote for a policy specification and that of processing a claim against an existing policy.

![Figure 4.5: Services are shown in a separate services compartment.]

4.8.1 Service inputs

Each service is identified by a name and the types of input objects which need to be provided when requesting that service. The service may have a list of parameters encapsulated within round brackets. By default, these parameters resemble input parameters, i.e. information or physical objects which the client provides to the service provider upon service request, but which are not returned to the client once the service has been completed.

The input parameters are objects which the service provider requires in order to be able to render the service. These may resemble information which must be provided to the service provider or physical objects. In our example the client will have to provide the policy specification and his/her personal details when requesting a quote for a policy.

The information conveyed with these objects is specified in separate class diagrams. We would thus add a class diagram for the PersonalDetails class specifying in detail the type of personal information the insurer requires as well as class diagram specifying the information which would be provided as the PolicyRequirements.
Note
For each input and return type one would introduce a class diagram to specify the information which is conveyed. For example, the class diagram for a Quote would specify the information which would be contained in the quote.

4.8.1.1 Assigning a business process to a service

UML allows one to attach a business or system process to a service. The latter would normally be specified using activity diagrams, sequence diagrams as well as potentially other dynamic model diagrams.

4.8.1.2 Assigning role names to the input objects

Figure 4.5 specifies that an instance of a claim (:Claim) must be provided when requesting the processing of a claim. The instance of the claim received is not given a role name. It simply plays the role of a claim. In cases where the role played by a provided object is obvious, one need not specify a role name for the input object.

However, in cases where the role of an input object is not obvious from the service signature, one needs to assign a role name to that object. This is often the case when multiple instances of the same classes are provided to the service provider upon service request.

For example, the shipper provides a service to ship a package given a package identifier and two addresses. Figure 4.6 uses role names to specify that the one address is the address where the package should be collected from and that the other address is the delivery address.

For example, the shipper provides a service to ship a package given a package identifier and two addresses. Figure 4.6 uses role names to specify that the one address is the address where the package should be collected from and that the other address is the delivery address.

Note
It is usually preferable to combine multiple parameters into a single higher level request parameter. For example, the packetID, collection address and destination address could be combined into a single deliveryRequest parameter whose structure would be defined in a class diagram for a DeliveryRequest.

4.8.1.3 What if the service provider does not require any information from the client?

At times service providers are willing and able to deliver a service without the user/client having to provide any other information than to request the service itself. In such cases the input objects list is left empty.
Figure 4.7 specifies that training institutions can provide the current course schedule without requiring any further information from the client requesting the course schedule.

### 4.8.2 Return types

The end of the service signature may optionally specify a return type. For example, Figure 4.5 specifies that the `provideQuote` service returns a quote and that the `processClaim` service provides a `SettlementReport`. The information returned with the quote and the settlement report would then be specified in separate class diagrams for these objects.

#### 4.8.2.1 What if multiple deliverables?

UML allows for only a single return value. The return value can, however, be a composite object which has many components. For example, if the result of a purchase is a package of items plus the invoice, then these can be packaged within a higher level object, say a `Package`, which contains the package of items and the invoice.

Alternatively one can use output parameters to provide multiple deliverables (see Figure 4.8).

#### 4.8.2.2 What if nothing is returned?

At time, service providers perform some activity upon service request without returning anything to the client. For example, a radio station may provide a service to the weather bureau to announce a weather alert (e.g. a storm warning) without returning anything to the weather bureau. The radio station simply announces the weather alert. Such services are simply specified without return value.
4.8.3 Input, output and input/output parameters

So far we have viewed all parameters as input parameters, i.e. the parameter objects were provided to the service provider upon request, but were not returned.

However, at times clients provide an object which is modified and returned by the service provider. For example, you may request an audio equipment servicer to repair your audio equipment supplying the service provider the audio equipment.

Figure 4.10: Parameters may be specified as in, out and inout.

In this case the full service signature would be

\[
\text{repair}(\text{in problemReport:ProblemReport, inout item:AudioEquipment}) : \text{Invoice}
\]

specifying that the item to be repaired is provided upon service request and returned when the service has been provided.

**Note**
The UML tool will support the specification of a parameter as an in, out or inout parameter. The information would be maintained in the model and need not be shown on the class diagrams.

4.8.4 Multiple services with same service name (overloading)

UML supports the concept of following different business processes depending on the types of parameters provided when the service is requested. For example, a bank could specify different business processes for processing different types of loan applications.
4.9 Mapping UML service declarations onto Java

Service declarations within a UML class diagram map onto Java method header declarations. The method bodies are not provided by the information contained in class and object diagrams. It is the dynamic model (sequence, activity, state, timing, ... diagrams) which fill in the method bodies.

For example, the UML diagram for the account shown in Figure 4.12 would map onto the following Java code:

```java
public class Account {
    public TransactionConfirmation credit(Money amount) {
        ...
    }
}
```

4.9.1 Output and input/output parameters

Java natively only supports input parameters. Output and input/output parameters are implemented using a holder class which wraps the output or input/output parameter.

Consider, for example, a runningAverage service which iteratively updates the running average across a sequence of readings, every time receiving the last value of the running average, the new reading and the total number of readings as parameters. In addition to this it also calculates the deviation of the last reading from the average. Here

- the number of readings and the last reading are in (input) parameters,
- the running average is an inout (input/output) parameter,
- and the deviation from the average is an out (output) parameter.

The Java mapping would be something like the following:
public class RunningAverageUtility
{
    public void updateRunningAverage (DoubleHolder runningAverage,
        double reading,
        int numReadings,
        DoubleHolder deviationFromAverage)
    {
        double newRunningAvg
            = (runningAverage*(numReadings-1)+reading)/numReadings;
        runningAverage.setValue(newRunningAvg);
        deviationFromAverage.setValue (reading.getValue()-runningAverage.getValue());
    }
}

where the holder class would be something like the following:

class DoubleHolder
{
    public DoubleHolder(Double value) {this.value;}  
    public double getValue() {return value;}
    public void setValue(double newValue) {value = newValue;}
    private double value;
}

Note
In many cases input/output parameters are simply handled by the client providing the service provider a reference to the object which is modified within a method. For example, in

public class SubscriptionManager
{
    public TransactionConfirmation raiseSubscriptionFee (Client client, Account account)
    {
        ...
        account.debit(client.getTotalSubscriptionFee(period));
        ...
    }
}

the raiseSubscriptionFee service obtains a handle to the client account which is debited.

4.10 Access levels (visibility)

UML provides some rudimentary support for constraining access. This is done by assigning an access level to a member (e.g. attribute or a service). It specifies whether the member can be accessed from outside the container (e.g. object) which owns it, and if so, to what extent.

UML defines 4 access levels:
• **public (+)** The member can be accessed from any object which has a handle/message path to the container. For example, a client of an organization can use the public services of the organization. Similarly any system object which has a message path to another system object can access that objects public methods and attributes.

• **package (-)** The member can be accessed from any object which is in the same package as the container.

• **protected (#)** The member can be accessed from within the container itself or from within instances of specializations of the container. For example, protected members of a class can be accessed from within sub-class instances.

• **private (-)** The member can only be accessed from within the container itself. For example, the private attributes of an instance of a class can only be accessed from within any of the business or system processes of that same instance.

### 4.11 Mapping UML access levels onto Java

Java supports very similar access levels to those provided by UML. It supports both, the **public** and **package** access levels with package access level being the default for Java. There is thus no keyword for package scope. If no access level is requested, package scope is assumed.

The **protected** access level in Java provides not only access to sub-class instances, but also to any object from any class which is within the same package -- i.e. **protected** includes **package scope**.

The **private** access level in Java does not mean private to the instance, but private to the class. Thus, from within one instance of, for example, the account class, you can access directly the private balance of another instance of an account.

Generally one should use only the **private** access level for class fields, i.e. fields should preferably be encapsulated to within a class as they represent an implementation detail of the class). Services, on the other hand, are commonly defined with public and protected scope. Package scope is used seldom for services.

At times, classes are meant to be an implementation detail of a framework. Such classes should only be accessible from other classes within that same framework. This is achieved by defining a package for the framework and declaring such a class with **package** scope.

As an example of the Java syntax for access level specification, consider the following code extract:

```java
package java.util;

public class LinkedList
{
    public Iterator iterator() {...}

    protected void optimizeStorage() {...}

    private Node head;
}

class Node
{
    ...
}
```

Here the `Node` class has been defined with package scope. It is reusable within the package for the collections framework, but not visible by users of that framework.

### 4.12 The camel naming convention

Even though the Camel convention is not officially part of the UML specification, it is a very widely used naming convention across object oriented approaches including UML, XML and object oriented programming languages.

The convention is very simple:
• Class (and interface) names start with capital letter.
• Everything else, including services and object names, starts with a lower case letter.
• Word boundaries are capitalized in either case.

![Account Class Diagram](image)

For example, in Figure 4.13, the Camel naming convention is applied:

• Class names like `Account`, `Value` and `TransactionConfirmation` start with capital letter, with word boundaries capitalized as is done using a capital C for `TransactionConfirmation`.

• Object names like `myAccount`, the parameters `amount`, `balance` as well as the class attributes or properties like `accountNumber` all start with lower case letter with word boundaries still capitalized.

• Service names like `credit` and `generateStatement` start with lower case letter.

### 4.13 Interfaces

#### 4.13.1 Introduction

Interfaces provide a mechanism for decoupling clients from specific service providers. The services the client requires for his business processes are encapsulated within an interface. The aim is to be able to use any service provider realising the client’s service requirements (any service provider implementing the interface) without making any changes to the client’s business process.

An interface is usually used to encapsulates the client’s requirements around the services from a particular domain of responsibility. It can be seen as the core of a services contract, listing the services required from some responsibility domain. The client dependencies are thus abstracted from a dependency on some particular service provider to a dependency on certain services being provided. As such, interfaces provide a mechanism for abstracting from any particular service provider.

If a concrete service provider claims to implement or realise and interface, the service provider is required to provide all the services specified in the interface.

UML enables one to assign a business or system process to a service of a class. This is, however, not possible for interfaces. The interface specifies the service requirements and ultimately the services contract. The business processes used to realise these services contracts is left to the individual service providers.
4.13.2 UML notation for interfaces

The UML notation for an interface is the same as a class diagram, except that one assigns an `<<interface>>` stereotype to the class diagram. Alternatively, and more commonly the interface can be identified through the interface stereotype icon (an empty circle) shown in the name compartment of the class diagram.

Figure 4.14: An interface specifying the services required from an assessor.

Figure 4.14 uses an UML interface specification to specify that the services required from an assessor are that of assessing the value of an item and that of assessing the value of a claim. In the one case the assessment should return a `ClaimValuation` while in the case of valuing an item, an `ItemValuationReport` is returned.

4.13.3 Realizing/implementing interfaces

A class provides a mechanism through which one can assign the concrete business processes which should be followed when realizing a service. We can have thus multiple classes which use different business processes to realize the same services, i.e. multiple classes can provide their own realization or implementation of an interface.

Figure 4.15: Estate agents and the property sales register both realize the services required from a property valuator.

Figure 4.15 specifies that both, `EstateAgent` and `PropertySalesRegister` realize the services required from a `PropertyValuator`. For each of these classes one would specify the business process used to realize the property valuation service by assigning the appropriate activity and sequence diagrams to the service.

4.13.4 Decoupling from service providers

One of the benefits of using interfaces is that clients are decoupled from actual service providers. The clients business process is hence no longer dependent on any particular type of service provider which uses a specific business process to realize a service.
Instead one can plug in different service providers which may realize the required service(s) through different business processes. To show this in a UML diagram, the uses relationship is not linked to a particular class, but to the interface.

Figure 4.16: Home loans is decoupled from any concrete realization of a property valuator.

In Figure 4.16 home loans is not locked into any particular type of property valuator, i.e. the business process for processing a home loan application need not be modified when changing from one property valuator to another. Each property valuator implementation (class) can use a different business process when realizing the property valuation service.

Figure 4.17: The property valuator interface collapsed into its stereotype icon.

In order to simplify the diagrams, one often one shows the dependency on an interface without showing the details of the services required from service providers implementing the interface. In such cases the interface diagram is collapsed into its stereotype icon for an interface as is done in Figure 4.17.

Note
Note also that the realization relationship is now shown via the so-called lollipop notation, i.e. as a solid line between the interface and the class realizing the interface.

4.13.4.1 Required and provided interfaces

UML supports a second notation highlighting an interface required by a client and the interface provided by a service provider. The notation uses the dish to show the required side of an interface and the lollipop notation to show the provided side of an interface.
One can show a required interface without specifying any object which provides it as illustrated in Figure 4.19. This notation is particularly useful when designing system or business processes without being yet concerned about the implementation details. The latter would be addressed during the implementation phase where the design is mapped onto a concrete realization of the business or system process.

**4.13.5 Extending interfaces**

UML supports the definition of extended interfaces. The extended interface requires that service providers implementing it must realize not only the services defined in the extended interface, but also those defined in the interface it extends. In UML one uses the specialization relationship to specify that one interface extends another.
Some service providers may choose to implement only the base interface, offering only the core services. Other service providers may choose to realize the extended interface. These service providers must realize all the services specified in the extended interface as well as those specified in the base interface.

4.13.5.1 Join interfaces

We can join two multiple interfaces into a single join interface. Service providers implementing the join interface must provide all the services specified in any of the interfaces which are extended via the join interface.

4.13.6 Using a UML interface to specify a services contract

One can view an interface as the core of a services contract. To complete the services contract one needs to specify the pre- and post-conditions for each service as well as the quality requirements (either at a per-service level or for the service provider as a whole) and the requirements for the exchanged value objects. These can all be packaged within a single contract package.

In order not to violate substitutability, a service provider who claims to realize the contract must

- provide realizations for all services specified in the contract,
- may not add further pre-conditions other than those allowed in the contract specification (pre-conditions may be reduced but not increased),
must realize at least the post-conditions as specified in the contract (post-conditions may be increased, but not reduced), and
must provide the services with at least the minimum qualities required by the contract.

**Note**
If no pre-conditions are specified, then the service needs to be provided unconditionally.

### 4.13.6.1 Example: a services contract for a caterer

As an example, consider the simple services contract for a caterer shown in Figure 4.22.

![Figure 4.22: A simple services contract for a caterer](image)

The caterer must be able to provide lunch orders and account statements. The business processes of the client are such that the lunch requests are obtained at least 3 hours before delivery is required and the client pays invoices weekly. Hence, the client is happy to accept the if the order is not placed at least 3 hours prior to delivery or if invoices older than 7 days have not been settled, the caterer may refuse the service request and it would not imply a breach of contract.

Furthermore, the client requires lunches for up to 40 people and that at least 95% of the orders placed are delivered within 15 minutes of the requested delivery time.

Any service provider which can realize the caterer contract, guaranteeing that the specified pre- and post-conditions and quality requirements are met would be pluggable. The client can switch from one to another without having to make any changes to their business or system processes.

### 4.13.6.2 Example: a message sender

The message sender service contract shown in Figure 4.23 requires two services to be provided by message senders. The sendMessage service has a number of pre and post-conditions as well as quality requirements. The second, reporting service needs to be provided unconditionally. It has no post conditions beyond that of having to provide the return value which contains the report and no quality requirements are specified.
4.13.7 Guidelines for defining interfaces

1. **Define interfaces from the client perspective** An interface should be specified from the perspective of the client, not from the perspective of any particular service provider. It should encapsulate the service requirements for the client’s business processes and enable the client to replace one service provider with another without having to make any changes to its business processes.

2. **Include only services from a single responsibility domain** A specific interface should only include narrowly related services around a single responsibility domain. A specific service provider can always implement multiple interfaces. This encourages conceptual simplicity and reusability.

3. **In a client-server relationship, always decouple via interfaces** Why would a client want to lock into a particular service provider? Specifying, in the business process a dependency on an interface instead of a particular implementation results in more flexible business processes where one service provider can easily be replaced with another without changing the business process.

4.14 Mapping UML interfaces onto Java

UML interfaces map directly onto Java interfaces. The realization relationship maps to the Java `implements` keyword.
Consider the interface with pre- and post-conditions shown in Figure 4.24. This interface maps onto the following Java interface with each pre-condition mapping onto a notifiable (non-run time) exception:

```java
public interface Account {
    public TransactionConfirmation credit(double amount);
    public TransactionConfirmation debit(double amount) throws InsufficientFundsException;
    public double getBalance();
}
```

**Note**
The notifiable (non run-time) exceptions are the contract exceptions. They are used to notify the client that a the service requested is not going to be provided as one of the pre-conditions was not met. The runtime exceptions are used to specify application errors. These should be thrown when the contractual requirements could not be met.

A class which implements the interface will have to method bodies for the business processes through which the services are realized.

```java
public class SavingsAccountImpl implements Account {
    public TransactionConfirmation credit(double amount) {
        // business process for crediting a savings account here
    }
    public TransactionConfirmation debit(double amount) throws InsufficientFundsException {
        // business process for debiting a savings account here
    }
    public double getBalance() { return balance; }
}
```
The Java language attempts to enforce that the pre-conditions are not increased by forbidding that any implementing class may not throw any other non-run-time exceptions than those specified int the interface.

**Note**
To check whether one's implementation satisfied the post-conditions, one commonly uses inserts `assert` statements just before returning from the method to the client.

Interface (or contract) extension is supported in Java. For example, if one would like to specify that any class which implements `Printable` should also be `Viewable` (i.e. that one can obtain a preview for any printable resource), then one can specify that the `Printable` interface extends the `Viewable` interface:

```java
public interface Viewable
{
    public void show();
}

public interface Printable extends Viewable
{
    public void print();
}
```

### 4.15 Interfaces versus classes versus objects

An interface contains solely a specification of the services which need to be provided when the interface is implemented. The business or system process which should be followed when a service is requested cannot be specified for an interface. It is solely the core of a services contract.

A class specifies the processes which are executed when the services are requested, but the class does not execute these processes. Nor does the class maintain the instance state.

One can, however, assign a process to the service of a class. This is done in UML by assigning the appropriate diagrams from the dynamic model like sequence and activity diagrams to the service.

### 4.16 Class specialization

#### 4.16.1 Introduction

Specialization or generalization is one of two primary mechanisms in UML enabling one to work at different levels of abstraction, the other one being the abstraction from service providers via interfaces. Specialization provides one mechanism through which one can abstract from actual (concrete) classes or concepts to more abstract concepts which usually have some commonalities.

#### 4.16.2 UML notation for specialization

In UML one specifies that instances of one class are specializations of instances of another class by drawing a solid line with a triangular head pointing from the specialized to the more general class.
Figure 4.25: CreditCardAccount is a specialization of Account

Figure 4.25 uses a UML specialization relationship to specify that every credit card account is a special type of an account.

A class can have multiple specializations. For example, Figure 4.26 specifies that both, credit card and electronic payments are special types of payments.

4.16.3 Substitutability

Specialization implies substitutability, i.e. when an particular type of object (i.e. an instance of some class) is required, one can always provide any of its specializations (an instance of any of its sub-classes).

Figure 4.27: One may substitute a credit card account for an account when requesting to raise the subscription fee for a particular subscription number.

For example Figure 4.27 specifies that account managers provide a service to raise the subscription fee for a subscription number, given an instance of an Account from which the subscription fees should be raised. Since credit card accounts are special types of accounts, one can substitute the instance of an account with an instance of a credit card account.
4.16.4 Inheritance

The specialization relationship implies inheritance, i.e. any instance of the specialized class, the sub-class, inherits all attributes and services of the more generic class, the super-class. It thus enables one to encapsulate the commonalities across different types of objects within a common super or base class. Any changes to this more abstract class would then feed through to all its specializations, i.e. to all derived or sub-classes.

Figure 4.28: CreditCardAccount inherits all attributes and services of Account.

In Figure 4.28 credit card accounts will inherit the account number and balance from the Account class as well as the credit and debit services.

Figure 4.29: For all payments we capture the payment amount and date.

Inheritance is commonly used for data or value objects. These objects are used to exchange or store some business information. For specialized data objects we may require additional information to be captured. For example, Figure 4.29 specifies that for any type of payment we need to capture the payment amount and date. For credit card payments we need to also capture the

Inheritance is commonly used for data or value objects. These objects are used to exchange or store some business information. For specialized data objects we may require additional information to be captured. For example, Figure 4.29 specifies that for any type of payment we need to capture the payment amount and date. For credit card payments we need to also capture the
credit card details and for electronic payment we need to capture, in addition to the amount and payment date, the details of the source account.

### 4.16.4.1 Inheriting and overriding business processes

The business process which should be followed when realizing a service is typically specified by attaching sequence, activity and potentially communication diagrams to the service. The sub-class inherits not only the attributes or properties of the super-class, but also the services with the business processes through which they are realized.

One may, however, define a different business process for the sub-class service. This is done by redefining the service in the sub-class and attaching a different business process specification to the sub-class service.

### 4.16.5 Polymorphism

UML supports polymorphism on message recipient as well as polymorphism on message parameters, i.e. different business or system processes may be followed depending on

- the type of service provider used within a specific execution of the process, and
- the type of objects received by a service provider upon service request.

#### 4.16.5.1 Polymorphism on message recipient

Within a business process we may require a service from some type of service provider. When executing the business process we may select to use a specialized service provider which renders the service in some specialized way. The actual way the lower level service is rendered may thus vary from one execution of the business process to the next depending on the type of service provider chosen for the individual executions of that business process.

For example, we may have a subscription manager which raises subscription fees from a provided account. Depending on the type of account provided, the debit service may be realized in a different way. In the case of some accounts there will be a service fee raised for the transaction while other accounts may earn the account holder voyager miles.

**Note**

This type of polymorphism is in the spirit of Frank Sinatra’s “I do it my way.” Different objects may provide the same service, but each may potentially do it its way.

#### 4.16.5.2 Polymorphism on message parameters

The second type of polymorphism supported in UML is polymorphism on message parameters. Here a service provider may render a service differently, depending on the type of parameter(s) provided.
4.16.6 Abstract classes

Abstract classes are classes which cannot be instantiated, i.e. classes for which one cannot create any instances/objects. They are used to

- introduce an abstract concept which one can work with,
- encapsulate commonalities across the specialized classes, and to
- lay down requirements specification for concrete subclasses.

UML tools provide a mechanism which enables the user to declare a class abstract. The class name for an abstract class is rendered in italics.
Figure 4.31: A class hierarchy of various levels of abstract chargeables with concrete leaf chargeables.

Figure 4.31 shows various chargeables. The class Chargeable as well as the sub-classes Product and Service are all abstract and cannot be directly instantiated. However, they may still specify attributes (and potentially services) which will be inherited by the sub-classes. For example, all Chargeables have a code and an identifier for an income account into which the associated income will be accumulated. The lower level abstract classes like Product and Service introduce additional attributes which are inherited by the concrete leaf classes.

4.16.6.1 Working with abstract concepts

However, even if a particular abstract class does not add any attributes or services, it still introduces a concept. For example the fact that a Consultation is a sub-class of a Service specifies that is charged as a service.

Having introduced the concept of a service enables us to specify logic around that concept. For example, we may want to do a resource allocation for scheduled services.
As another example, consider the account hierarchy shown in Figure 4.32. The account class itself has been declared abstract as clients will not be able to create instances of this generic account class -- instead they have to choose a concrete account like a *CreditCardAccount* or a *ChequeAccount*. A further level of abstract classes introduces the concepts of current, loan and investment accounts. Having introduced the concept of a current account, we can now specify that an order can be processed against any current account. The client requesting the processing of an order can thus provide, for example, a credit or debit card account, but not a loan or an investment account.

4.16.6.2 Abstract services

For an abstract class one may specify abstract services. These are services for which no business process has been specified. Hence, the business process used to realize the service is left to individual sub-classes which may each have their own business process for realizing the required service.

For example, we may have different property valuators following different business processes when valuating a property. Some property valuators may request the property valuation through a property agent, others may use a business process which takes the average of the last $n$ properties sold within an area and algorithmically adjusts for above average features.
We may want to specify that property valuators must be able to value a property, leaving the actual process which should be used for the valuation to the implementation of the individual concrete valuators. In such a case we would declare the valuation service itself as abstract as is done in Figure 4.33.

4.16.7 Multiple inheritance

At times we need to model objects which are substitutable for multiple other types of objects. In such cases one could consider using multiple inheritance.

For example, in Figure 4.34 the PersonalCard class inherits from both, IdentityCard and DriversLicense. It would thus be substitutable for both (i.e. one may provide the personal card if either an identity card or a drivers license is required) and would contain the information of an identity card as well as that of a driver’s license. If the attributes (or services) of an identity card or a driver’s license are modified, then these modifications would be inherited by the PersonalCard class.

4.16.8 Completeness constraints

If an aspect of a model should be fixed such that it cannot be refined any further, one assigns a complete constraint to that model element.
4.16.8.1 Fixing a business or system process

At times one may want to fix a business or system process such that it may not be redefined or modified for specialized classes. This requirement is specified in UML by placing a complete constraint on the respective service.

For example, we could (and I am not saying that we should) define an authenticate service for the user class which authenticates a user based on some authentication credentials. We may want to fix the authentication algorithm, that it may not be changed for specialized users, i.e., for sub-classes of the user class. If that is the case, then we would assign a complete constraint to the authenticate service.

![Diagram of User class with complete constraint on authenticate service]

Assigning a complete constraint to a service prevents the service from being overridden, i.e., no sub-class may replace the business or system process for that service.

Figure 4.35: Complete constraint on a service prevents the service from being overridden

4.16.8.2 Preventing specialization

At other times one may want to require that a instances of a class can never be modified, i.e., that they are immutable. For example, we may require that once an invoice has been issued, it may no longer be modified. In such circumstances it may be advisable to prevent subclassing as a subclass may potentially add services through which the invoice could be modified. This can be done by applying a complete constraint to the class itself.

![Diagram of Invoice and LineItem classes with complete constraint on Invoice]

The complete constraint on the class prevents the definition of a sub-class which may potentially offer services to change the state of the object.

Figure 4.36: Complete constraint on a class prevents specialization of that class

4.17 Mapping specialization relationships onto Java

UML specialization relationships map directly onto Java sub-classing. Consider, for example, the UML diagram shown in Figure 4.37.
Assuming that the `Account` class is defined elsewhere, the `CreditCardAccount` class may look something like this:

```java
public class CreditCardAccount extends Account {
    public double getVoyagerMiles() { return voyagerMiles; }
    public void debit(double amount) {
        // business process for debiting credit card accounts here
    }
    private double voyagerMiles;
}
```

Credit card accounts will inherit a balance and a default implementation for the credit service which has not been overridden in the sub-class, `CreditCardAccount`. Thus having a credit card account, we can request the credit service and the balance, even though there is no credit service nor a balance field nor a query method for the balance defined in the `CreditCardAccount` class.

```java
CreditCardAccount acc = new CreditCardAccount(...);
double lots = 1e6;
acc.credit(lots);
System.out.println(acc.getBalance());
```

### 4.17.1 Substitutability in Java

Substitutability is directly supported in Java, i.e. when an instance of some or other class is required any instance of that class or any of its sub-classes may be provided. For example, the `raiseSubscriptionFee` service, provided by the `SubscriptionManager` class

```java
public class SubscriptionManager {
    public TransactionConfirmation raiseSubscriptionFees (Client client, Account acc) {
        ...
    }
}
```

may be called providing a `CreditCardAccount` instead of an account:

```java
Account acc = new CreditCardAccount(...);
subscriptionManager.raiseSubscriptionFee(someClient, acc);
```
4.17.2 Polymorphism in Java

Java supports polymorphism on message recipient, but not polymorphism on message arguments. Thus, while

```java
public class SubscriptionManager
{
    public TransactionConfirmation raiseSubscriptionFees(Client client, Account acc)
    {
        ...
        acc.debit(fee);
        ...
    }
}
```

will call a different debit service depending on the type of account received, different methods will not be resolved upon providing different specializations as method parameter.

To see the latter, consider a loan processor for which different business processes have been defined depending on whether the client applies for a home loan or some other loan:

```java
public class LoanProcessor
{
    public LoanApplicationResult process(LoanApplication loanApplication)
    {
        // business process for general loan approval here
    }

    public LoanApplicationResult(HomeLoanApplication homeLoanApplication)
    {
        // business process to be followed for the
        // processing of home loan applications
    }
}
```

Now, assume that a loan application is submitted for processing which happens to be a home loan application:

```java
LoanApplication loanApplication = new HomeLoanApplication(...);
loanProcessor.process(loanApplication);
```

The code will unfortunately follow the business process for general loan applications and not the business process defined for home loan applications, even though the loan application provided is in fact a home loan application.

4.17.3 Abstract classes in Java

Abstract classes and services are natively supported in Java. One simply adds the `abstract` keyword to the respective declaration. The compiler and run-time environments will prevent the instantiation of an abstract class. Furthermore the developer will be forced to declare any subclass of an abstract class itself abstract if implementations have not been provided for all inherited abstract methods.

![Abstract class diagram](image)

Figure 4.38: An abstract graphics object class
For example, Figure 4.38 specifies an abstract graphics object class with an abstract `draw` service, but a concrete `move` service. The code mapping would be something like the following:

```java
public abstract class GraphicsObject {
    public void move(Point2D newPosition) {
        // implementation for concrete move method here.
    }

    public abstract void draw();
    // no implementation for an abstract method.
}
```

### 4.17.4 Multiple inheritance in Java

Multiple inheritance is not supported in Java. This is particularly problematic if instances of one class should be substitutable for multiple other classes.

Java does, however, allow for one interfaces to be an extension of multiple other interfaces. This mechanism enables one to have instances of one class be substitutable for different interfaces.

Consider the example of a personal card being substitutable for both, an identity card and a driver’s license. To achieve substitutability on both ends when mapping onto Java, we need to make both, `IdentityCard` and `DriversLicense` an interface. We might as well then define an interface for `PersonalCard` too. `PersonalCard` will be defined as an extension of both, the `IdentityCard` and the `DriversLicense` interfaces.

![Diagram](image)

Figure 4.39: Mapping multiple inheritance relationships onto Java

Now any implementation of a personal card is indeed substitutable for both, an identity card and a driver’s license. To reduce the code duplication, we still let `PersonalCard` inherit from `IdentityCard`. However, since Java does not support multiple inheritance of classes, we cannot also inherit from `DriversLicense`. The consequence of this is that there will be code duplication across the `DriversLicense` and the `PersonalCard` classes.

**Note**

The code duplication can be reduced by encapsulating the commonalities within a class which is used by both, `DriversLicense` and `PersonalCard`. However, at least the code delegating the responsibilities to that class would have to be duplicated.
4.17.5 Avoid overusing sub-classing

Sub-classing is often overused. Design experts have continuously recommended to favour both, delegation and interfaces above sub-classing. But, why is subclassing often undesirable.

In order to illustrate some of the problems encountered with sub-classing, consider the employee hierarchy in Figure 4.40.

![Class hierarchy for persons](image)

**Figure 4.40: A class hierarchy for persons**

The question on whether the class hierarchy represents "good" design or not is not necessarily easy to answer, particular if we do not have an insight into the use cases within which the classes will be used. It is not, however, unreasonable to expect that some developers may become managers or vice versa. Let us have a look at how well the design would support such a use case.

When a developer becomes a manager we will have to

- delete the developer object,
- create a manager,
- reconstruct the full state including employment history, ... (complex, error prone and a recipe for a maintenance nightmare), and
- find all the objects which had a reference to the developer and change there reference to point to the newly created manager (good luck).

Furthermore, how would we facilitate a scenario where the same person is both, a client and an employee or even a developer and a manager? We would have to create multiple instances of the person, even though there is only a single person.

Looking at the above, one may wonder whether using sub-classing was that good an idea after all. One may want to move over to an aggregation and interfaces based design which facilitates multiple roles for the same person as well as the changing of roles and responsibilities (see ).
These types of problems are not confined to some selected examples. Take another example, that of various types of accounts. Once again, the design may look good until the bank decides to change their account offerings such that the client may pay, on a single account, via cheque or via credit card and that the client may also have a loan facility on that same account. Suddenly the design which uses sub-classing becomes a lot less attractive. Instead one may want to change the design to one where a client can buy multiple services for an account.

It is difficult to anticipate requirements changes. In general class hierarchies tend to be very rigid and are not particularly maintenance friendly.
4.18 Mapping UML specialization relationships onto XML

XML schemas support inheritance, substitutability and the definition of abstract classes. Consider, for example, the class hierarchy shows in Figure 4.44.

![Figure 4.44: An accounts collection which may contain different concrete accounts](image)

The UML diagram specifies that instances of Accounts will contain zero or more instances of Account. Account itself, however, has been declared abstract. Hence every instance contained within the Accounts collection must be an instance of one of the concrete subclasses of account. The accounts collection is a polymorphic collection and we may substitute any specialized account for the account entries.

Mapping the UML diagram onto an XML schema yields

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.solms.co.za/finance/accounts"
xmlns="http://www.solms.co.za/finance/accounts"
elementFormDefault="qualified" attributeFormDefault="unqualified">
  <xs:complexType name="Accounts">
    <xs:sequence>
      <xs:element name="account" type="Account" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="Account" abstract="true">
    <xs:sequence>
      <xs:element name="accountNumber" type="xs:string"/>
      <xs:element name="balance" type="xs:decimal"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="CreditCardAccount">
    <xs:complexContent>
      <xs:extension base="Account">
        <xs:sequence>
          <xs:element name="creditLimit" type="xs:decimal"/>
          <xs:element name="creditCard" type="CreditCard" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:schema>
```
An example XML document complying to the schema specification is shown below:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<accounts xmlns="http://www.solms.co.za/finance/accounts"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.solms.co.za/finance/accounts ./accounts.xsd">
    <account xsi:type="SavingsAccount">
        <accountNumber>S111</accountNumber>
        <balance>100</balance>
        <interestRate>4.3</interestRate>
    </account>
    <account xsi:type="CreditCardAccount">
        <accountNumber>CC121</accountNumber>
        <balance>345</balance>
        <creditLimit>1000</creditLimit>
        <creditCard>
            <cardNumber>4444 5555 3333 2221</cardNumber>
            <clearingHouse>AllClear</clearingHouse>
            <holderName>John Smith</holderName>
            <expiryDate>2010-01-02</expiryDate>
        </creditCard>
        <creditCard>
            <cardNumber>4444 5555 3333 2222</cardNumber>
            <clearingHouse>AllClear</clearingHouse>
            <holderName>Tandi Smith</holderName>
            <expiryDate>2010-01-02</expiryDate>
        </creditCard>
    </account>
    <account xsi:type="SavingsAccount">
        <accountNumber>S112</accountNumber>
        <balance>15000</balance>
        <interestRate>5.1</interestRate>
    </account>
</accounts>
```
4.18.1 The problem of multiple inheritance in XML schemas

XML schemas do not support multiple inheritance. This is problematic as XML schemas do not provide any alternative mechanism for obtaining substitutability for instances of one class for instances of multiple other classes.

Thus, assume, for example, that we require that a personal card is substitutable for both, an identity card and a drivers license card. The XML schema specification does not provide a clean mechanism through which we can specify such a data structure.

4.19 Association

4.19.1 Introduction

Associations are largely used for two purposes:

- to specify service request channels which clients maintain for certain service providers, enabling the client to, at any stage, request further services from that service provider, and
- to specify navigation paths within data objects or entities which enables one to resolve one piece of information or one entity from another.

4.19.2 UML notation for association

UML uses the same notation for client server type associations and for specifying navigation paths between domain or data objects. In either case a uni-directional association is shown as a solid line with an open arrow pointing in the direction of navigatability, i.e. from the client to the server (in the direction in which service request messages flow) or from the domain object from which you can resolve another to the resolved domain object.

Figure 4.45: The basic notation for a unary association.
4.19.2.1 Role names and cardinalities

At times the role played by an object in the context of a client server relationship may not be obvious. Furthermore, at times the same object can play multiple roles in different contexts. In such cases it may be useful to specify the role name for either the client or the server. This is done by putting a label at the appropriate end of the association.

![Figure 4.46: Role names, association labels and cardinalities]

Clients could use multiple instances of the service provider and service providers could provide services to multiple instances of clients. Cardinality constraint can be used to specify acceptable ranges of multiplicities. In our example, the source funds from multiple banks who play the role of lenders, but has a current account with only one bank which plays the account provider role. The company plays the borrower and account holder roles in the context of these relationships.

Association labels can be used to further specify the nature of an association. In our example we use the `SourcesFundsFrom` and `Provides`. Note that one can optionally specify a direction arrow. The direction arrow specifies in which direction the association label should be read. In cases where there is no direction arrow, the association is read in the direction of the association itself.

4.19.3 Navigatibility

An association relationship can be used to purely provide accessibility of related information. In other words, an association between two entities enables one to locate the one entity from the other.

![Figure 4.47: An association facilitating the location of associated information.]

For example, we may require that for an insurance policy we are able to determine the beneficiary. We would model this in UML as an association from the policy to the beneficiary, providing the ability to navigate from an insurance policy to the associated beneficiary.

4.19.4 Association for client-server relationships

If one object, the client, regularly requires to use the services of another object, the server, then the client needs to maintain information about the message path used to deliver these service requests. This would be modeled in UML as an association. Both, the client and the server could be any of the following:

- an organization,
- a business unit within an organization,
4.19.4.1 Abstracting from service providers

In the context of a client-server relationship it is virtually always desirable for the client to decouple from any particular service provider. The decoupling is realized by specifying service agreements (contracts) which encapsulate the client’s requirements for a particular role player.

In such a case the association would not point to a class, but to the interface encapsulating the services required by the client around some responsibility domain. Such an interface would normally be expanded into a services contract or SLA.

For example, an organization may, as part of the monthly interest calculations on loan accounts identify loan accounts which are in arrears. The organization may require that the owners of these accounts are contacted in order to notify them of the situation. This responsibility could be assigned to an internal call centre. Alternatively the organization could outsource this responsibility to an external service provider. A third option would be to assign the responsibility to a system which dispatched the appropriate messages. The rest of the business process need not be affected by the choice of communications services provider.

One could introduce a client communications interface encapsulating the services required from such a service provider.

4.19.4.2 Example message paths

Clients use message paths client to deliver service requests to servers. Examples of commonly used message paths include:

- the service provider’s telephone or e-mail number,
- a courier/transport service used to deliver requests with physical input objects to the service provider (e.g. for a training institution to deliver exams to an external examiner),
- the URL for a web service,
- a message queue used by one system to deliver messages to another, or
- a reference or pointer to a system component.
4.19.4.2.1 When are the types of message paths to be used determined?

If one follows an MDA based approach, then one first does a implementation neutral business process design resulting in the technology and architecture neutral Platform Independent Model (PIM). When implementing the business process the PIM is mapped onto an implementation model, the Platform Specific Model (PSM) of the MDA. This mapping is usually done by the technical team.

The implementation neutral PIM will typically not specify the actual message paths used by the objects which require the service (the ones which play the client role) to deliver service requests to the objects which provide the service (the ones which play the server role). The decision on which message paths to use is usually only made when implementing the business process, i.e. when the technical team maps PIM onto the PSM.

4.19.5 Peer to peer relationships

A peer-to-peer relationship is a bi-directional client server relationship where each of the two objects maintains a message path to the other in order to be able to request services from the other.

![Diagram](Diagram.png)

Figure 4.49: A peer-to-peer relationship can be documented using bi-directional associations.

Often the two parties use each other for different purposes. Then one should split the bidirectional association into two unidirectional associations and define separate role names for the two role players.

4.19.5.1 Decoupling in peer-to-peer relationships

Irrespective of whether the two parties use one another for the same or for different services, one would usually want to decouple the client side from the concrete service provider by

- splitting the binary association into two uni-directional associations, and
- inserting for each role player an interface which represents the core of the contract (SLA).

For example, a jazz club may source catering services from a neighbouring restaurant while the restaurant may use the jazz club to provide it with entertainment services. This binary association with strong coupling between the two classes can be decoupled by decomposing the binary association into two unary associations and inserting the relevant services contracts (see Figure 4.50).
Should the two parties use each other for the same services, they would play the same role for each other. In such a case one would insert a single interface (contract) for that role with both parties realizing that interface and making use of another party which realizes that same interface, inserting services contract.

### 4.19.6 Association classes

An association represents a message path. Message paths are themselves objects and hence instances of classes. So far we have not exposed, in our UML diagrams, any properties or services which are offered by the object which provides the message path itself. UML enables one to do this via association classes.

#### 4.19.6.1 Uni-directional association classes

Uni-directional association classes represent a message path for a unary association. They provide a message path in one direction only.

A credit card can be seen as an example of an association class which provides users a message path to the associated credit card account.

The credit card has its own attributes like the card number and expiration dates and may even potentially render services. For example, the card could have encoded the pin and could validate the pin through logic encoded on the card itself.
Note
Aggregation and composition relationships are special type of associations and hence association classes can be used for aggregation, composition and association relationships.

4.19.6.2 Bi-directional association classes

Bi-directional association classes can be used as an implementation of a binary association. They provide a message path in both directions.

In some cases one may even use an association class for bi-directional message paths. In such examples either of the two parties communicate with each other through an instance of the associations class.

For example, in the context of the sale of a property, the seller and buyer could communicate with one another through a property agent. In this case the property agent provides the message path for either of the two directions.

![Diagram](image)

Figure 4.52: A property agent as an bi-directional association class.

4.19.7 N-ary associations

N-ary associations provide a convenient notation to hide a mess. A harmlessly looking diamond with aesthetically pleasing straight lines to a number of classes specifies binary associations between all these classes introducing a strong coupling between all of them.

![Diagram](image)

Figure 4.53: An n-ary association between the objects relevant for the sale of a property

On usually identifies an object as a mediator between them. There will be (in the worst case) binary associations between the mediator and each of the components. From any object within this pattern you can get to any other object via the mediator. At times, one finds a natural mediator amongst the objects participating in the n-ary association. At other times one introduces a new, higher level, class which links the participant of the n-ary association.
4.20 Mapping UML association relationships onto Java

An association is implemented in Java as either a local or a remote (typically RMI) reference. Thus, if each retail outlet has an association to a region, then we simply have

```java
public class RetailOutlet
{
    public Region getRegion() {return region;}
    public void setRegion(Region newRegion)
    {
        this.region = newRegion;
    }

    private Region region;
}
```

4.21 Mapping UML association relationships onto XML

UML associations would typically be mapped onto

- *key references* in the case of intra-document references (i.e. if the referenced object is contained in the same XML document as the object which has the association),
- or onto *xlinks* in the case of external links (i.e. where the referenced object is eternal to the document hosting the object which has the association).

4.21.1 Intra-document associations

Intra-document associations are preferable specified using keys and key references. A key is a special type of uniqueness constraint which is referencable. It may be a composite key which is assembled across multiple elements and/or attributes.

For a key we need to specify

1. the scope across which the key is unique,
2. the object which is uniquely identified by the key, and
3. the fields within the object from which a unique identifier for the object is assembled.

The scope is specified by putting the key definition within that element within which the key needs to be unique. The selector element of a key definition points to the object which is uniquely identified by the key. Finally, the field element(s) refer(s) to the element(s) and/or attribute(s) which make up the unique identifier for the key.

Similarly, for a keyRef element (which represents the actual association or pointer) we need to specify the object which hosts the association or pointer and the fields which make up the data for the association (pointer) variable.

4.21.1.1 Example: Retail outlets associated with regions

Consider, for example, the UML diagram in Figure 4.54.
To implement the association in XML schema, we need to specify a key definition for the Region class and a keyRef definition for the association variable of the RetailOutlet class.

```xml
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://www.solms.co.za/retail/outlets"
           xmlns:r="http://www.solms.co.za/retail/outlets"
           elementFormDefault="qualified" attributeFormDefault="unqualified">

  <xs:element name="retailOutlets" type="r:RetailOutlets">
    <xs:key name="RegionKey">
      <xs:selector xpath="./r:region"/>
      <xs:field xpath="@id"/>
    </xs:key>

    <xs:keyref refer="r:RegionKey" name="RegionRef">
      <xs:selector xpath="./r:retailOutlet"/>
      <xs:field xpath="r:region"/>
    </xs:keyref>
  </xs:element>

  <xs:complexType name="RetailOutlets">
    <xs:sequence>
      <xs:element name="retailOutlet" type="r:RetailOutlet" maxOccurs="unbounded"/>
      <xs:element name="region" type="r:Region" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="RetailOutlet">
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="address" type="xs:string"/>
      <xs:element name="region" type="xs:string"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:string" use="required"/>
  </xs:complexType>

  <xs:complexType name="Region">
    <xs:sequence>
      <xs:element name="name"/>
      <xs:element name="description" type="xs:string"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:string" use="required"/>
  </xs:complexType>

</xs:schema>
```
An XML instance document complying to this schema is shown below:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<retailOutlets xmlns="http://www.solms.co.za/retail/outlets"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.solms.co.za/retail/outlets
 ./retailOutlets.xsd">
    <retailOutlet id="out1">
        <name>Soul Flutes</name>
        <address>12 Groovy Rd, Jazz Town</address>
        <region>Deep South</region>
    </retailOutlet>
    <retailOutlet id="out2">
        <name>Chanting Drums</name>
        <address>7 Rhythm ave, Beat Town</address>
        <region>Far West</region>
    </retailOutlet>
    <retailOutlet id="out3">
        <name>His Master’s Voice</name>
        <address>13 Armstrong Ave, Freedom Town</address>
        <region>Deep South</region>
    </retailOutlet>
    <region id="Deep South">
        <name>South Border</name>
        <description>Below the mellow river</description>
    </region>
    <region id="Far West">
        <name>West Coast</name>
        <description>Behind the spooky forest</description>
    </region>
</retailOutlets>
```

**Note**

Even though the UML diagram defines the association for the class, the `keyRef` declaration is at role level. If the same class is reused in different places, we need to redefine the association for each usage of the class.

### 4.21.2 Inter-document associations

An association to an object or resource which is not contained within the XML document containing the object which has the association can be implemented using extended links, XLinks.

```xml
<out:retailOutlet id="out1"
    xlink:href="http://www.deepSouth.com"
    xlink:role="http://www.solms.co.za/region"
    xlink:title="Outlet region">
    <name>Soul Flutes</name>
    <address>12 Groovy Rd, Jazz Town</address>
    <region>Deep South</region>
</my:crossReference>
```
4.22 Composition

4.22.1 Introduction

Composition is a “strong has a” relationship where one object, the component, is intrinsically part of another object, the owner. Composition enforces encapsulation and limits the life span of the component to that of the owner. Furthermore, the owner is usually responsible for its components.

4.22.2 UML notation for composition

The UML notation for composition is a solid line with a solid diamond on the owner end of the relationship and an arrow pointing to the component.

![Figure 4.55: The notation for a composition relationship.](image)

For example, Figure 4.55 specifies that an account has a balance via composition.

![Figure 4.56:](image)

A period has two instances of a date-time, one playing the role of the start of the period, the other playing the role of the end of the period.

Instances of one class may have multiple instances of another class as components. If these components play different roles with respect to the owner, they are given different role names. This is illustrated in Figure 4.56.

4.22.2.1 Using the attributes notation for composition

Technically speaking one can use the attributes notation for association, aggregation and composition relationships. It is, however, advisable to use it exclusively for composition relationships.
4.22.3 Encapsulation

A composition relationship enforces encapsulation, i.e. the components can only be accessed by the owner.

For example, an account would have a balance via composition. The balance can only be accessed from within the account. Anyone who would like to change the balance of the account would have to do so through the services (credit or debit services) of the account. The balance itself cannot be accessed from outside the account.

4.22.4 Limited life span

In a composition relationship, the component cannot survive the owner. It would have no purpose to survive anyway as the component can only be accessed via its owner.

For example, the balance of an account does not survive the account itself. If the account no longer exists, the balance of the account will also no longer exist.

4.22.5 The owner takes responsibility for its components

In a composition relationship, the owner is responsible for its components. For example, if the processor of your computer fails, you day the computer is broken. That party which provides the warranty on your computer will be responsible for replacing the processor.

4.23 Mapping UML composition relationships onto Java

Note that Java does not natively support the concept of a composition relationship. For example, looking at the code below

```java
class Order {
    private Client client;
    private Address deliveryAddress;
    private LineItems lineItems;
}
```

we would not know whether the order has association, aggregation or composition relationships to the Client and Address classes. We need to look at the method bodies to see what type of behaviour has been implemented for these elements. Understanding the domain, we might expect that...
• each order is associated with a client,
• that an order has a delivery address via aggregation, and
• that the order has line items via composition.

When mapping composition onto a Java implementation, one should be guided by the requirements for a composition relationship, i.e. by that

• the component is only accessible via the owner,
• the component does not survive the owner, and
• the component state is part of the owner state.

One way to satisfy the first two requirements is to ensure that only the owner object ever has a reference to the component. If only the owner has a reference to the component, the component will become unreferenced and garbage collected.

The third requirement states that composition is a special form of an aggregation relationship. If we observe state observability we need to also implement state change notification. In aggregation the aggregate object had to register as state change listener with its components and, upon receiving a state change event, the aggregate object had to assess which aspects of its state is affected by the state change of the component and publish the relevant state change notification event to its state listeners. In the case of composition the component is only accessible via the owner and hence its state can only be modified by the owner (or one of the components of the owner). The owner need not necessarily register as state change listener with the component -- it might directly issue a state change event upon having changed the state of its component.

As an example, consider an account which has an instance of `Money` (amount in a currency) for the balance. The relationship must be composition as the balance should only be accessible by the account itself. If you want to change the balance you will need to credit or debit the account. Furthermore, if the account is deleted, the balance should be deleted too.

One way to implement this is to ensure that the account never returns a handle to the balance.

```java
public class Account implements Cloneable
{
    public Account(Money openingBalance)
    {
        this.credit(openingBalance);
    }

    public void credit(Money amount) throws IncorrectCurrency
    {
        if (!amount.getCurrency().equals(balance.getCurrency()))
            throw new IncorrectCurrency();

        balance.setAmount(balance.getAmount() + amount.getAmount());
        propertyChangeSupport.firePropertyChange("balance",null,null);
    }
}
```
public void debit(Money amount) throws IncorrectCurrency
{
    if (!amount.getCurrency().equals(balance.getCurrency()))
        throw new IncorrectCurrency();

    balance.setAmount(balance.getAmount() - amount.getAmount());
    propertyChangeSupport.firePropertyChange("balance",null,null);
}

public Account clone()
{
    Account copy = null;
    try
    {
        copy = (Account)super.clone();
        copy.balance = this.balance.clone();
        // Need to clone components
    }
    catch (CloneNotSupportedException e)
    {
        /* never thrown, merely removing precondition */
    }
    return copy;
}

public Money getBalance()
{
    return balance.clone();
}
...

private Money balance;

public static class IncorrectCurrency extends Exception {}
}

with the Money class simply defined as:

public class Money implements Cloneable
{
    public Money(double amount, String currency)
    {
        setAmount(amount);
        setCurrency(currency);
    }

    public double getAmount()
    {
        return amount;
    }

    public void setAmount(double newAmount)
    {
        this.amount = amount;
        // can do this because primitive
    }

    public String getCurrency()
```java
{
    return currency;
}

public vois setCurrency(String newCurrency)
{
    this.currency = currency;
    // can do this because immutable
}

public Money clone()
{
    Money copy = null;
    try
    {
        copy = (Money)super.clone();
    }
    catch (CloneNotSupportedException e)
    {
        /* never thrown, merely removing precondition */
    }
    return copy;
}

private double amount;
private String currency;
```

### 4.24 Mapping UML composition relationships onto XML

Composition maps onto XML sub-elements. Thus, if an Account has a balance which is an instance of Money by composition, then the XML schema would be something like this:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:complexType name="Account">
        <xs:sequence>
            <xs:element name="balance" type="Money"/>
        </xs:sequence>
        <xs:attribute name="accountNo" type="xs:string" use="required"/>
    </xs:complexType>

    <xs:complexType name="Money">
        <xs:sequence>
            <xs:element name="amount" type="xs:decimal"/>
            <xs:element name="currency" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>

    <xs:element name="account" type="Account"/>
</xs:schema>
```

An XML instance document complying to this schema would look something like this:

```xml
<account accountNo="A123">
    <balance>
        <amount>100</amount>
    </balance>
</account>
```
4.25 Aggregation

4.25.1 Introduction

Aggregation is a “weak has a” relationship where an aggregate object has a component without enforcing full ownership, i.e. without enforcing encapsulation and limited life span on the component. The component may hence

• be accessed directly without going through the aggregate object,
• survive the aggregate object, and
• may be also part of another object.

Furthermore, the aggregate object does not usually take responsibility of the component.

The component is, however, still part of the state of the aggregate object. If the state of the component changes, the state of the aggregate object changes.

For example, a team has multiple people via aggregation. If a the state of a team member changes, i.e. if a member of the team falls sick, then the state of the team changes. Furthermore, if the team is dissolved, then the people are likely to continue living. Also, the same person could be a member of multiple teams.

4.25.2 UML notation for aggregation

An aggregation relationship is specified in UML by drawing a hollow diamond on the aggregate object side with a solid line and an open arrow pointing to the component. For example, Figure 4.59 shows that a team has one or more persons via aggregation. These persons are the members of the team.

![Figure 4.59: A team has multiple persons via aggregation.](image)

4.25.3 Difference between aggregation and composition

To understand the difference between aggregation and composition, consider the example of a drawing composed of multiple graphics objects, each of which has a style. We could specify the graphics object has a style via composition or via aggregation.
Consider the first scenario where a graphics object has a style via composition. In this case the style can be accessed only via its owner, for example by right-clicking on a graphics object and editing its style. Changing the style of a particular graphics object would not change the style of any other graphics object as a component can have only a single owner. If the graphics object is deleted, so will its style be -- the style cannot survive the graphics object.

Now consider the second scenario where the graphics object has a style via aggregation. In this case the style could be accessed and edited directly, perhaps by selecting it from a styles menu or palette. Changing the style may now change multiple graphics objects -- all those which have that style (in aggregation a component could be part of multiple aggregate objects). Furthermore, the style may survive the deletion of any graphics object which has that style.

It should be clear that looking only at the diagrams we cannot say which is correct. That would depend on what the client wants. However, changing from aggregation to composition results in a very different behaviour of our drawing application.

### 4.26 Mapping UML aggregation relationships onto Java

By default, aggregation is implemented in the same way as association is. The aggregate object simply maintains a message path to the component. In addition to this, the aggregate object will typically have certain properties whose value is affected by state changes in the components.

Assume, for example, that client has multiple accounts via aggregation and that the client has an exposure property which represents the total exposure the organization has with respect to the client. Assume this is calculated as the sum total of the balances of the client’s accounts.

```
public class Client
{
    ...

    public double getExposure()
    {
        double exposure = 0;
        for (Account acc:accounts)
            exposure += acc.getBalance();
        return exposure;
    }
}
```
public void addAccount(Account acc)  
{  
    accounts.add(acc);  
} 

private Set<Account> accounts = new HashSet<Account>();

Note  
The client state depends on the state of the components. If the balance of one of the accounts changes the exposure of the client changes.

You may, however, want to support state change notification. This is usually done within the Java Beans framework. To do this, the client will have to register as state change listener with all its accounts. Upon receiving a balance change event from one of its accounts, the client will have to issue an exposure change event to its state change listeners:

public class Client  
{  
    ...  

    public double getExposure()  
    {  
        return exposure;  
    }  

    protected double calcExposure()  
    {  
        double exposure = 0;  
        for (Account acc:accounts)  
            exposure += acc.getBalance();  
        return exposure;  
    }  

    public void addAccount(Account acc)  
    {  
        accounts.add(acc);  
        acc.addPropertyChangeListener(this, "balance");  
    }  

    public void propertyChanged(PropertyChangeEvent event)  
    {  
        double oldExposure = exposure;  
        exposure = calcExposure();  

        if (exposure != oldExposure)  
            propertyChangeSupport.firePropertyChange("exposure", oldExposure, exposure);  
    }  

    private Set<Account> accounts = new HashSet<Account>();  
    private PropertyChangeSupport propertyChangeSupport;  
    private double exposure;  
}
4.27 Mapping UML aggregation relationships onto XML

Aggregation is implemented in XML just like association.

**Note**

One should not include any derived attributes in an XML schema definition as this would be non-validatable by an XML parser and introduces the risk of data integrity.

---

![Diagram](image.png)

**Figure 4.62:** Stock items have a product via aggregation

For example, we may have a number of retail outlets which carry stock. Each stock item represents stock of a particular quantity of a particular product. We may want to use aggregation between stock item and product as a change in the product price should affect the stock items across all stores.

This design would map onto the following XML schema:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.solms.co.za/retail/stockReport"
  xmlns:sr="http://www.solms.co.za/retail/stockReport"
  elementFormDefault="qualified" attributeFormDefault="unqualified">
  <xs:element name="stockReport" type="sr:StockReport">
    <xs:keyref refer="sr:ProductKey" name="ProductRef">
      <xs:field xpath="sr:product"/>
    </xs:keyref>
    <xs:key name="ProductKey">
      <xs:selector xpath="./sr:products/sr:product"/>
      <xs:field xpath="@code"/>
    </xs:key>
    <xs:complexType name="RetailOutlet">
      <xs:sequence>
        <xs:element name="retailOutlet" type="sr:RetailOutlet" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:complexType>
</xs:schema>
```
An SML instance document complying to this schema is shown below:

```xml
<retailOutlet id="A!">
  <name>Albania Carpets</name>
  <address>12 Albany Rd, Tirana</address>
  <stockItems>
    <stockItem>
      <product>C1</product>
      <quantity>22</quantity>
    </stockItem>
    <stockItem>
      <product>C2</product>
      <quantity>6</quantity>
    </stockItem>
  </stockItems>
</retailOutlet>
```

**Schema Definition:**

```
<xs:complexType name="RetailOutlet">
  <xs:sequence>
    <xs:element name="name" type="xs:string"/>
    <xs:element name="address" type="xs:string"/>
    <xs:element name="stockItems" type="sr:StockItems"/>
  </xs:sequence>
  <xs:attribute name="id" type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="StockItems">
  <xs:sequence>
    <xs:element name="stockItem" type="sr:StockItem" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="StockItem">
  <xs:sequence>
    <xs:element name="product" type="xs:string"/>
    <xs:element name="quantity" type="xs:nonNegativeInteger"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="Products">
  <xs:sequence>
    <xs:element name="product" type="sr:Product" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="Product">
  <xs:sequence>
    <xs:element name="name" type="xs:string"/>
    <xs:element name="price" type="xs:decimal"/>
  </xs:sequence>
  <xs:attribute name="code" type="xs:string" use="required"/>
</xs:complexType>
```

**An SML instance document:**

```xml
<?xml version="1.0" encoding="UTF-8"?>
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.solms.co.za/retail/stockReport
  ./stockReport.xsd">
  <retailOutlets>
    <retailOutlet id="A!">
      <name>Albania Carpets</name>
      <address>12 Albany Rd, Tirana</address>
      <stockItems>
        <stockItem>
          <product>C1</product>
          <quantity>22</quantity>
        </stockItem>
        <stockItem>
          <product>C2</product>
          <quantity>6</quantity>
        </stockItem>
      </stockItems>
    </retailOutlet>
  </retailOutlets>
</stockReport>
```
4.28 Dependencies

A dependency represents a very weak relationship which specifies that instances of one class are dependent on instances of another. Other relationships like association, aggregation, composition and specialization are all stronger forms of a dependency which enforce additional requirements on the relationship.

A dependency between two classes is shown in a UML diagram by drawing a dashed line with an open arrow pointing from the class which has the dependency to the class whose instances the other class depends on.
4.29 Dependency relationships in Java

Dependency relationships are not explicitly specified in Java. They would be implicit by, for example, defining in the class specification that an instance of a particular class is provided with a service request, or by specifying in the dynamic model that at some stage in the business process, an instance of one class is created from within an instance of the class hosting the business process.

4.30 Meta classes

A class diagram describes the instances of the class. It specifies the attributes, services and relationships which each instance of the class will have. As such one can see a class diagram as a meta-object, i.e. it describes the object.
A meta-class then describes the class itself. Any attributes or services specified in the meta-class are attributes and services of the class, not of instances (objects) of the class.

A common usage of meta-classes is to specify the services which create instances of the class as class members. They cannot be instance members as one would not be able to create the first instance.

A meta-class is specified in UML by assigning a metaclass stereotype to the class. Figure 4.66 shows the meta class with the class members (the factory method, create, an attribute which keeps track of the number of instances of the class and a query service for this attribute). The class diagram for account specifies the instance members including the balance, the account number and the credit service.

4.31 Mapping UML meta-classes onto Java

UML meta-classes map onto static members in Java. Thus if we have a metaclass for the Account class which specifies the numInstances, getNumInstances and a create method as meta-class members, then each of these would map onto a static member.

```java
public class Account {
    /* static */ Account(String accNo) {
        // now in an instance service initializing the instance
        this.accNo = accNo;
        ++Account.numInstances;
    }

    // an instance service
    public TransactionConfirmation credit(double amount) {...}

    // a class service
    public static int getNumInstances() {return numInstances;}

    // a class attribute
```
private static int numInstances = 0;

// instance attributes
private double balance;
private String accNo;

Factory services like create are normally mapped onto constructors. A constructor in Java is a combination of a class service creating the instance and an initialization section which initializes the instance. While the service as a whole is a class service (and should, in principle, have been declared static), the body of the constructor is for an instance service.

Class services should preferably be requested by sending the service request message to the class itself:

```java
int numInstances = Account.getNumInstances();
```

### 4.32 Containment

Containment is a packaging relationship. It is used in class diagrams to specify inner classes. An inner class is a class which

- can only exist within an instance of its outer class and
- has access to the private members of the outer class.

When using containment, one actively discourages reuse -- i.e. the class is not meant for re-use.

The UML notation for an inner class is a circle with a cross at the outer class end with a solid line leading to the inner class.

![Diagram](Diagram)

**Figure 4.67: Voyager mile account as an inner class**

Consider, for example, a voyager mile account. Assume that a voyager mile account cannot exist as a stand-alone account and that every instance of a voyager mile account always exists within an instance of a credit card account. Furthermore, one may want that the voyager mile account is exclusively used for the credit card account and that it is not re-used for other purposes. If this is the case, then the voyager mile account is always packaged within a credit card account and one would use a containment relationship to specify this in a UML diagram.

### 4.33 Mapping UML containment relationships onto Java

Commonly we use containment relationships between

- packages,
- a meta-class and a class,
- classes,
- interfaces, and
- an interface and a class.
4.33.1 Containment relationship between packages

A containment relationship between two packages specifies that the one package is a sub-package of another package. This maps directly onto Java sub-packages.

![Diagram showing containment relationship between packages](image)

Figure 4.68: The accounts package is contained within the finance package

For example, Figure 4.68 specifies that the accounts package is a sub-package of the finance package. This maps onto the following package specification in Java:

```java
package finance.accounts;

// some package elements here
```

4.33.2 Static nested classes

A class contained in a meta class is mapped onto a static nested class in Java i.e. onto a class which is a class which is a class member, not an instance member of the outer class. Static nested classes are used if one wants to nest one class within another class.

The Java2D package contains examples of static nested classes. It defines shapes like `Ellipse` and `Rectangles` as abstract classes. For these abstract classes there are single and double precision implementations. These concrete classes are nested within their abstract super classes.

![Diagram showing static nested classes](image)

Figure 4.69: Concrete single and double precision implementation classes nested in an abstract superclass

One benefit of such nestings is that there is reduced namespace pollution. The user may be using an `Ellipse::Float` or an `Ellipse::Double` without the single and double precision implementation classes being new independent entries in the global namespace.

The classes contained in the metaclass are mapped onto static nested classes in Java:

```java
public abstract class Ellipse
{
    ...

    public static class Float extends Ellipse
```
4.33.3 Inner classes

A containment relationship between two classes maps onto inner classes in Java. An inner class is a class which is defined as an instance member of another class. The following holds for an inner class:

- Instances of the inner class can only exist within an instance of the outer class.
- The inner class instance has a handle to its outer class instance.
- The inner class instance has access to the private members of its outer class instance.

Normally, one would define the inner class private. Private inner classes can only be referenced within the outer class itself. No external class may have any dependency on a private inner class. This does not mean that external classes cannot obtain a handle to an instance of a private inner class. However, if they do obtain a handle to an instance of a private (from the outer class) then that handle will typically be the type of some public interface.

Below we show an simplified excerpt of the Java mapping:

```java
public static class Double extends Ellipse {
    ...
}
```

```java
public interface Iterator {...}
public class LinkedList implements List {
    public Iterator iterator()
}
```

Figure 4.70: A linked list using inner classes for nodes and its iterator implementation
```java
{
    return new LLIterator();
}

...

private class LLIterator implements Iterator
{
    public LLIterator()
    {
        current = head;
        // possible because inner class instance has
        // handle to outer class instance
    }

    public Object next()
    {
        Object content = current.getContent();
        current = current.next();
        return content;
    }
    ...
}

private static class Node
{
    ...

    private Object content;
    private Node next, previous;
}

private Node head;
}
```

Every instance of a linked list iterator is specific to a specific instance of a linked list, i.e. an instance of a linked list iterator can only exist within an instance of a linked list. Using an inner class, enforces this requirement.

The linked list iterator is specific to the implementation of a linked list. It is not meant for reuse outside the linked list implementation. As modifications are made to the linked list implementation, it may require making modifications to the iterator. Such modifications will not affect anything else as no external class may have a dependency on the implementation class of a linked list iterator.

Users of a linked list request iterators and will receive a handle to an object which is of type `LLIterator`. The handle type and hence the dependency is, however, the public `Iterator` interface.

Within the linked list iterator implementation we can directly access the private members of its outer class, the `LinkedList`. This allows for narrowly collaborating classes without breaking encapsulation (the private members are still only accessible from within the class).

**Note**

Unlike the friendship relationship which breaks encapsulation, inner classes provide a mechanism for specifying intimately collaborating classes without breaking encapsulation.

The node is a class used by the linked list implementation. It is not meant for re-use. Declaring it a private static nested class prevents any other class having a dependency on the linked list’s node class. It does not, however, require a handle to the linked list itself and the linked list may, at times, create working nodes which are not part of the linked list. For these reasons the `Node` class has been declared a static nested class and not an inner class.
4.3.3.1 Classes local to a method and anonymous inner classes

Java also supports classes which are defined within a method. This is not supported in UML. Furthermore, these classes may be anonymous in Java. An example of an anonymous inner class is an unnamed subclass of a Thread class:

```java
new Thread() {public void run() {doSomething();}}.start();
```

Here we define an unnamed subclass of Thread which overrides the run() method, create an instance of the anonymous class (without ourselves maintaining a handle to that instance) and register that instance with the thread scheduler via the call to start().

4.3.3.4 Nesting interfaces

At times one wants to nest one interface within another. This is commonly used if the one interface represents a base contract while the second interface extends the base contract.

Consider, for example, interfaces used to provide a read-only view and a read-write view onto an object. The read-only view is an interface which publishes those services which do not change the state of the object. The read-write view is a nested sub-interface which adds setters and other services which change the state of the object. The implementation class implements the sub interface, providing an implementation of all the services defined in the base interface as well as the contained sub-interface.

The mapping onto Java is straightforward:

```java
public interface Cheque {
    public Date getPayDate();
    public Money getAmount();
}

public interface Mutable extends Cheque {
    public void setPayDate(Date newPayDate);
    public void setAmount(Money newAmount);
}
```

The implementation class now implements Cheque.Mutable.

```java
public class ChequeImpl implements Cheque.Mutable {

```
public Date getPayDate() {...}

public Money getAmount() {...}

public void setPayDate(Date newPayDate) {...}

public void setAmount(Money newAmount) {...}

Depending on the user role, we can provide a read-only handle to the user by returning the ChequeImpl as a Cheque or a read-write view by giving them access via a handle of type Cheque.Mutable.

### 4.3.3.5 Nesting a class in an interface

This is often done in order to package the classes required for a contract with the contract (the interface) itself.

For example, a lunch order sent to a caterer may have a data structure which is different to that of other orders. Similarly, the invoice we require from a caterer may have information specific to catering invoices. In such a case we may want to package the order and invoice classes with the contract using a containment relationship.

![Diagram](image)

Figure 4.72: Nesting classes within an interface

Similarly, we can define an InsufficientFunds thrown upon there being insufficient funds in an account with the Account contract itself.

The mapping onto Java is straightforward:

```java
public interface Caterer {
    public Invoice processOrder(Order order);

    public class Order {...}

    public class Invoice {...}
}

This defines a Caterer.Order and a Caterer.Invoice.

In a similar way we may nest the InsufficientFunds exception within the Account interface:

```java
public interface Account {
    ...

    public TransactionConfirmation debit(double amount)
        throws InsufficientFunds;
    ...

    public class InsufficientFunds extends Exception {...}
```
4.34 Mapping UML containment relationships onto XML

An inner class in UML can be mapped onto an anonymous inner class in XML. For example a line item is specific to an order and one may want to prevent any re-use of the LineItem class outside an instance of an Order. This may prompt one to define the LineItem as an inner class of the Order class.

![Figure 4.73: Line item as an inner class of order](image)

This inner class would be mapped onto an anonymous inner class in XML schema:

```xml
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:complexType name="Order">
    <xs:sequence>
      <xs:element name="orderNo" type="xs:string"/>
      <xs:element name="lineItem" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="productId" type="xs:string"/>
            <xs:element name="quantity" type="xs:positiveInteger"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Here we define, within a lineItem element, a complex type (class) which does not have a name. This is a class specifically defined to specify the lineItem element and cannot be reused.

---

**Note**

Anonymous inner classes are used in XML if one explicitly wants to prevent reuse.

4.35 Summary of UML relationships

Figure 4.74 summarizes the UML relationships. It shows that these are conceptually specializations of each other and that we have weak and strong variants of “is a”, “has a” and “uses”.
4.35.1 Dependency

Instances of the one class, the user, make, at times, use of instances of the other class, the service provider. The latter is often modelled as an interface in order to decouple the user from any particular implementation of a service provider. For example, clients of the bank, upon spotting an ATM, may decide to use it in order to withdraw some cash from their account, but they do not maintain a message path to any particular ATM.

A dependency is called a "weak uses" because the user does not maintain a message path and is not in a position to, at any stage, send further service requests to the service provider.

4.35.2 Association

Association is used for two purposes. On the one side it is used purely for navigability. In the second case it is used for a client server relationship (or peer-to-peer in the case of binary associations). In either case, the object which has the association maintains a message path to the associated object.

It is conceptually a special form of dependency where the client still, at times, makes use of the service provider, but now the client maintains a message path to the service provider. For example, an amplifier has a message path to the speakers (the cables) in order to send service requests to them.

An association is called a "strong uses" because the client maintains the relationship and is in a position to send, at any stage, further service requests to the service provider.

4.35.3 Aggregation

Aggregation is a special form of association. The aggregate object still maintains a message path to the component. It still can make use of the components. For example, in the context of a portfolio calculating its value, it will request the value of each asset and sum them up.
However, in aggregation a state transition in the component may imply a state transition in the aggregate object, i.e. aspects of the component state are part of the state of the aggregate object. In our example, a change in the value of any of the assets results in a change in the value of the portfolio.

Aggregation is a weak has a relationship because it does not take exclusive control of the component. The component can be accessed directly and may be part of other aggregate objects. Furthermore, the asset can survive the portfolio. For example, a particular asset may be part of a number of different portfolios. A change in its value results in the value of multiple portfolios changing. Furthermore, one may decide to remove a portfolio (a particular grouping view onto one’s assets), but the assets would still survive.

### 4.35.4 Composition

Composition is a special type of aggregation (and hence also a special type of association and a special type of a dependency). If the component state changes, the state of the owner also changes. The owner also maintains the message path and may, at any stage, issue further service requests to the component.

Now we have, however, a “strong has a” relationship where the owner takes full responsibility for the component and encapsulates the component. If a user of the DVD player wants to send a service request to its laser, it will have to do so via the services offered by the DVD player itself. If the laser is broken, the DVD player is broken too (it is responsible for the laser). Finally, should we decide to scrap the DVD player, the laser will be scrapped also.

### 4.35.5 Realisation

Realisation is a weak is a relationship. It is used to show that a service provider implements an interface (and often a complete contract). This facilitates substitutability of one service provider with any other realising the same contract.

### 4.35.6 Specialisation

Specialisation is a very strong relationship which should be used with care. It is commonly used for data or value objects. Specialisation can be conceptually seen as special form of realisation in that the sub-class is a specialised realisation of the super-class. One can say, specialisation inherits substitutability from realisation.

It can also be seen as a special form of composition as every sub-class instance will create an encapsulated super-class instance through which it obtains the super-class attributes, services and relationships. The super-class instance for the sub-class cannot be accessed directly from outside the sub-class instance. It will also not survive the sub-class instance.

The super-class instance is part of the state of the sub-class instance. If the state of the super-class instance changes, the state of the sub-class instance changes too. For example, assume a home loan application inherits a loan amount from loan application. If the loan amount changes the state of the home loan application changes.

The sub-class instance also maintains a message path to the super-class instance (super in Java and base in C#). It is thus also a special for of association. It may, for example make use of a superclass service via super.serviceRequest().

### 4.35.7 Containment

Containment is a separate relationship where instances of one class can only exist in instances of another. There are examples of such relationships in nature.

### 4.35.8 Shopping for relationships

In order to determine the correct relationship between two classes one can take a requirements driven approach - similar to a shopping list for relationships. In either case one should always choose the weakest relationship which fulfils one’s requirements.

The process of determining the correct relationship goes along two legs. On the one side you are trying to establish the type of dependency between the two classes. On the other side you will assess the level of substitutability and inheritance required.
First we assess whether there is a dependency between the classes. If instances of one class, $A$, never make use of instances of another class, $B$, and if one also does not need to be able to navigate from an $A$ to a $B$, then there is not much of a relationship between these classes. Otherwise there is at least a dependency of $A$ on $B$.

Next ask yourself whether instances of $A$ should maintain a message path to instances of $B$. If so, upgrade the dependency to an association. If not, leave the relationship as a dependency.

If we reached this point, we have at least an association from $A$ to $B$. Next you can ask yourself whether any change in the state of an instance of $B$ results in a change of state in the instance of $A$ which maintains an association to it. If the answer is yes, then upgrade the association to an aggregation relationship. Otherwise leave it as an association.

If we reached this point we have at least an aggregation relationship from $A$ to $B$. Next, you can ask yourself whether the aggregate object needs to take full control of the component, or whether other objects should be allowed to access the component directly. If full control is required, then upgrade the relationship to a composition relationship. Otherwise leave it as an aggregation relationship.

---

**Note**

If you decided on composition, you can do the following test to check whether you perhaps made an error. Check whether it would make sense for the component to outlast (survive) the owner. If the answer is yes, then the relationship could not have been a composition relationship.

---

Next let us look at the plug-ability requirements. If the class should be pluggable (i.e. if the service provider should be substitutable), then one should introduce a contract for the service requirements. In the bare form, the contract is simply an interface and we have a realisation relationship.

In order to assess whether you should upgrade the realisation relationship to a specialisation relationship, assess whether you want to inherit common properties and services.

---

**Note**

In general we would recommend to favour interfaces and realisation above inheritance and specialisation. The latter tends to result in very rigid designs which are difficult to modify. One may choose to use specialisation only for value or data objects which do not perform significant functionality.

---

### 4.36 Templates

UML templates are model elements with unbound formal parameters. They are commonly used to define families of classes or operations, but may also be used to define families of packages.

A template based model elements are abstract. Binding the template parameters to actual types generates concrete model elements (e.g. concrete classes).

#### 4.36.1 Template classes

The most common use of templates is that of class templates. A class template defines a family of classes whose members are generated by binding the template parameters to different types, constants or operations.
Figure 4.75: Vector template

For example, Figure 4.75 shows a vector class template, defining a parametrized vector class. \( \text{Vector<Double>} \) and \( \text{Vector<Complex>} \) are two different vector classes generated from that template by binding the template type to \( \text{Double} \) and \( \text{Complex} \) respectively.

We can either show how the data type, \( \text{DoubleVector} \) is generated from the template by binding the template parameter, \( T \) to \( \text{Double} \), or we can directly define an object as an instance of \( \text{Vector<Complex>} \).

### 4.37 Selective views

Recall that the UML diagrams provide a mechanism

- to insert information into the UML model, and
- to provide a graphical view onto selective aspects of the underlying model.

There is thus no need to show all static information of a class in each class diagram for that class. Some class diagrams may just refer to the class without showing any details. In others we hide the access levels or show only public members of the class. Alternatively we may hide the method signatures (the parameters clients need to provide when requesting a service or the type of information/object the service provider will return to the client upon successful completion.

In addition we may delete certain attributes and/or services from a diagram (removing them from that view onto the underlying UML model) without removing them from the model itself. The UML tool will then put three dots in the appropriate compartment to let the viewer know that this is an incomplete view of a class and that there are further attributes and/or services which are not shown in this view (see Figure 4.76).
Figure 4.76: A class diagram which only shows a subset of the attributes and services of a class

When removing information from a diagram, the information is not necessarily removed from the model. Upon deleting a feature the UML tool may prompt you whether you would like to have this feature removed from the underlying model or not. The complete information of all model elements will be maintained in the underlying UML model. It is this UML model which assists one in ensuring that the model is consistent.

**Note**

In general one would like to hide information which is not relevant to what one would like to communicate through the diagram. For example, when documenting a particular business or system process, one would like to show only those aspects of the static structure and the dynamics which is relevant to that particular process and hide everything else.
Chapter 5

Sequence diagrams

5.1 Introduction

UML sequence diagram have evolved from scenario diagrams. Like scenario diagrams they can be used to show examples of how objects interact in the context of collaborating to realize some use case (at some level of granularity). They show the services requested as well as any object exchanged between the role players.

UML has, however, expanded the notation for sequence diagrams considerably in order to be able to

- reference interaction patterns defined in another sequence diagram,
- show multiple alternative scenarios in a single diagram by having improves support for conditional flow,
- provide a more solid notation for iterations, and
- have more robust support for concurrencies in sequence diagrams.

Nevertheless, sequence diagram do not provide a very natural framework for displaying multiple scenarios, i.e. conditional flows, iteration and concurrencies. There main value still lies in being able to intuitively and simply document the interaction around a particular scenario.

5.2 Simple sequence diagrams

In a sequence diagram, the role players (i.e. the objects participating in the collaboration) are aligned along the horizontal axis while time increases along the vertical axis. The sequence diagram then shows messages exchanged between these role players in time order.
5.2.1 The time axis

Time increases in the vertical direction. The time sequence of the messages is thus specified by their vertical positioning.

5.2.2 The objects

Along the horizontal axis we have aligned the objects participating in the collaboration. Normally one draws the sequence diagram for a particular level of granularity. The objects (the user interface, controller, cashier and product dispenser of the vending outlet) are all from the same level of granularity. The diagram does not show any low level components like the coin counter or the mechanism which extracts the products from the store.

When specifying an object one may choose an instance of a class (represented by :ClassName). Alternatively and often preferably one would leave the selection of an implementation class to the implementation phase. During the design phase one would only specify that the object is an instance of some or other class which implements the specified interface. Any object chosen to realize the business or system process will have to implement the interface (and preferably the full services contract).
5.2.2.1 The life line

Below each object there is a dashed line. This is the life line, i.e. the period over which the object exists. In our example all objects exist across the entire time of the business process.

5.2.2.2 The activation bar

The bars on top of the life line of the objects are the activity bars. They specify periods where the object performs some activity which is relevant for the business or system process being documented. When this information is not important (as is often the case) one may either leave out the activity bars entirely or one may draw activity bars spanning the full time of the process.

5.2.3 Service requests

Service requests are shown as solid lines with an arrow at the service provider side, i.e. the arrow points from the client requesting the service to the service provider from whom the service is requested.

The service request will specify the name of the service requested as well as a list of objects (within round brackets) which the client provides to the service provider upon requesting the service.

5.2.3.1 Message to self

If the service request message returns onto the object itself, the service is requested from itself. This effectively models an activity performed by the object itself. For example, the customer performs the activity to take the change and the product from the vending outlet.

5.2.4 Returns

One may optionally show the returns from the service provider to the client. These are drawn as dashed lines with an arrow pointing from the service provider to the client who requested that service.

The return message may specify an object which is returned to the client. This object may, of course, be an instance of a composite class with a whole range of components.

5.2.5 Levels of granularity

The services requested from the interfaces (or classes) in a sequence diagram would automatically feed in as services in the static model for that interface or class. Each of these services would have to be realized through a lower level business or system process. There would be lower level sequence and activity diagrams for these lower level processes.

5.3 Message types

A message is used to either communicate some information or to request a service. In either case a client may choose whether to wait for a response or not.
UML supports the following message types in interaction diagrams:

- **Synchronous request and return** A synchronous request is a blocking service request where the business or system process of the client or sender blocks until a response has been received. A synchronous request is shown in UML using a solid line with a filled arrow head pointing from the client or message sender to the service provider or message consumer. The return of a synchronous request must always go back to the client who made the initial synchronous request. It is shown as a dashed line with an open arrow head pointing from the service provider or message consumer to the client or message sender.

- **Asynchronous messages** Asynchronous messages are non-blocking. The client does not wait for a response and directly continues with its business or system process after having dispatched the message. Asynchronous messages are shown in UML using a dashed line with an open arrow.

- **Object creation** Often one needs to domain objects within a business or system process. For example, at some stage in a vending process one needs to create an invoice. This is done in UML using a create message which points to the actual object diagram. The object did not exist before the create message and no life line is present for these earlier times.

**Note**
In some UML tools you first put the object on the horizontal axis and then draw a create message to its life line. The tool will pull down the object diagram to the point where the create message was received.

- **Destroy message** A destroy message destroys the object, i.e. the object will no longer exist after the destroy message has been received. This is shown in UML by putting a cross at the end of the life line of the destroyed object.
• **Lost message** A lost message is a message for which the sender is known, but the receiver is not. It is shown in UML by drawing a message from the life line of the sending object to a circle representing an unknown message recipient.

• **Found message** A found message is a message for which the receiver is known, but the sender is not known. A found message is shown by drawing a message from a solid circle to the life line of the object which receives the message.

### 5.4 Timing constraints

UML supports specifying timing and duration constraints in sequence diagrams. The sequence diagram provides a natural platform for this as time in on the vertical axis. One simply puts labels on the axis and specifies the time or duration constraints relative to these labels.

For example, Figure 5.3 specifies that

- payment must be received within 10 seconds from the payment request,
- the cashier must have completed the process of dispensing cash within 4 seconds of having been requested to do so, and
- the product store needs to have completed the dispensing of a product within 6 seconds of having received the dispensing request.
5.5 Interaction references

At time the interaction shown within a sequence diagram may be overwhelming. In such cases one may want to take a cohesive part of the interaction out into a separate sequence diagram. Another reason for wanting to do this is that there may be aspects of the interaction which are common with the interactions for other use cases. One would like to define the common interaction in a single sequence diagram and then refer to that interaction from within any sequence diagram where that interaction is required. This can be done via interaction references.

Consider, for example, the interaction for processing a claim shown in Figure 5.4.

![Sequence Diagram](image)

Figure 5.4: Interaction for a process claim use case references the determineClaimPayout interaction

The referenced interaction will have to specified in another sequence diagram. For example, the referenced *determineClaimPayout* interaction is shown in
Note
Interaction references should be used sparingly. An overuse of interaction references is usually symptomatic for not having managed the levels of granularity effectively. A natural way of managing complexity without using interaction references to use a work break down structure approach where the one first shows the interaction across high level role players providing high level services and the shows. At the next lower level of granularity these high level services will be the use cases for the lower level components. The lower level sequence diagram shows the lower level interaction which realizes the higher level service.

5.6 Conditional flow (alt)

One would, for sake of simplicity, often use sequence diagrams to document only single scenarios, particularly when discussing a business process with the client. Nevertheless, sequence diagrams can be used to show alternate flows, i.e. multiple scenarios. This is done using alt fragments with multiple compartments for the alternate scenarios.
Figure 5.6: Sequence diagram for the buyProduct use case of a vending outlet showing alternative flows

To show alternate flows one uses an \textit{alt} (alternate) fragment with multiple compartments separated by a dashed line. Each compartment encapsulates a flow which is followed if a condition holds true. The condition is inserted in the compartment using the standard conditional notation (square brackets).

In a business process execution one will thus execute one of mutually exclusive flows, each within its own compartment in an \textit{alt} fragment. Having completed the sequence specified in the relevant compartment of the alt fragment, flow continues after the alt fragment.

In order to show multiple branchings of a flow, one may nest \textit{alt} fragments as is done in Figure 5.6.

### 5.7 Iteration in sequence diagrams (loop)

Iteration is not natural to sequence diagrams. UML has, however added a looping construct to support a notation for specifying iterative execution of an interaction. This is done by encapsulating in a fragment containing a loop operator.
Figure 5.7: Iteration via the loop operator
Figure 5.7 shows an ATM repetitively requesting the pin for an ATM card until either the correct pin has been entered or until the user entered the pin three times incorrectly.

5.8 Concurrencies in sequence diagrams (par)

Sequence diagrams do not accommodate concurrencies in a particularly natural way. Activity diagrams have a more intuitive and readable notation for concurrencies and synchronization points within a business or system process.

If one wants to show concurrencies in a sequence diagram, one uses a par (parallel) fragment which shows the messages exchanged concurrently in separate sub-divisions of the fragment. There may be two or more subdivisions, each representing a separate concurrent process (a separate thread of activities).
Chapter 6

Activity diagrams

6.1 Introduction

An activity diagram represents a decomposition of an activity into its components.

Activity diagrams provide a very natural and intuitive notation for documenting processes. They are commonly used to document both, system and business processes.

Unlike sequence and communication diagrams, which are particularly suited to documenting individual scenarios, activity diagrams are used to show a process in general.

6.2 Basic activity diagrams

A simple activity diagram shows a sequence of actions performed within some process defined by the activity diagram. For example, Figure 6.1 shows a simple activity diagram for the process of a sale.

- Activity diagram shows process across activities (rounded rectangles). Each activity is only performed once the previous activity has been completed.
- Filled round circle = entry (start of process) node. Small filled circle embedded in empty circle = exit node (end of process).
- Lines with arrows = edges. Edge shows transition from performing one activity to performing another activity.
- A transition may be triggered by an event. The event name is written on the edge.
- If there is no event, the next activity is automatically executed after the previous one has been completed (this is an automatic transition).

Figure 6.1: Simple activity diagram for the process of a sale

6.2.1 Activity and actions

An activity is an executable behaviour for which there may be a process defined. The process for an activity is shown in an activity diagram. It may be composed of lower level activities executed sequentially or concurrently, decision nodes and ultimately actions.

An action node represents a single processing step within an activity, i.e. a step which is not broken down into lower level work flow steps.
6.2.2 Edges and events

An activity edge represents the transition from one activity to another. The activity the edge points to is only executed once the source activity has completed executing.

6.2.3 Events and automatic transitions

If there is no event on the transition, the transition to the next activity is automatically executed after the previous activity has been completed. This is called an automatic transition.

Alternatively the transition can be triggered by an external event. In such a case the event name is put on the transition.

For example, the transition from the entry node to the generate invoice activity is performed when an order is received. On the other hand, the transition from process payment to package goods is an automatic transition which is not triggered by an external event. Automatically after having processed the payment, the goods are packaged.

6.2.4 Entry and exit activities

An entry node is shown as a solid circle. It represents the start of the process. An exit node, which represents the end of the process is shown as a filled circle embedded in an empty circle.

The advantage of entry and exit nodes is that one can encapsulate the entire activity and that external activities or states can use the activity without knowledge of the internal process.

6.3 Decision and merge nodes

Activity diagrams provide an intuitive notation for showing decision points in a business or system process and hence for showing multiple scenarios in a single diagram.

A decision node is shown in UML as a diamond. On each of the outgoing edges there must be a condition. The flow will continue along that path for which the condition evaluates to true.
6.3.1 Formulating the conditionals

In order to have a well defined process the conditionals on the outgoing edges must satisfy the following criteria

1. They must be non-overlapping or mutually exclusive -- only one of the conditionals may, for any scenario value to true.

2. The conditionals must cover the full domain in order to always lead to one path being followed. Otherwise the process would get stuck at the decision point.

6.3.2 Merge nodes

At times a process may follow different paths for some work flow steps before merging back into the same core process. To merge alternative paths back into a single flow, one uses a merge node drawn as a diamond with multiple edges leading into it and a single edge leading to the next activity of the main flow.

Figure 6.2: A simple activity diagram for the buyProduct use case of a vending outlet
Figure 6.3 shows a work flow for the preparation of the course notes one week prior to the start of a course. If the course is presented for the first time, it is sent for proof reading. On the other hand, if the course has been presented before, then all corrections and additions requested by the previous presenter are applied. In either case we then merge the process into a common flow for the printing of the course material.

6.4 Activity partitions (swim lanes)

For both, system and business process modeling, it is often useful to show which object or role player is responsible for which activity. This is particularly important when one wants to show how different role players collaborate to realize a use case.

For example, Figure 6.4 has one partition for the customer and one for the vendor. It shows how the two parties collaborate to realize a sale. The activities drawn in the customer’s partition are done by the customer, while those drawn in the vendor partition are the responsibility of the vendor.
6.5 Object flow in activity diagrams

Interaction diagrams provide perhaps the clearest view of the objects exchanged between role players in a process. Activity diagrams can, however, also be used to show object flow within a business or system process. This can be done by inserting an object diagram into the edge representing the transition from one activity to another. The object flows from that role player who is responsible for the previous activity to that role player who is responsible for the next activity.

Figure 6.5 shows the objects exchanged between the customer and the vendor during the process of a sale.
Structured or nested activities are used

- to abstract the business process by showing only higher level activities hiding the detailed actions for them, and
- to show common transitions which are available for all nested activities.

A structured activity is shown as a dashed rectangle with rounded corners with the stereotype `<structured>`. A transition which is common to the nested activities is shown by drawing an edge from the structured activity to the new activity.
In Figure 6.6 one can request help in any of the activities performed while the game is running. Upon the ok event the system is taken back to the activity it came from. Similarly, as your boss enters your office you can immediately exit the game, irrespective of whether you are busy selecting the players, the level or you are busy playing.

Note that we defined an entry node for the internal process. This decouples the external process from the internals of the structured activity.

Structured activities are also common used to show common cancel paths accessible for a subset of the user workflow. For example, in Figure 6.7 the customer can cancel the purchase while the product has not yet been fully paid for.
6.7 Concurrences in activity diagrams

If certain steps in a process are independent of one another, they can be performed concurrently. Such activities should be shown as concurrent in a implementation neutral process specification. When the implementation mapping is done, the implementors may still choose to implement such work flow steps sequentially. Modeling them as concurrent implies that they could be implemented concurrently.

6.7.1 Forking

Concurrences are introduced in an activity diagram via forking, i.e. a single thread of activity forks into multiple concurrent threads of activity. A fork is shown as a bar with a single edge leading into the bar and multiple edges leading out of the bar, each edge representing a flow or thread of activity.
6.7.2 Flow final node

A flow final node is used to show the end of a flow or thread of activity. It is drawn as a circle with a diagonal cross. The process may still continue as other threads of activity may still be active.

6.7.3 Synchronization

At times one needs to block a business or system process until certain concurrent activities have been completed. This is specified in UML using a synchronization bar. Synchronization bars are similar to forks except that multiple threads of activity lead into the bar while a single thread of activity leads out of the bar.
6.8 Sending and accepting signals

UML defines special activity types for sending and receiving signals.
Note
A signal is the asynchronous communication of information.

This enables one to specify that at certain points in a workflow one waits for a signal while other activities send signals to external objects.

For example, Figure 6.10 shows a business process which waits for an enrollment request signal. Having received one, it spawns a workflow processing the enrollment request while waiting for the next enrollment signal.

At some stage in the business process signals are sent to the client, providing the client the invoice and sending payment requests asynchronously to the client.

6.8.1 Interruptible activities

Figure 6.10 uses an interruptable activity to specify that the process of sending payment requests in regular intervals to the client is interrupted by a payment notification.
6.9 Object pins

Instead of showing the object flows as object diagrams on the transition edge, one can specify the inputs and outputs of an activity more explicitly using object pins. This is done by attaching the object pin to the activity.

For example, Figure 6.11 shows that the pantry sources the order ingredients from an order it receives from the waiter and provides the ingredients to the kitchen.

![Object pins diagram](image)

Figure 6.11: Object pins showing the inputs and outputs across the activities for preparing a meal.

An activity may generate multiple outputs or consume multiple inputs. In UML one can use one pin per input or output parameter. For example, the kitchen receives the order from the waiter and the ingredients from the pantry. It only commences the preparation of the meal once all input parameters have been received.

6.10 Expansion regions

Expansion regions are used to specify a set of activities which is done across a number of objects, typically on all the elements of a collection of objects received as input.

The activities per object can be performed

- in parallel (concurrently),
- sequentially (iteratively), or
- streamed (i.e. processing objects extracted from a input stream and feeding the results into an output stream).
For example, the kitchen receives a collection of order items. For each order item it sources the ingredients, prepares the order item and places it on a plate.

### 6.11 Exception and error handlers

Activity diagrams can be used to show the transfer of control to exception and error handlers. In a contracts based approach

- *exceptions* are used by service providers to notify clients that a service they requested is not going to be provided because a particular precondition was not met, while
- *errors* are used to notify the client that a service which was requested is not going to be provided because the service provider encountered a problem which prevents it to satisfy its contractual obligations.

In either case, the client may have a process which should be followed in the case where a service provider does not provide the requested service. This is done via exception and error handlers.

In UML the transition to an exception or error handler is shown by a transition which has the stereotype icon for a exception handler transition, that of a zig-zag line with an open error pointing to the exception or error node of an exception handler. The exception or error node is the object node for the exception which is caught. Alternatively the stereotype icon can be used directly, showing the transition itself as a zig-zag line.
For example, if the product dispenser encounters an internal problem preventing it from realizing the dispense product service, it may notify the client via a suitable error (or exception). Figure 6.13 shows the error handler which is executed when such an exception is caught.

6.12 Call operations, activity parameters and assigning behaviours/processes to services

UML supports the concept of a call operation resembling the activity of requesting a service from some object. This is particularly useful when showing how a service provider / controller assembles the process realizing the service it provides from lower level services it sources from other service providers. The core benefit is that the activity is directly related to a service providing traceability between the activities and the services they realize.

6.12.1 Call operations

UML supports the concept of an activity which requests a service. The service concrete service provider (class) or more abstract service provider role (interface representing a services contract) responsible for realizing the service is then shown in round brackets. The inputs and outputs of the service are shown as input and output nodes and do correspond to the service signature specification of the class or interface.

Note
Most UML tools will allow you to drag a service from a class or interface onto an activity diagram, generating a corresponding call operation for that service.
6.12.2 Assigning an activity/process to a service

Your UML tool will enable you to assign an activity diagram/collaboration to a service, thereby specifying that the service provider uses the process as specified in the collaboration to realize one of the services it offers.

As soon as a process has been assigned as behaviour for a service, the input and output parameter nodes corresponding to the inputs and outputs of the service are shown on the boundary of the structured activity representing the higher level process.

One can now use call operations to show how the service provider assembles the service it offers from services it sources from other service providers. An example is shown in figure 6.15: The business process for processing a sale.

**Note**
This is an alternative and often preferable notation to swim lanes, showing how a controller assembles a business process from services available to the organization/system.
Chapter 7

Communication diagrams

Communication diagrams provide another view onto an interaction diagram. Communication diagrams show more explicitly the message paths required between objects and the messages sent along these message paths.

Like a sequence diagram it documents the messages exchanged between objects in time order, but while time in a sequence diagram is represented by an axis, time in a communication diagram is represented by a numbering system.

Figure 7.1: Communication diagram for the buy product use case of a vending machine
Chapter 8

Composite structure diagrams

8.1 Introduction

Composite structure diagrams are used to specify

• the objects participating in a collaboration,

Note
The purpose of a collaboration between objects is usually that of realizing a use case.

• the internal parts of components and the connectivity between them,

• the ports through which the object interfaces with its environment.

8.2 Documenting the structure supporting a collaboration

Composite structure diagrams can be used to specify the objects which participate in a collaboration, as well as the connectivity between them. This is done by inserting the classifiers (e.g. classes or the more abstract interfaces representing contracts for the components participating in the collaboration) into a collaboration. The collaboration itself is drawn as a dashed ellipse.
8.3 Composite structure diagrams to specify object interfaces

Composite structure diagrams can be used to specify the ports through which

- external objects can interface with an object,
- the object interfaces with external objects.

A port is an interfacing object through which an object either sends or receives messages. Often these are service request messages which are either sent or processed by the object.

If an object publishes certain services through a particular port, one can encapsulate those services within an interface which can be expanded into a services contract.

Similarly, if the object requires a certain set of services around a responsibility domain from external service providers, then one can
• specify through which port (interfacing object) one interfaces with the external service provider, and
• the interface or services contract which must be realized by the external service providers.

![Figure 8.2: Ports through which the restaurant interfaces with the environment](image)

For example, Figure 8.2 shows that
• the waiter is the interface of a restaurant through which the dining services are requested,
• take away services are requested via a telephone portal,
• the restaurant requests delivery services from delivery services providers through the telephone portal, and
• it places orders with vendors via its telephone portal.

### 8.4 Documenting the internal structure of objects

Composite structure diagrams are useful to show

• the internal parts of an object including those parts which interface with the environment (the ports), and
• the connectivity between these internal parts.

An internal part is shown by inserting a part into a classifier (e.g. a class) and a port is inserted on the boundary of the classifier.
For example, Figure 8.3 shows the internal parts of the vending machine including the ports through which the vending machine interfaces with the customer and the connectivity between the internal parts.
Chapter 9

Component diagrams

9.1 Introduction

Component diagrams show

- pluggable components,
- the interfaces they provide,
- the interfaces they require, and potentially
- component artifacts.

9.2 Components

Components are pluggable service providers. They are pluggable by virtue of realizing one or more services contracts. Components are thus concrete modular parts which hide any implementation details around the services they provide and their current state behind external interfaces. Components realizing the same services contract may be readily exchanged.

Components should be only loosely coupled to other components. As such one would not want a direct dependency between two components. Instead the coupling would be via services contracts for any required service providers. These would be represented by required interfaces.

In UML a component is shown as a class diagram with the component stereotype. This can be represented as a component stereotype icon which is a rectangle with two smaller rectangles protruding on the left edge.

Figure 9.1 shows a show-case product dispenser as a pluggable product dispenser (providing the services required from a product dispenser). The product dispenser requires the services of a product store which is itself pluggable. This is shown via a required interface for product store.
9.3 Artifacts

Artifacts are physical resources which contain the information about how the objects which will be executing the business or system processes are created and deployed. They may be

- software resources (e.g. libraries, JAR files, dynamic link libraries) from which infrastructure components (e.g. application servers, operating system components, ...) can create the objects executing the system processes,
- the training and deployment plans through which an organization can develop organizational components executing specified business processes (e.g. the detailed process followed to create the components and deploy a new retail outlet),
- the detailed construction plans from which manufacturing machines and manual labourers can produce physical machines executing system processes (e.g. the production plans for a vending machine),

![Diagram of artifact documenting the manufacturing process for a vending machine](image)

Figure 9.2: Artifact documenting the manufacturing process for a vending machine

Figure 9.2 shows the manufacturing plans as an artifact for the deluxe vending machine.

9.4 Relationships between components

Components should preferably not have any direct dependencies on one another. Instead, they should be only loosely coupled via respective services contracts or interfaces.

Component diagrams can be used to show how components make use of each other and how higher level components are assembled from lower level components. For example
Figure 9.3: High level component diagram for a vending machine
Chapter 10

URDAD for technology neutral business process design

10.1 Background

10.1.1 Introduction

URDAD, the Use Case, Responsibility Driven Analysis and Design methodology [Solms_2007_URDAD, KGK_2007_AFCS], [Solms_2008_GeneratingMdaModelViaUrdad] aims to provide a simple and implementable algorithmic analysis and design methodology which generates a technology neutral design model satisfying accepted requirements for a "good design". The resultant model is meant to represent the Platform Independent Model (PIM) of the Model Driven Architecture (MDA) published by the Object Management Group (OMG), [Siegel_2001_DIOM, Frankel_2003_MDAA].

URDAD has evolved from the need to

• make design a repeatable engineering process with well defined inputs and outputs,
• make the design simpler by defining a step for step algorithm for the design process,
• enforce accepted requirements for “good design” through the design methodology itself,
• separate conceptual design from its realization (e.g. the implementation technologies and deployment architecture), i.e. to have a design which survives architecture and technology changes,
• have a design methodology which enables one to manage levels of granularity/refinement effectively, and
• the need to project out services contracts for all service providers required across all levels of granularity.

10.1.1.1 Bibliography


10.1.2 Architecture versus design

The architecture of an organization provides the core infrastructure within which the business processes are deployed. It needs to provide a suitable environment within which the organization can realize its vision and mission and its core quality offerings. Business process design, on the other hand, is concerned with defining the processes through which the services which generate stakeholder value are realized. These are driven by specific needs and functional requirements. The processes will specify what needs to be done when and ultimately which organizational components are responsible for which work flow steps.

The business processes deployed within a particular architecture acquire through the architecture the core organizational qualities

- reliability, the ability to realize one’s obligations to clients and other stakeholders reliably,
- scalability, the ability to handle large and/or varying volumes of service requests,
- low cost, the ability to offer services or products at low cost

and potentially a range of other quality requirements.

The architecture of an organization should encompass the full infrastructure into which business processes are to be deployed. This infrastructure which we refer to as the enterprise architecture spans organizational and systems architecture. The business processes are deployed across the entire enterprise architecture.

The enterprise architecture must realize the architectural requirements for the organization. These requirements include

- the scope of services to be deployed within the enterprise architecture,
- the quality requirements which are to be realized across the business processes deployed within the enterprise architecture,
- the integration requirements between the organization and its environment (clients, service providers, and observers),
- as well as any architectural constraints placed by the stake holders on the architecture.

The functional requirements for a user service (use case), on the other hand, include

- the deliverables as required across the stake holders,
- the conditions under which the service may be refused without breaking the contract,
- the desired user work flow
- the messages and objects which need to be exchanged with any other external role players as well as any requirements requirements around them (e.g. data structure requirements), and
- any activities required by any of the stake holders.

While the functional requirements are realized through work flow, the architectural requirements are realized through infrastructure.

10.1.3 URDAD in the context of OMG’s Model Driven Architecture (MDA)

OMG’s MDA forms the bases for most model driven development (MDD) processes [Selic_2003_TPMD, Schmidt_2006_MDE]. A high level view of a model driven approach is shown in Figure 10.1].
The input for the technology neutral business process design are the functional or use-case requirements. The output of the design phase is the Platform Independent Model (PIM) which can be mapped onto one’s choice of implementation architecture and technologies resulting in a Platform Specific Model (PSM). Both, the PIM and the PSM are UML models. The Platform specific model is then taken through an implementation mapping which includes the generation of all deployable artifacts including the code, the database structures, the deployment scripts, the user documentation. MDA effectively separates design from architecture.

URDAD targets
• the analysis phase resulting in a use case contract, as well as
• the design phase resulting in a technology neutral business process design.

The use case contract contains both, functional and non-functional requirements around a use case. The functional requirements drive the design process while the non-functional and in particular the quality requirements like scaleability, reliability, integrability, ... requirements drive the architectural process. The output of the architectural process is an infrastructure into which the business processes are to be deployed. This may span, both, organizational and systems architecture as business processes will often be realized across manual and automated processes.

10.1.3.1 Bibliography


10.1.4 URDAD in the context of OMG’s Model Driven Architecture (MDA)

URDAD is usually embedded within an iterative realization or development process. A typical model driven development process is shown in Figure 10.2. Note that the technology neutral business process design is performed by business analysis. The technical team comprising both, architecture and implementation (development), is responsible for the realization of the business process within the chosen architecture and technologies.

1 The implementation mapping around manual business process steps would typically involve training of workers.
After passing quality assurance and actual deployment, operations takes over the management of the business process execution.

10.1.5 Architecture versus design

URDAD has grown out of Responsibility Driven Design (RDD) methodology pioneered by Rebecca Wirfs-Brock and Brian Wilkerson (see [WW_1989_OOD, WM_2002_OD, WirfsBrock_2007_TDS]). Like RDD, URDAD focuses during the early stages of the design on identifying and assigning responsibilities. Also, like RDD, URDAD puts a lot of emphasis on client-server contracts. URDAD adds a step-by-step algorithm which generates different layers of granularity, enforces decoupling within each level of granularity via work flow controllers and enforces, through the methodology, a number of widely accepted design principles.

The ICONIX process from Doug Rosenberg discussed in [RD_1999_UDOM] provides a structured process for evolving the static model from the collaboration requirements, but are not really responsibility driven, nor do they project out clean layers of granularity.

Methodologies like the Rational Unified Process [?] and Extreme Programming are incorporate aspects of a design methodology, but are, in many respects more software development than design methodologies.
10.1.5.1 Bibliography


10.2 Requirements for an analysis and design methodology

10.2.1 Introduction

The design methodology should satisfy general requirements placed on a methodology. Here we can be guided by Edward Berard who lists some general requirements for a methodology.

In addition, the design methodology should lead one to a "good design" To this end we need to obtain an understanding of the criteria one would use to assess two various design solutions in order to determine "the best solution". For this we will look at widely accepted design principles.

10.2.2 Berard’s requirements for methodologies in general

Berard has formulated some general requirements for a methodology [Berard_1995_WIAM]. These are useful for assessing the usefulness of a methodology in general.

In particular, Berard states that one should require of any methodology that it

• can be used repeatedly, each time achieving similar results,
• can be taught to others within a reasonable time frame,
• can be applied by others with a reasonable level of success,
• is applicable to a relatively large percentage of cases, and
• is able, on average, to achieve better results than either other techniques, or an ad hoc approach.

10.2.2.1 Bibliography


10.2.3 Stake holder requirements for the technology neutral (business) model

We take the approach that there is no absolute quality, but that quality is a measure of the extend to which the stake holder’s quality requirements are fulfilled. Thus, before we can assess the quality of a (business) model, we need to identify the stake holders who have an interest in the model and then we need to elicit their quality requirements for the model.

The stake holders who have an interest in the analysis and technology neutral (business) model include

• Strategic management which is responsible for evolving the organization’s or systems vision and mission and the the business strategies through which these are realized.
• **Business Analysis** which is responsible for performing the stake holder requirements analysis and the technology neutral (business) process design itself.

• **Architecture** which is responsible for designing a suitable infrastructure hosting the functionality (business processes) in such a way that it enables the organization/system to realize its vision and mission.

• **Implementation** which is responsible for performing the implementation of the designed business processes. This includes developers who develop the automated implementation mapping as well as managers who train their staff to perform certain business process steps manually.

• **Quality assurance** which is responsible for assessing the extend to which the model realizes the stake holder’s functional and non-functional requirements and reporting any defects.

• **Operations** which is responsible for executing / overseeing the execution of the business processes and ensuring that the services are rendered to the user’s satisfaction.

<table>
<thead>
<tr>
<th>Quality requirement</th>
<th>Stake holders</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Business, Strategic Management</strong></td>
<td>Business Analysis</td>
</tr>
<tr>
<td>Completeness</td>
<td>x</td>
</tr>
<tr>
<td>Consistency</td>
<td>x</td>
</tr>
<tr>
<td>Simplicity, understandability</td>
<td>x</td>
</tr>
<tr>
<td>Modifiability</td>
<td>x</td>
</tr>
<tr>
<td>Cohesion</td>
<td>x</td>
</tr>
<tr>
<td>Testability</td>
<td>x</td>
</tr>
<tr>
<td>Traceability</td>
<td>x</td>
</tr>
<tr>
<td>Reusability</td>
<td>x</td>
</tr>
<tr>
<td>Technology neutral</td>
<td>x</td>
</tr>
</tbody>
</table>

Table 10.1: Stake holders in the technology neutral (business) model and their quality requirements

### 10.2.4 Accepted design principles

Robert C. Martin has compiled a widely quoted list of accepted design principles [Martin_2002_ASD]. Many of these design principles including the interface segregation, dependency injection and the Liskov substitution principles are realized by following a strict contracts based approach. One of the most important principles is the single responsibility principle which requires that at any level of granularity, each contract or class should address only a single responsibility. All the services should be narrowly aligned with the this responsibility focus. The reuse/release equivalence principle can be addressed by enforcing that one only reuses components which are released with realizing a published contract. If one would like to enforce the open-closed principle, one would do so separately from a design methodology.

Other widely accepted design principles include

• the simplicity principle, [WirfsBrock_2007_TDS],

• realizing a high level of reusability [LSW_1987_SRTB],

• enforcing clean layers of granularity, [Martin_2002_ASD, ?],

• testability across the levels of granularity [VM_2006_STNV], and

• bidirectional traceability across layers of granularity [Dick_2005_DT, ANRY_2006_MT].
10.2.4.1 Bibliography


10.3 Analysis and design principles supporting design qualities

Using guidelines provided for methodologies in general as well as the widely accepted design principles, we have identified a list of design activities which assist in enforcing these design principles. Figure Table 10.2 shows the final list of attributes we would want to realize within a design generated by the design methodology.

10.3.1 Enforce the single responsibility principle

The single responsibility principle and thus a high level of cohesion is enforced by grouping functional requirements into responsibility domains and assigning each responsibility domain to a separate services contract. Any system or organizational component as well as any external service provider realizing the full contract would be pluggable. Enforcing the single responsibility principle directly drives pluggability and reusability.

In addition, it makes each object more understandable through being able to understand the contract(s) it realizes without having to understand the way in which the services specified in the contract are implemented.

Finally, enforcing the single responsibility principle also facilitates simpler maintainability as

- maintenance is often required around a particular responsibility (enforcing the single responsibility principle leads to localized maintenance), and
- one can verify whether, after maintenance work, the contractual obligations are still met.

10.3.2 Fix the levels of granularity

In the context of a work break down structure, one is automatically led to define different levels of granularity [DM_1979_SASS]. In order to generate clean layers of granularity, URDAD starts by identifying the first level responsibilities and assigns them to contracts. The business process and ultimately the service provider contracts are specified for this level of granularity before going, in a structured way, to the next lower level of granularity.

This improves the maintainability of the design as changes to a business process often only need to be applied to the controller of a particular level of granularity.

Fixing the levels of granularity also improves the understandability and usability. It facilitates incremental understanding of a design, enabling one to look at a high level business process before specifying how each of the individual high level work flow steps are realized through lower level business processes. Furthermore, a particular role player often only needs to work at a specific level of granularity without there being a need to understand either the higher or lower levels of granularity.
<table>
<thead>
<tr>
<th>Design qualities</th>
<th>Complete</th>
<th>Consistent</th>
<th>Simple, understandable</th>
<th>Modifiable</th>
<th>Cohesive</th>
<th>Testable</th>
<th>Traceable</th>
<th>High reusability</th>
<th>Technology neutral</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single responsibility principle</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Clean layers of granularity</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Decouple via contracts</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Controller assemble process across service providers</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Connect layers of granularity</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Structure from process</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Well defined PIM elements</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Diagrams as views onto model</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Standard modeling language</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>

Table 10.2: Analysis and design principles supporting model qualities
10.3.3 Decoupling via services contracts

For each responsibility domain one assigns a separate services contract. The business process is designed to be realized across abstract service providers realizing these contracts. Design by contract rules are enforced ensuring the pluggability of service providers realizing the contract as well as the pluggability of specializations. This results in a loosely coupled design.

Enforcing that service providers realize services contracts increases the reusability of such service providers as the client can compare the services requirements with what is guaranteed through the contract.

Furthermore, enforcing contracts facilitates testability. It is difficult to write a sensible test if one does not know what the contractual obligations are which need to be tested.

A contracts based also approach improves maintainability and extensibility through enhanced pluggability and testability.

If all participants in a business process lock into contracts, the individual contracts can be realized within different technologies. A contract driven approach can be used to generate a technology neutral design.

10.3.4 Define for each level of granularity and each responsibility domain a controller

The controller is that object which takes ownership of the service and which controls the business process realizing the service. Introducing for each level of granularity and each responsibility domain a controller localizes the business process information within the controller and decouples the service providers from that level of granularity. Taking any business process decisions out of individual service providers and localizing it within a controller results in simpler business process management and maintenance. Furthermore, the increased decoupling leads to a higher level of reusability.

The introduction of a controller for each level of granularity also simplifies the design and improves understandability as one only needs to look at the controller logic to understand the business process for the current level of granularity.

10.3.5 Derive structure from process

Going from a use case directly to defining structure is difficult and often leads to complexity which may not be required. A simpler approach which leads to reduced complexity is that of defining first the process through which the use case is realized at a particular level of granularity. One can then project out the minimal structure required to support the process.

Furthermore, driving structure from process facilitates the traceability of any structural element to the process it supports and across the layers of granularity to the use cases for which it is required. Similarly, one can trace from a use case to the structural elements required across the levels of granularity to realize the use case.

10.3.6 Document relationships between layers of granularity

Finally, documenting the relationships between the layers of granularity is required to support full bidirectional traceability across the layers of granularity.

10.3.7 Business benefits of desired design attributes

Ensuring that the technology neutral business process design has the desired design attributes results in some direct business benefits.

In particular, maintainability/extensibility, simplicity and a high level of reusability all contribute to reduced cost and reduced time to market. The design being technology neutral, simplicity and maintainability/extensibility all result in increased business flexibility. Reliability is increased through simplicity and testability. Finally, design simplicity and the design being technology neutral enables business itself to take control and ownership around the business processes - i.e. instead of technology being in control of the business processes.

---

2 This strategy is also directly used within Services Oriented Architectures.
10.3.8 Bibliography


10.4 URDAD - the methodology

10.4.1 Introduction

URDAD provides a use case driven algorithmic analysis and design methodology generating a technology neutral business process design. One thus starts with a concrete user service for which the business process is to be designed.

The methodology enforces those design activities which have been identified as drivers for the desired design attributes and the resultant business benefits.

One of the core aspects of URDAD is the consistent management of level of granularity. Each level of granularity has both, an analysis and a design phase. This ensures that one does not have to identify all requirements across all levels of granularity up-front.

![Figure 10.3: High-level view of the URDAD methodology](image)

The methodology starts with the initial analysis phase followed by a design phase for the current level of granularity. This will project out the high level service providers required to realize the use case. One of these is selected as the subject for the next lower level of granularity. One then repeats the process for each of the lower level services (use cases) required from that services provider. One thus performs the requirements analysis followed by the business process design for each of these lower level services.

Figure 10.4 shows the URDAD analysis and design methodology in more detail.
Figure 10.4: More detailed outline of the URDAD methodology

We use the example of processing an insurance claim to illustrate the algorithm. This example is taken through two levels of granularity in order to illustrate the incremental refinement of the technology neutral business process design.

10.4.2 The analysis phase

10.4.2.1 Introduction

The analysis phase aims to elicit, verify and document the stake holder requirements. As one takes the business process design through lower levels of granularity, one revisits the analysis phase to elicit the lower level requirements around the individual workflow steps.

Often the high level analysis around the core business process and the lower level analysis around a functional requirement from a specific responsibility domain is done by different business analysts who focus on different business areas.

For example, a business analyst from the claims department would analyze the requirements and design the high level business process for processing a claim. The lower level requirements and business processes around how, for example, the claim is to be paid out or how the claim is to be valued could be performed by business analysts from the finance and valuations department of the organization.

10.4.2.2 Functional requirements

During the analysis phase one first identifies all those stake holders who have an interest in the use case. Stake holders are those objects who place requirements around a use case. Only once one has identified all the stake holders in a use case can one hope to elicit all functional requirements for that use case. Maintaining the linkage of any functional requirement to the stake holder who requires it facilitates full traceability of any business or system activity back to the stake holder requirements they realize and ultimately to the stake holder itself.
Figure 10.5: Functional requirements for the process claim use case.

Figure 10.5 shows the high level functional requirements for a process claim use case. Only the first level functional requirements should be included at this level of granularity. The detailed lower level functional requirements around the higher level ones are specified at lower levels of granularity.

For example, the functional requirement of generating a settlement offer may include lower level functional requirements like that of determining the value of the claim items and that of assessing to what extend the policy covers the claim.

Note
Often the detailed requirements around the different domains of responsibility are obtained from different role players; i.e. while certain domains of business may be able to provide information around the higher level business process, the details concerning lower level responsibilities are often determined from domain experts in the appropriate domains of responsibility.

10.4.2.2.1 Pre-conditions, post-conditions and quality requirements

In contract-driven development a services contract is specified by the

- service signature with inputs and outputs,
- pre-conditions,
- post-conditions, and
- quality requirements.

The pre-conditions are those conditions under which the service may be refused without breaking the contract.

The post-conditions are those conditions which must hold once the service has been provided. They apply to the success scenarios of the use case. Each functional requirement directly maps onto a post-condition.

Finally, there may be quality requirements which are specific to this use case. Quality requirements are non-functional requirements referring to the realizable quality of service [BCK_2003_SAIP]. They refer to aspects like scaleability, reliability, performance, integrability, ... and are the core drivers behind architecture and infrastructure. While the pre- and post-conditions are part of the functional requirements which are realized through design, the quality requirements are used to assess whether the target architecture for the use case can indeed host the use case or whether architectural adjustments need to be made in order to realize the required quality requirements.

10.4.2.2.1.1 Bibliography

10.4.2.3 User work flow

The required user work flow is documented via interaction diagrams showing only the messages exchanged between the subject responsible for realizing the use case and the actors.

Figure 10.6: The user work flow for a success scenario of the use case.

For example, Figure 10.6, shows the interactions of the subject with the actors for a particular scenario and specifies the value objects exchanged between them.

10.4.2.4 The services contract

In contract-driven development a services contract is specified by the

- service signature with inputs and outputs,
- class diagrams specifying the data structure requirements for the inputs and outputs,
• pre-conditions,
• post-conditions, and
• quality requirements.

Figure 10.7: The services contract for the process claim service.

10.4.3 The design phase

10.4.3.1 Defining the use case contract

During an URDAD design phase one identifies the responsibilities for the current level of granularity, assigns them to services contracts and specifies the business process the role players realizing the contract need to execute. One then projects out the collaboration context, i.e. the static structure supporting the collaboration which realizes the use case.

The output of the design phase is the technology neutral business process design for that level of granularity. This will include

• the service providers required for the current level of granularity,
• the business process for the current level of granularity,
• the collaboration context which resembles, in a technology neutral way, that subset of the static structure required to support the business process for the current level of granularity, and
• class diagrams for any object exchanged between the role players of the current level of granularity.

10.4.3.2 Responsibility identification and allocation

During the first step of an URDAD design phase one groups functional requirements into responsibility domains and assigns each responsibility domain to a separate services contract. Note that the technology neutral design assigns responsibilities not to concrete implementation classes, but instead to service provider contracts. These contracts can be realized by implementations in different technologies.

In the context of a model driven development process, the choice of a concrete service provider or the technology within which a service provider is to be realized is made during the implementation mapping phase. A services contract can be realized by a system, a system component, an organizational component (e.g. a business unit) or an external service provider to whom the organization has outsourced the responsibility covered by the services contract.
URDAD requires that one adds the responsibility for managing the work flow and assigns the responsibility to a separate services contract. This decouples the service providers from one another, localizes the business process information for the current level of granularity and removes any business process information from the service providers themselves. They are simply there to provide reusable services around a responsibility domain without knowledge of the business processes for which these services are required.

Figure 10.8 shows the responsibility identification and allocation for the process claim use case.

**10.4.3.3 Business process specification**

The services contracts are first introduced abstractly without specifying the services which service providers realizing the services contract need to provide. Instead one next designs the business process for the current level of granularity, showing how these abstract service providers need to collaborate in order to realize the use case. The business process design feeds the services required for the business process into the services contracts for the service providers required for the business process.
10.4.3.3.1 The activity diagram specifying the business process for the processClaim service

Figure 10.9: Activity diagram showing how the controller assembles the business process across services sourced from service providers.

10.4.3.4 Projecting out the collaboration context

The collaboration context shows the service providers required, at a specific level of granularity, to realize the use case, the services they need to provide for this use case and the message paths we require in order for the service providers to be able to collaborate to realize the use case.

Figure 10.10 shows the collaboration context for the process claim use case. Note that the dynamics (i.e. the business process specification) will already have fed in the services required for the business process into the contracts for the individual service providers.

Figure 10.10: The collaboration context for the process claim use case
10.4.4 Transition to next level of granularity

Having completed one analysis/design cycle, one needs to ask oneself whether the business process for the use case has been fully specified or not. If not, one may need to go to lower levels of granularity for some or all of the service providers from the current level of granularity.

**Note**

Often the lower level granularity design is done by different business analysts who understand that domain of responsibility (e.g. from a different department of the organization) or by the business analysts of other organizations to whom the realization of the services contract is outsourced.

In order to execute the transition to the next lower level of granularity, one selects one of the service providers as the new context. The services from the current level of granularity become the lower level use cases. After all, a use case is defined as a service of value[?]. One then selects a particular service or use case and repeats the lower level analysis and design process.

Figure 10.11 shows an example of stake holders around the lower level use case of providing a settlement offer together with their functional requirements around that use case.

![Figure 10.11: Functional requirements for the provide settlement offer service](image)

The lower level design phase is executed in the same way as one was done for the higher level of granularity. It start with the grouping of functional requirements into responsibility domains and the allocation of each responsibility domain to a separate services contract. Figure 10.11 Figure `fig:assessCoverageResponsibilityAllocation` shows an example of identifying and allocating the lower level responsibilities around assessing the policy coverage.
10.4.4.1 Facilitating navigation across levels of granularity

In URDAD a service from one level of granularity is mapped onto a use case at the next lower level of granularity. In order to be able to conveniently navigate across levels of granularity, one needs to maintain the link between the service and its corresponding use case. This can be done in various UML tools by adding a link with an appropriate stereotype.

10.4.4.2 Bibliography


10.5 URDAD views

URDAD defines six standard views (diagrams) which can be used to construct and present an URDAD model. Three views are part of the analysis for the service/use case whilst the other 3 views are part of the The views include

- the *services contract view* specifying the stake holder requirements,
- the *user work flow view* specifying the required interaction between the user and the service provider
- the *functional requirements view* specifying the functionality (lower level services) required to address the stakeholder requirements,
- the *responsibility allocation view* specifying the services contracts to which the functional requirements are assigned to,
- the *process specification view* specifying the (business) process through which the service is realized, and
- the *collaboration context view* specifying the role players (contracts) collaborating in the process realizing the service and the
10.6 How are the design activities realizing the desired design attributes embedded in URDAD?

The single responsibility principle is directly enforced by grouping functional requirements into responsibility domains and requiring the each responsibility domain is assigned to a separate contract.

The levels of granularity are fixed by including only those contracts to which the responsibilities for a particular level of granularity have been assigned. Furthermore, the level of granularity is further fixed by requiring that the lowest level service requests at a particular level of granularity are those which come from the controller for that level of granularity.

The locking into services contracts is enforced by directly assigning responsibility domains to contracts and specifying the work flow across these contracts. The URDAD design process then generates the contract details and requires the specification of pre-and post-conditions as well as quality requirements.

URDAD directly enforces the introduction of a work flow controller for each responsibility domain and each level of granularity, resulting in the localization of the business process information and decoupling of the service providers used in the business process.

The relationship between the layers of granularity are documented through an explicit transition across the layers of granularity, facilitating bidirectional traceability.

Finally, the minimal conceptual (technology neutral) structure supporting the collaboration is projected out from the dynamics of the business process realizing the use case.

10.7 Evaluating an URDAD based design

In order to assess an URDAD based design one will

• validate that each functional requirement is indeed addressed by the business process,
• assess the grouping of functional requirements into responsibility domains in order to verify that there are no overlaps between responsibility domains and that each responsibility domain does indeed comprise a single responsibility,
• verify that the process at any level of granularity is intuitive and simple,
• verify that the service providers are represented by services contracts (UML interfaces) and not by implementation or technology specific classes,
• verify that each services contract has been fully specified including the functional and non-functional requirements,
• verify that the structure of all exchanged value objects is defined using class diagrams.

10.8 The URDAD profile

10.9 URDAD model organization

10.9.1 Why is the model organization important?

If one does not have a well defined strategy for organizing a model, it is very likely that the model will deteriorate very quickly into an unmanageable mess. You need to effectively organize your model such that you are able to

• find elements more rapidly,
• efficiently navigate and work on the model,
• more simply delete and move elements, i.e. moving or deleting elements moves or deletes everything which is only relevant for that element,
• easily modularize the model as it grows.
10.9.2 Guiding principles for model organization

The core guiding principles behind the URDAD model organization are:

- **Responsibility localization** Group laterally into responsibility domains.
- **Layering** Group horizontally in terms of layers of granularity.
- **Cohesion** Package element specific artifacts within the element.

10.9.3 URDAD rules for model organization

1. Use cases at a particular level of granularity are grouped into packages according to their responsibility domains, i.e. financial services into finance, legal services into legal, ...

2. Use cases across levels of granularity are packages in sub-packages.

3. All use case artifacts are packaged within the use case. These include
   - the functional requirements specification,
   - the contract specification for the use case,
   - the user work flow specification,
   - the responsibility allocation,
   - the business process specification, and
   - the collaboration context specification for the use case.

4. The higher level request and response objects which are service specific are packaged with the service.

5. Entities and value objects which are not service specific are packaged within the conceptual domain, irrespective of the use cases/services within which they are used.

6. Constraints which are element specific are packages inside the element for which they were created.
10.9.4 Example of the organization of an URDAD model

Figure 10.13: Example of the organization of an URDAD model for an insurer.
10.10 URDAD documentation generation

10.10.1 Introduction

To generate useful documentation from a general UML model is very difficult due to

- the richness and complexity of the UML language,
- the purpose and semantics of the different model elements is not necessarily known, and
- as there is no predefined structure for the UML model - different UML models can have very different structures.

However, when following the URDAD analysis and design methodology, both documentation and code generation are considerably simpler because

- URDAD enforces the use of a very small subset of the UML,
- the URDAD components have a well defined semantics and
- an URDAD model has a well defined, simple structure which is the same across levels of granularity.

10.10.2 Why documentation generation?

One usually needs to provide documentation for various role players in order to make the information relevant to them accessible. If one does not use documentation generation to generate the documentation from the model, then the documentation must manually be kept synchronized with the information contained in the model.

Generating the documentation from the model

- ensures that the documentation remains synchronized with that of the evolving model, and
- reduces the probability that there are errors, i.e. that the documentation does not reflect the true semantics and content of the model.

Need for version control

Since the model will typically evolve over time one needs support for version control. Most UML tools either

- provide version control through add-on components, or
- provide adapters which integrate with external version control systems.

In this case the generated documentation would not have independent versioning, but would adopt the versioning of the model from which it is generated.

Further benefits of documentation generation include that one can

- extract information for different role players and different purposes, excluding any information which is not required,
- present the information in different ways depending on the background and needs of the information consumers.
10.10.3 Types of documentation

The types of documentation one may find useful may really be very varied and may evolve continuously. Usually different role players and stake holders may find different types of reports useful. Commonly one may want to generate documentation for

- business,
- business analysts,
- architects,
- developers,
- quality assurance,
- service providers,
- operations, and for
- users (e.g. clients).

10.10.3.1 Documentation required by business

Business would typically want to see

- the scope of services offered by the organization or a component of the organization,
- the requirements for a service offered by the organization or a component of the organization,
- the business process for a service offered by the organization,
- a list of service providers used by the organization.

10.10.3.2 Documentation required by business analysts

Business/domain analysts are the ones which perform the requirements analysis and the technology neutral (business) process design constructing the platform independent model (PIM) which is often called the business model in URDAD. They would thus be comfortable working with the model itself. Nevertheless, for them too it is often useful to generate reports rendering specific information extracted from the model. In particular, they would typically require

- documents capturing the requirements specification for a use case including the data structure specification for exchanged value objects, and
- document capturing the technology neutral business process design for a use case.

In particular, they would typically require analysis and design information across levels of granularity.

10.10.3.3 Documentation required by architecture

Architecture requires from an URDAD-based technology neutral (business) model reports for

- the scope of services, and
- use case contracts.
10.10.3.4 Documentation required by developers

Developers require sufficient implementation to perform the implementation mapping. In addition they want to generate unit tests for their implementation. The reports they would typically want would include

- the services contracts for the services they need to implement and test, and
- the technology neutral design for these services.

This information is often required across levels of granularity.

10.10.3.5 Documentation required by quality assurance

Quality assurance is responsible for assessing whether the use case contracts are realized. In particular they need to assess whether both, the functional and non-functional requirements are met. For this they require reports which contain the full contract specifications for use cases which include the functional and quality requirements.

10.10.3.6 Documentation required by service providers

Service providers need to know what the services required of them need to provide. As such they require reports which contain the full services contract specifications for use cases which include the functional and quality requirements.

10.10.3.7 Documentation required by operations

Operations is responsible for executing and overseeing the execution of the business processes. They need reports which contain

- the services contracts for the services they need to provide,
- the technology neutral design for these services.

10.10.3.8 Documentation required by users

Users typically need to know what the services will do for them and what their obligations are when making use of the services. To this end they require reports containing

- the services contracts for the services offered,
- the user work flow specification, and
- the services contracts which contain the user obligations.

10.10.4 Documentation generation approaches

There are two main approaches typically taken to documentation generation. One either

- uses the report generation support for a particular UML tool, or
- generating all documents directly off the tool-independent object model.

10.10.4.1 Using the report generation facilities of a UML tool

The report generation facilities provided by UML tools will be very tool specific. One will thus have to look at the They typically make simple report generation easy, often providing a set of predefined report types.
10.10.4.1.1 Advantages of UML tool specific report generation

- Usually it is very easy to define a simple report.
- Often there are a number of simple report templates which one can modify according to one’s needs.
- One has access to the drawn diagrams for inclusion in the report.

**Note**
This is actually often not an advantage but a disadvantage. For example, there is no guarantee that the functional requirements diagram drawn by a business analyst contains all functional requirements and other elements defined in the model or that it does not contain elements that should not be shown in that view (e.g., including functional requirement from a lower level of granularity). When following a solid methodology like URDAD with a well defined PIM structure (i.e., a well defined structure for the technology neutral business model), then one can generate the diagrams from the model.

10.10.4.1.2 Disadvantages of UML tool specific report generation

- The report generation is tied to a tool and will have to be redeveloped when changing tools.
- Often the report generation is quite limited with, at times, very limited access to the full object model.
- The tools used often do not scale well with complexity of the report, i.e., the maintenance costs for the report generation templates and tools may become very high.

10.10.4.2 Generating reports directly off the object model

One can generate reports directly from the object model usually stored in XMI. Doing this decouples the report generation from the actual UML tool used.

10.10.4.2.1 Advantages of generating reports directly from the object model

- One can change UML tools without having to redevelop or modify the documentation generation templates and tools.
- One will have to generate the diagrams form the model, but doing this ensures that the diagrams are complete and correct, i.e., that they have all relevant information from the model and that they do not include elements which should not be contained in the diagram.
- One is not limited to the functionality provided by a particular tool.
- One can typically generate a wider variety of reports and render them in a wider variety of rendering technologies.
- The complexity of the documentation generation templates and tools is typically more manageable.

10.10.4.2.2 Using APIs into the object model

There are a number of APIs like Eclipse EMF framework which provide access to the object model. These can be used in conjunction with other text generation frameworks like Apache Velocity to generate reports off the UML model.

10.10.4.2.3 Generating reports using QVT

The QVT is OMG’s standard language and tools framework for performing model transformations. Report generation can be seen as a form of model transformation similar to code generation. While the inputs for code generation will be the technology neutral model (the PIM) and the platform description model (the PDM), the inputs for documentation or report generation will be the technology neutral (business) process design (the PIM) and the report specification.
10.10.5 Documentation generation in MagicDraw

MagicDraw has quite powerful report generation facilities which enable one to generate a wide variety of reports in a wide variety of formats. By default the report generation generates reports in RTF, but one can relatively easily define one’s own script which generates a report in a different format.

10.10.5.1 Default URDAD service documentation

The current URDAD default report is available in the form of open source apache velocity scripts executed by MagicDraw within the standard MagicDraw report wizards. It makes extensive use of MagicDraw’s OpenAPI to interrogate the MagicDraw object model in order to

- generate a services report off an URDAD model, and
- perform some basic URDAD model validation, checking for correctness and completeness like
  - that the user role for each use case has been specified,
  - that each functional requirement is required,
  - that each functional requirement has been assigned to a services contract,
  - that the data structures for the inputs and outputs of services have been defined, ...

Outlook

It is envisaged that

- that the URDAD documentation generation will move towards directly extracting the information from an XMI based object model,
- the QVT framework will be used to perform the model transformation for the documentation generation,
- that the diagrams will be generated from the model and no longer requested from an UML tool - this ensures the integrity of the diagrams and also enables the generation of diagrams which were never drawn (e.g. the collaboration context),
- that the model validation will be done via OCL based validation suites which will be used by the documentation generation in order to report model deficiencies.

10.10.5.1.1 Output format

The output format for the current URDAD reports is OASIS docbook, an open specification for XML based documentation. This is an open format which, being XML, is easy to work with. Furthermore, it is easy to validate that the document structure is correct.

From here one can render to a variety of formats like PDF, LaTeX or even ODF, the opendocument format. Alternatively one can chain the commands to render directly into PDF.

10.10.5.1.2 Obtaining the default URDAD report templates

The URDAD report templates are open sourced and can be downloaded from http://sourceforge.net/projects/urdaddoctempl/.

10.10.5.1.3 Running the default URDAD report

You will first have to register a new report template with MagicDraw before you can generate reports.
10.10.5.1.3.1 Registering the URDAD default report template

2. In the dialog box
   - Give the report a name, e.g. Default URDAD use case report, and
   - assign it to a category (here you could type in URDAD as a new category for the various URDAD reports).

10.10.5.1.3.2 Generating the report against a use case from your project

To run the URDAD report from within MagicDraw you

2. Select default for the report data and press next.
3. In the user defined fields add the targetUseCaseQualifiedName variable and assign its value to the fully qualified name of the use case for which you would like to run the report, e.g. insurer::claims::Claims::processClaim and press next.
4. In the select element scope dialog, select the entire model via add all (the report generator should be able to extract whatever it needs from the model as a whole) and press next.
5. In the final dialog, enter the file name you would like to have for the report (e.g. useCaseName.docbook) and press generate to generate the report.

10.10.5.1.4 Use case: processClaim

URDAD based Analysis and Design, February 21, 2009

10.10.5.1.4.1 Introduction

This is an URDAD compliant document for the requirements analysis and business process design of the processClaim use case. The document shows the requirements and design for this level of granularity. The details of the services used within the business process are shown in the analysis and design of the respective use cases for these lower level services.

This use case is offered to policy holders enabling them to request the processing of a claim against a policy. The fully qualified name of the use case is insurer::claims::Claims::processClaim.

10.10.5.1.4.2 Analysis

The analysis section specifies the stakeholder requirements, the user work flow, the data structures for the exchanged value objects and the service contract with the pre- and post-conditions and quality requirements. It contains the requirements specification for the current level of granularity.

10.10.5.1.4.3 Functional requirements: processClaim

The functional requirements diagram shown in Figure 10.14 shows the processClaim use case, the stake holders who have an interest in that use case and their functional requirements.

Figure 10.14: The stakeholder requirements diagram for the processClaim use case.
10.10.5.1.4.4 The user role

The user role is the role played by those objects which make use of the use case. It is represented by an interface/contract which accumulates the contractual obligations of the user itself.

- **PolicyHolder** Objects who play the role of a PolicyHolder make use of the processClaim use case. The role played by a party which owns an insurance policy.

10.10.5.1.4.5 Stake holders

The following stake holders have an interest in the processClaim use case:

- **InsuranceRegulator** The role played by a party which regulates the insurance sector.
- **PolicyHolder** The role played by a party which owns an insurance policy.
- **Shareholder** The role played by an investor in the insurance company.

10.10.5.1.4.6 Mandatory functional requirements

The following functional requirements need to be addressed for any success scenario of the use case, i.e. for any scenario where the user obtains the value from the use case.

- **acceptSettlementOffer** Accept the settlement offer in a legally binding way. This is required by
  - Shareholder
- **provideSettlementOffer** Provide a settlement offer for a claim. This is required by
  - InsuranceRegulator
  - PolicyHolder
- **settleClaim** Perform any financial and other transactions required to settle a claim according to a provided settlement offer. This is required by
  - PolicyHolder
- **providePolicyStatus** Determine and provide the current state of a policy. This is required by
  - Shareholder
- **update claims history** The claims history must be updated with any settled claim. This is required by
  - Shareholder
- **recuperateLosses** Perform the legal processes against accountable parties which recuperate the losses or part thereof incurred due to an insurance claim.

**ERROR: Missing stakeholder specification**

URDAD requires that each functional requirement is linked to the stakeholder(s) who require it via a requires relationship. The recuperateLosses requirement is not linked to a stakeholder via a requires relationship.

10.10.5.1.4.7 Conditional functional requirements

The following functional requirements need to be addressed only under certain conditions:

- **performFraudInvestigation** Perform a routine fraud investigation in order to determine whether there is a reasonable likelihood of fraud. This functional requirement needs to be addressed if \( \text{settlementAmount} > \text{fraudInvestigationThreshold} \). This is required by
  - Shareholder
10.10.5.1.4.8 User work flow: processClaim

The user work flow diagram shown in Figure 10.15 specifies how users interact with the service provider in the context of making use of the processClaim use case. It shows the messages exchanged in the various scenarios.

Policy holders request the processing of a claim by submitting the claim to the insurer. If the policy against which the claim is made is inactive, the policy holder is informed of this and the claims processing is aborted. Otherwise the insurer does some pre-processing of the claim resulting in a settlement offer which the policy holder is requested to accept. If the policy holder accepts the settlement offer, the insurer completes the processing of the claim and provides a claim settlement report to the policy holder. Otherwise the insurer confirms with the policy holder that the settlement offer was rejected.

Figure 10.15: The user work flow for the processClaim use case.

10.10.5.1.4.9 Service request specifications

This section specifies the services requested in the user work flow including the data structures for the inputs and outputs (i.e. requests and responses) for each service.

10.10.5.1.4.10 Service: acceptSettlementOffer

Policy holders need to accept settlement offers before claims are settled. This is the service through which a policy holder is requested to accept a settlement offer.

10.10.5.1.4.11 Input (request object): SettlementOffer

The request object for the acceptSettlementOffer service is a SettlementOffer. It contains the information which must be provided with the service request.

ERROR: missing data structure specification
The data structure requirements for each exchanged value object needs to be specified. The data structure of SettlementOffer has not been specified.

10.10.5.1.4.12 Output (response object): SettlementOfferAcceptance

ERROR: missing data structure specification
The data structure requirements for each exchanged value object needs to be specified. The data structure of SettlementOfferAcceptance has not been specified.

10.10.5.1.4.13 Service: processClaim

This is the service for processing a claim against an insurance policy.

10.10.5.1.4.14 Input (request object): Claim

The request object for the processClaim service is a Claim. It contains the information which must be provided with the service request.

The claim contains information required for the processing of a claim including information about the claimant, the policy against which the claims is made, the claim items as well as any supporting information.

Figure 10.16: Data structure (class) diagram for Claim
10.10.5.1.4.15 **Output (response object): ClaimSettlementReport**

The claim settlement report contains information of how the settlement amounts for each item as well as potentially information pertaining to the calculation of the settlement amount and information on how the settlement was made, e.g., information about the financial transaction for the settlement.

Figure 10.17: Data structure (class) diagram for ClaimSettlementReport

10.10.5.1.4.16 **Service contract: processClaim**

The service contract specification diagram shown in Figure 10.18 specifies the contract (interface) name, the service with the request and response objects, the pre- and post-conditions and the quality requirements for the service.

The pre-conditions are those conditions under which the service may be refused without being in breach of the service contract. The post-conditions are those conditions which must hold true once the service has been rendered. The quality requirements are the non-functional requirements.

The process claim service receives as input a Claim and returns, upon successful completion, a claim settlement report. The claim processing is aborted if either the policy is not active or the policy holder does not accept the settlement offer. Otherwise the claim processing is completed. On completion of the claims processing, the policy holder will have accepted the settlement offer, the claim will have been settled and the loss recuperation process will have been started.

Figure 10.18: The service contract for the processClaim use case.

10.10.5.1.4.17 **Technology neutral process design**

This section specifies the technology neutral design realizing the use case requirements. In particular, it specifies which functional requirement is assigned to which services contract, how the business process is assembled from these services and the collaboration context containing the services required from the various service providers together with the inputs and outputs for each service and the message paths to the service providers.

10.10.5.1.4.18 **Responsibility identification and allocation: processClaim**

Figure 10.19 shows the grouping of functional requirements into responsibility domains. Each responsibility domain is assigned to a separate services contract.

The responsibility of settling the claim is assigned to Finance. The claims process is managed by Policies is responsible for providing a settlement offer for the claim and for providing the status of a policy. The policy holder is required to accept the settlement offer while legal services is responsible for recuperating losses and performing fraud investigations.

Figure 10.19: The responsibility allocation diagram for the processClaim use case.

10.10.5.1.4.19 **Controller**

The controller takes ownership of the business process and manages the business process for the use case. The controller is responsible for assembling the business process across lower level services sourced from service providers realizing lower level services contracts. All workflow decisions and control logic are executed by the controller. As such the controller also decouples the lower level service providers from one another, i.e., the lower level service providers have no knowledge of either the business process within which the services are used or any other service providers participating in the business process.

- The controller specifying the business process for the processClaim is class com.nomagic.uml2.impl.magicdraw.classes.mdkernel.Pack.
  - It realizes the use case via the processClaim service.
10.10.5.1.4.20 Services contracts for the required responsibility domains

The functional requirements for the processClaim have been allocated to the following responsibility domains:

- **Finance** Provide financial services. The following functional requirements have been assigned to the Finance services contract:
  - **settleClaim** Perform any financial and other transactions required to settle a claim according to a provided settlement offer.

- **Policies** Manage policies. The following functional requirements have been assigned to the Policies services contract:
  - **provideSettlementOffer** Provide a settlement offer for a claim.
  - **provideSettlementOffer** Provide a settlement offer for a claim.
  - **providePolicyStatus** Determine and provide the current state of a policy.
  - **update claims history** The claims history must be updated with any settled claim.

- **PolicyHolder** The role played by a party which owns an insurance policy. The following functional requirements have been assigned to the PolicyHolder services contract:
  - **acceptSettlementOffer** Accept the settlement offer in a legally binding way.

- **Legal** Provide legal services. The following functional requirements have been assigned to the Legal services contract:
  - **performFraudInvestigation** Perform a routine fraud investigation in order to determine whether there is a reasonable likelihood of fraud.
  - **recuperateLosses** Perform the legal processes against accountable parties which recuperate the losses or part thereof incurred due to an insurance claim.

10.10.5.1.4.21 Process specification: processClaim

Figure 10.20 shows how the processClaim service is assembled from services sourced from the service providers to whom the functional requirements have been assigned.

This is the business process for the processClaim service as offered by Claims. Claims receives a claim and first requests policies to provide the policy status for the policy against which the claim is made. If the policy is inactive, Claims terminates the claims processing process by throwing a PolicyInactiveException. If the policy is active, Claims requests policies to provide a settlement offer for the claim. Having received the settlement offer, Claims checks whether the settlement amount exceeds the fraud investigation threshold or not. If it does, legal services is requested to launch a routine fraud investigation for the claim. In either case, the policy holder is requested to accept the settlement offer. If the policy holder rejects the settlement offer, the business process is aborted, notifying the client via a SettlementOfferRejectedException. Otherwise, legal services is requested to start the process of recuperating the losses for the claim while finance is requested to settle the claim and a ClaimSettlementReport is provided to the policy holder.

Figure 10.20: The business process specification diagram for the processClaim use case.

10.10.5.1.4.22 Collaboration context: processClaim

Figure 10.21 shows the services required for the processClaim use case from the different service providers as well as the required message paths through which the services can be requested.

In order to process a claim, Claims requires from Policies the services to provide a settlement offer for the claim and to provide the policy status. The policy holder needs to accept the settlement offer, Finance needs to provide the claim settlement service and Legal needs to perform fraud investigations and recuperate losses for a claim.

Figure 10.21: The collaboration context diagram for the processClaim use case.
10.10.5.1.5 The template for the default URDAD service report

The current URDAD documentation generation makes use of Apache Velocity to generate a docbook document.

10.10.5.1.5.1 urdadStandardReport.txt

This is the main file which is modularized in order to make the report template more manageable. It defines

- variables for the URDAD stereotypes,
- variables for UML/MagicDraw concepts, and
- a range of macros which extract information from the UML model and obtains diagrams from MagicDraw. Examples include
  - obtain all functional requirements for a use case,
  - to obtain the inputs and outputs for a service,...

Finally it imports the modules for the templates defining the introduction, the analysis section and the design section of the generated document.

```xml
<?xml version="1.0" encoding="utf-8"?>
<oxygen RNGSchema="http://www.oasis-open.org/docbook/xml/5.0b5/rng/docbookxi.rng" type="xml">
  <!--Authored by mailto:fritz@solms.co.za on 2007-06-01 -->
  #*
  This is the Open Document Standard template for the default docbook based URDAD report.

  Notes:
  ----- 
  - the $targetUseCaseQualifiedName must be provided as a user-defined variable
  - the $documentType variable which specifies the type of document to be generated. Currently this will be one of
    * standard
    * minimal
    * contract

  TODOs:
  ------
  - actors of use case diagrams
  *
  ##------------------------------------------------
  ## URDAD stereotypes
  ##------------------------------------------------
  #set($urdad_functionalRequirementsDiagram = "functionalRequirementsDiagram")
  #set($urdad_userWorkflowDiagram = "userWorkflowDiagram")
  #set($urdad_serviceContractDiagram = "serviceContractDiagram")
  #set($urdad_responsibilityAllocationDiagram = "responsibilityAllocationDiagram")
  #set($urdad_businessProcessDiagram = "businessProcessDiagram")
  #set($urdad_collaborationContextDiagram = "collaborationContextDiagram")
  #set($urdad_requires = "requires")
  #set($urdad_valueObject = "valueObject")
  #set($urdad_raiseException = "raiseException")
  #------------------------------------------------
  #------------------------------------------------
  ## UML concepts
  #------------------------------------------------
  #STRUCTURE
  #set ($umlInterfaceClass = "com.nomagic.uml2.impl.magicdraw.classes.mdinterfaces. InterfaceImpl")
```
Technology-Neutral Analysis and Design using UML and URDAD
#macro ( getConditionalFunctionalRequirements $useCase)
#set ($result = $array.createHashSet())
#foreach ($uc in $useCase)
#foreach ($extendRelationship in $uc.extend)
#if ($report.getSupplierElement($extendRelationship) == $useCase)
#set ($dummy = $result.add($report.getClientElement($extendRelationship)))
#end
#end
#end
##-------------------------------------------------

#macro ( getFunctionalRequirements $useCase)
#set ($functionalRequirements = $array.createHashSet())
#getMandatoryFunctionalRequirements($targetUseCase)
#set ($dummy = $functionalRequirements.addAll($result))
#getConditionalFunctionalRequirements($targetUseCase)
#set ($dummy = $functionalRequirements.addAll($result))
#set ($result = $functionalRequirements)
#end
##-------------------------------------------------

#macro ( getStakeHolders $functionalRequirement)
#set ($result = $array.createHashSet())
#foreach ($relationship in $functionalRequirement.get_relationshipOfRelatedElement())
#if ($report.containsStereotype($relationship,$urdad.requires))
#set($dummy = $result.add($report.getClientElement($relationship)))
#end
#end
##-------------------------------------------------

#macro ( getStakeholdersForUseCase $useCase)
#getMandatoryFunctionalRequirements($useCase)
#set($mandatoryFunctionalRequirements = $result)
#getConditionalFunctionalRequirements($useCase)
#set($conditionalFunctionalRequirements = $result)
#set ($result = $array.createHashSet())
#foreach ($functionalRequirement in $mandatoryFunctionalRequirements)
#foreach ($relationship in $functionalRequirement.get_relationshipOfRelatedElement())
#if ($report.containsStereotype($relationship,$urdad.requires))
#set($dummy = $result.add($report.getClientElement($relationship)))
#end
#end
#end
#foreach ($functionalRequirement in $conditionalFunctionalRequirements)
#foreach ($relationship in $functionalRequirement.get_relationshipOfRelatedElement())
#if ($report.containsStereotype($relationship,$urdad.requires))
#set($dummy = $result.add($report.getClientElement($relationship)))
#end
#end
#end
##-------------------------------------------------

#macro ( getRequestedServices $interaction)
#set ($result = $array.createHashSet())
#set ($exchMsgs = $interaction.message)
#foreach ($msg in $exchMsgs)
#set($op = $msg.getReceiveEvent().getEvent().getOperation())
#if ($op)
#set($dummy = $result.add($op))
#end
#end
##-------------------------------------------------
##
#macro (getInputParameters $service)
#set ($result = $array.createHashSet())
#foreach ($par in $service.getOwnedParameter())
#if ($par.getDirection() == 'in')
#set($dummy = $result.add($par))
#end
#end
##-------------------------------------------------
##
#macro (getDataStructureDiagram $class)
#set($result = '')
#foreach ($diagram in $Diagram)
#if($diagram.getQualifiedName() == $class.getQualifiedName())
#set ($result = $diagram)
#end
#end
##-------------------------------------------------
##
#macro (getRelatedElements $element, $relationshipClass)
#set ($result = $array.createHashSet())
#foreach ($relationship in $element.get_relationshipOfRelatedElement())
#if($relationship.getClass().getName() == $relationshipClass)
#set($dummy = $result.add($report.getClientElement($relationship)))
#end
#end
##-------------------------------------------------
##
#macro (getRelationshipsFor $element, $relationshipClass)
#set ($result = $array.createHashSet())
#foreach ($relationship in $element.get_relationshipOfRelatedElement())
#if($relationship.getClass().getName() == $relationshipClass)
#set($dummy = $result.add($relationship))
#end
#end
##-------------------------------------------------
##
#macro (getRealizingServices, $useCase)
#getRelationshipsFor($targetUseCase, $umlRealizationRelationship)
#set($umlRealizationRelationships = $result)
#set ($result = $array.createHashSet())

10.10.5.1.5.2 introduction.txt

<section>
<title>Introduction</title>
<para>
This is an URDAD compliant document for the requirements analysis and business design of the use case. The document shows the requirements and design for this level of granularity. The details of the services used within the business process are shown in the analysis and design of the respective use cases for these lower level services.
</para>
<para>
$report.getComment($targetUseCase).body
The fully qualified name of the use case is $targetUseCase.qualifiedName.
10.10.5.1.5.3 Analysis.txt

The analysis section specifies the stakeholder requirements, the user workflow, the data structures for the exchanged value objects and the service contract with the pre- and post-conditions and quality requirements. It contains the requirements specification for the current level of granularity.

#parse("analysis/functionalRequirements/functionalRequirements.txt")
#parse("analysis/userWorkflow/userWorkflow.txt")
#parse("analysis/servicesContract/servicesContract.txt")

10.10.5.1.5.4 FunctionalRequirements.txt

<!-- FUNCTIONAL REQUIREMENTS SPECIFICATION -->
<!-- ********************************************* -->

<title>Functional requirements: $targetUseCase.name</title>
#getDiagramForUseCase($targetUseCase,$urdad_functionalRequirementsDiagram)
#set($diagram = $result)
#if ($diagram && $diagram != '')
#set($diagramId = $diagram.qualifiedName.replaceAll('::','.').replaceAll(' ','_'))
<para>
The functional requirements diagram shown below in <xref linkend="$diagramId"/>
shows the $targetUseCase.name use case, the stakeholders who have an interest in that use case and their functional requirements.
</para>
#set ($diagramComment = $report.getComment($diagram).body)
#if ($diagramComment && $diagramComment != '')
<para>$diagramComment</para>
#end
<figure xml:id="$diagramId">
<title>The stakeholder requirements diagram for the $targetUseCase.name use case.</title>
<mediaobject><imageobject>
<imagedata contentwidth="150mm" scalefit="1" fileref="$diagram.image"/>
</imageobject></mediaobject>
</figure>
#parse("analysis/functionalRequirements/user.txt")
Technology-Neutral Analysis and Design using UML and URDAD

#parse("analysis/functionalRequirements/stakeholders.txt")

#getMandatoryFunctionalRequirements($targetUseCase)
#set($mandatoryFunctionalRequirements = $result)
#getConditionalFunctionalRequirements($targetUseCase)
#set($conditionalFunctionalRequirements = $result)
#if (($mandatoryFunctionalRequirements.size() == 0) && ($conditionalFunctionalRequirements.size() == 0))
    <note>
        <title>ERROR: No functional requirements specified for use case</title>
        <para>
            URDAD requires that there must be at least one functional requirement for each use case. No functional requirements have been specified for the $targetUseCase.name use case.
        </para>
    </note>
#else
    #if ($mandatoryFunctionalRequirements.size() > 0)
        #parse("analysis/functionalRequirements/mandatoryFunctionalRequirements.txt")
    #end
    #if ($conditionalFunctionalRequirements.size() > 0)
        #parse("analysis/functionalRequirements/conditionalFunctionalRequirements.txt")
    #end
#else
    <note>
        <title>ERROR: Missing functional requirements diagram</title>
        <para>
            The URDAD process requires that the stakeholders and their functional requirements are specified
        </para>
    </note>
#end
</section>

10.10.5.1.5.5 user.txt

#getRelationshipsFor($targetUseCase, $umlAssociationRelationship)
#set($umlAssociationRelationships = $result)

<section>
    <title>The user role</title>
    <para>
        The user role is the role played by those objects which make use of the use case. It is represented by an interface/contract which accumulates the contractual obligations of the user itself.
    </para>
    <para>
        #if ($umlAssociationRelationships.size() == 0)
            <note>
                <title>ERROR: No user role specified for $targetUseCase.name use case</title>
                <para>
                    URDAD requires that a single user role is assigned to each use case. The user role needs to be represented by an interface and there should be an association relationship between the use case and interface/contract
                </para>
            </note>
        #end
    </para>
</section>
representing the user role.
</para>
</note>
#else
</itemizedlist>
#if ($umlAssociationRelationships.size() > 1)
<note>
<title>WARNING: Multiple user roles specified for $targetUseCase.name use case</title>
<para>
URDAD requires that a single user role is assigned to each use case. The user role needs to be represented by an interface and there should be an association relationship between the use case and interface/contract representing the user role.
</para>
</note>
#endif
#set ($userRole = '')
#foreach ($umlAssociationRelationship in $umlAssociationRelationships)
#if ($report.getClientElement($umlAssociationRelationship).getClass().getName().equals($umlInterfaceClass))
#set ($userRole = $report.getClientElement($umlAssociationRelationship))
#else
#if ($report.getSupplierElement($umlAssociationRelationship).getName().equals($umlInterfaceClass))
#set ($userRole = $report.getSupplierElement($umlAssociationRelationship))
#else
<note>
<title>ERROR: user not represented by an interface</title>
<para>
URDAD requires that the user role is represented by an interface which will accumulate the user responsibilities for the use case.
</para>
</note>
#endif
#endif
#if ($userRole != '')
<li><para>
<title>$userRole.name</title>
<para>
Objects who play the role of a $userRole.name make use of the $targetUseCase.name use case.
</para>
#set ($userRoleComment = $report.getComment($userRole).body)
#if ($userRoleComment && $userRoleComment != '')
$userRoleComment
#endif
</para>
</li>
#endif
</itemizedlist>
</section>
10.10.5.1.6 stakeholders.txt

```
#getStakeholdersForUseCase($targetUseCase)
#set($stakeholders = $result)
#if ($stakeholders.size() > 0)
<section>
<title>Stakeholders</title>
<para>
The following stakeholders have an interest in the $targetUseCase.name use case:
<itemizedlist>
#foreach($stakeholder in $stakeholders)
## Enforce that stakeholder represented by interface
#if ($stakeholder.getClass().getName() != $umlInterfaceClass)
<note>
<title>ERROR: Stakeholder not represented by contract/interface</title>
<para>
URDAD requires that each stakeholder be represented by a contract (interface). $stakeholder.name is not represented by an interface.
</para>
</note>
#end
</itemizedlist>
</para>
</section>
#end
```

10.10.5.1.7 conditionalFunctionalRequirements.txt

```
<section>
<title>Conditional functional requirements</title>
<para>
The following functional requirements need to be addressed only under certain conditions:
<itemizedlist>
#foreach($functionalRequirement in $conditionalFunctionalRequirements)
</itemizedlist>
$xmloutputElementgetValue($functionalRequirement.name)</xmloutputElement>
<para>
$report.getComment($functionalRequirement).body
</para>
</section>
```

This functional requirement needs to be addressed if
#foreach ($extendRelationship in $functionalRequirement.extend)
#if ($report.getSupplierElement($extendRelationship) == $targetUseCase)
<li>$extendRelationship.condition.specification.body</li>
#end
#end
#getStakeHolders($functionalRequirement)
#set($stakeholders = $result)
#if ($stakeholders.size() > 0)
This is required by
```
10.10.5.1.5.8 mandatoryFunctionalRequirements.txt

The following functional requirements need to be addressed for any success scenario of the use case, i.e. for any scenario where the user obtains the value from the use case.

#foreach($functionalRequirement in $mandatoryFunctionalRequirements)
  <listitem><formalpara>$functionalRequirement.name</formalpara>
  <para>$report.getComment($functionalRequirement).body
    #getStakeHolders($functionalRequirement)
    #set($stakeHolders = $result)
    #if ($stakeHolders.size() > 0)
      This is required by
      #foreach ($sh in $stakeHolders)
          <listitem><para>$sh.name</para></listitem>
      #end
    #else
      <note><title>ERROR: Missing stakeholder specification</title><para>URDAD requires that each functional requirement is linked to the stakeholder(s) who require it via a requires relationship. The $functionalRequirement.name requirement is not linked to a stakeholder via a requires relationship.</para></note>
    #end
  </para>
#end
10.10.5.1.5.9 userWorkflow.txt

<!-- USER WORKFLOW SPECIFICATION -->
<!-- ********************************* -->
#getDiagramForUseCase($targetUseCase,$urdad_userWorkflowDiagram)
#set($diagram = $result)
#if($diagram && ($diagram != "")
#set($diagramId = $diagram.qualifiedName.replaceAll('::','.').replaceAll(' ','_'))

<section>
<title>User work flow: $targetUseCase.name</title>
<para>
The user work flow diagram shown in
<xref linkend="$diagramId"/>
specifies how users interact with the service provider in the context of
making use of the $targetUseCase.name use case. It shows the messages exchanged
in the various scenarios.
</para>
#set($diagramComment = $report.getComment($diagram).body)
#if ($diagramComment && ($diagramComment != ""))
<para>
$diagramComment
</para>
#end
<figure xml:id="$diagramId">
<title>The user work flow for the $targetUseCase.name use case.</title>
<mediaobject><imageobject>
<imagedata contentwidth="150mm" scalefit="1" fileref="$diagram.image"/>
</imageobject></mediaobject>
</figure>
#parse("analysis/userWorkflow/services/services.txt")

</section> ## end of user workflow section
#
#else
<note>
<para>
The user work flow was not specified.
</para>
</note>
#*
#end

10.10.5.1.5.10 services.txt

#getDiagramForUseCase($targetUseCase,$urdad_userWorkflowDiagram ←
This section specifies the services requested in the user workflow including the data structures for the inputs and outputs (i.e. requests and responses) for each service.

The pre-conditions are those conditions under which the service may be refused without being in breach of services contract. The post-conditions are those conditions which must hold true once the service has been rendered. The quality requirements are the non-functional requirements.
#getRealizingServices($targetUseCase)
#set($services = $result)
#foreach($service in $services)
    <para>
The service realizing the services contract is $service.getName().
    #set($serviceComment = $report.getComment($service).body)
    #if ($serviceComment && $serviceComment != '')
        $serviceComment
    #end
    </para>
#end
#parse("analysis/userWorkflow/services/service/parameters/parameters.txt")
#parse("analysis/userWorkflow/services/service/returnValue/returnValue.txt")
#end
#else
    <section>
        <title>ERROR: Missing service contract specification</title>
        <para>
            URDAD requires the specification of a services contract showing the contract (← interface) with the services, their inputs and outputs and their quality requirements.
        </para>
    </section>
#end

10.10.5.1.5.12  design.txt

    <section>
        <title>Technology neutral process design</title>
        <para>
            This section specifies the technology neutral design realizing the use case ← requirements. In particular, it specifies which functional requirement is assigned to which services contract, how the business process is assembled from ← these services and the collaboration context containing the services required from the various service providers together with the inputs ← and outputs for each service and the message paths to the service providers.
        </para>
        #parse("design/responsibilityAllocation/responsibilityAllocation.txt")
        #parse("design/businessProcessSpecification/businessProcessSpecification.txt")
        #parse("design/collaborationContext/collaborationContext.txt")
    </section>

10.10.5.1.5.13  responsibilityAllocation.txt

    <!-- RESPONSIBILITY ALLOCATION -->
    <!-- ---------------------------------- -->
    #getDiagramForUseCase($targetUseCase,$urdad_responsibilityAllocationDiagram)
Responsibility identification and allocation: $targetUseCase.name

The responsibility allocation diagram for the $targetUseCase.name use case.

ERROR: missing responsibility allocation diagram

--- BUSINESS PROCESS SPECIFICATION ---

# TODO:
# - Actually should get behaviour diagram assigned to service realizing use case
# - Lots of validation and information extraction

#parse("design/responsibilityAllocation/controller/controller.txt")
#parse("design/responsibilityAllocation/serviceProviderContracts/ serviceProviderContracts.txt")
Process specification: $targetUseCase.name

<xref linkend="$diagramId"/>

shows how the $targetUseCase.name service is assembled from services sourced from the ← service providers to whom the functional requirements have been assigned.

#set ($diagramComment = $report.getComment($diagram).body)
#if ($diagramComment && $diagramComment != '')
<para>
$diagramComment
</para>
#end
<figure xml:id="$diagramId">
<title>The business process specification diagram for the $targetUseCase.name ← use case.</title>
<mediaobject>
<imageobject>
<imagedata contentwidth="150mm" scalefit="1" fileref="$diagram.image"/>
</imageobject>
</mediaobject>
</figure>

## %%%%%%%%%% #parse("design/businessProcessSpecification/englishMapping/englishMapping. ← txt")
#else
<section>
<title>ERROR: Missing process specification</title>
<para>
URDAD requires that a (business) process is specified for the use case and that ← the process is assigned as the behaviour for the service realizing the use case.
</para>
</section>
#end

### 10.10.5.1.5.15 responsibilityAllocation.txt
10.11 Implementation mappings

The implementation mappings to a particular SOA technology suite (such as a JBI-based EJB) would be quite technology-specific. Thus, while the technology neutral business process design is usually done by business analysts, the implementation mappings are usually done by the technical team, with guidance from the architect.

Often a business process is realised across a combination of manual (human) work flow steps, services provided by external service providers and automated processing steps executed within systems. The services contracts coming out of the technology neutral business process design can be used as a basis for the service provider contracts which are either realised by external service providers or by business units hosted within the organisation. The implementation mapping of such work flow steps may require training certain staff members to execute them.

Often, however, the technology mapping may result in mapping the technology neutral design onto

- a realisation using current systems, with perhaps some additional development,
- buying technology components and customising them, or
- developing an entire system hosting the various services from scratch.

MDA (Model-Driven Architecture) tools aim to automate this process fully in the near future.

10.11.1 Notes on mapping onto a Service-Oriented architecture (SOA)

10.11.1.1 Services Contracts

Services contracts are mapped to WSDL contracts which are typically abstract, i.e. without protocol binding definitions. The structure of the exchanged value objects are mapped to XML Schema definitions.
10.11.2 Workflow controllers

The service which plays the role of workflow controller at a certain level of granularity typically needs to invoke other, lower-level services. For this, we typically map it to a technology which could be used for service orchestration, such as WS-BPEL or a Java-based service.

The design for a workflow controller often involves dynamically routing requests to lower-level service providers based on certain criteria (message contents, context, environmental state). This implies a content-based router, often implemented in a rules or workflow technology such as Apache Camel or Drools.

Note

WS-BPEL has practical shortcomings in terms of data manipulation / querying when dealing with object-oriented XML structures, which render it practical only for relatively simple, course-grained and high-level activities, forcing the developer to outsource logic to a greater number of low-level services (implemented in, say, Java or XSLT) than what may have been otherwise necessary.

10.11.3 Service Adaptors

An URDAD-based design often includes, or implies, adaptors between services. These adaptors often primarily involve message transformation, to which a technology such as XSLT is very suited. If an adaptor contains a large amount of logic, or requires multiple helper services, it could be implemented as a special form of workflow controller.

10.11.4 Low-level business services

Individual services that perform atomic tasks (such as computation, database persistence, etc) are often implemented in a technology which has access to the necessary support technologies, such as Java (and often in a container such as EJB).

10.12 Summary

The set of accepted design principles which are seen as required characteristics of a good design can be supported by a set of design activities through which these design principles are realized. URDAD defines an algorithmic design process which incorporates these design activities. It generates a technology neutral business process design in the form of services contracts for each level of granularity together with the business process for that level of granularity. URDAD can be embedded within a model driven development process where the technology neutral business process design is ultimately mapped onto one’s choice of implementation architecture and technologies.
Chapter 11

Index

A
access levels, 42
action, 112
activity, 112
diagram, 13
activity diagram
object flow, 116
actor
definition of, 20
primary, see user
secondary
observer, 20
service provider, 20
stereotype icon for, 20
aggregation, 82
difference to composition, 82
state, 82
algorithm, 3
architecture
census design, 136
frequency of change, 3
of organization, 1
seen as high level design, 3
versus design, 3
association, 68
cardinality constraint, 69
decouple from service provider, 70
for client server relationship, 69
for client server relationships, 68
for navigatability, 68
in use case diagram, 18
label, 69
message path, 70
peer-to-peer, 71
role name, 69
UML notation for, 68
asynchronous message, see message
attribute, 33
collection, 33
derived, 34

B
business analysis

sub-disciplines, 1
business process
assessment, 2
design, 2
for level of granularity, 150
localization, 150
remove from service providers, 150
specification, 150
validation, 2

C
class, 30
definition of, 8
diagram
attributes, 33
versus interface and object, 52
class diagram, 13
client-server, see association
cohesion, see single responsibility principle
collaboration, 115
collaboration context, 149, 151
communication
diagram, 13
communication diagram, 14
component
definition of, 9
in composition relationship, 77
component diagram, 14
composite structure diagram, 14
composition, 77
encapsulation, 78
limited life span, 78
responsibility, 78
concurrency
activity diagram, 119
conditional, 22
decision node, 114
use case
extends relationship, 23
constraint
multiplicity, 33
contract
driven approach, 2
services, 9, 70, 148–150, 152, 154
use case, 2, 147
contracts
services, 144
controller, 154
create message, see message
decoupling, 144, 150, 154
destroy message, see message
decision node, 113
decoupling, 144, 150, 154
destroy message, see message
diagram
UML, 15
versus model, 15
edge in activity diagram, 113
encapsulation, see composition
in composition relationship, 77
transition to lower level of, 145
tree operator, 3

Object
definition of, 8
versus class and interface, 52
observer, see actor
owner
in composition relationship, 77

development, 149
merge node
activity diagram, 114
model
UML, 15
model-driven development, 149

nested activity, 117
node
in activity diagram, 112

event
in activity diagram, 113

fork, 119
found message, see message
functional requirement, 22
specified via use case, 22
functional requirements, see requirements

Generalization, see specialization

inheritance, 54
use case, see use case
granularity, 145
level of, 147
levels of, 142, 154
transition to lower level of, 145

Package
Diagram, 14
partition
in activity diagram, 115
Peer-to-peer relationship, see association
pluggability, 9
polymorphism, 55
on message parameters, 55
on message recipient, 55

Quality attributes
for organization, 136
low cost, 136
reliability, 136
scalability, 136

realization, 45
required, 46, 47
specification, 3
versus class and object, 52

Locality criterion, 3
loop operator, see sequence diagram
lost message, see message
low cost, see quality attributes

Merge node
activity diagram, 114
model
UML, 15
model-driven development, 149

Nested activity, 117
Node
in activity diagram, 112

Fork, 119
Found message, see message
Functional requirement, 22
Specified via use case, 22
Functional requirements, see requirements

Generalization, see specialization

Inheritance, 54
Use case, see use case
Granularity, 145
Level of, 147
Levels of, 142, 154
Transition to lower level of, 145

Holiday home, 3
House, 3

Implements, 45
Infrastructure, 1, 3
Inheritance, see generalization, see specialization
Interaction
Overview diagram, 13
Interaction diagram, 116
Interface, 9
Decoupling via, 45
Definition of, 9
Notation for, 45
Provided, 46

Realization, 45
Required, 46, 47
Specification, 3
Versus class and object, 52

Low cost, see quality attributes

Technology-Neutral Analysis and Design using UML and URDAD

reliability, see quality attributes
capability
requirement
  functional, 152
  conditional, 22
  mandatory, 22
  not use case, 22
requirements, see functional
  analysis, 2
  communication, 2
  documentation, 2
  elicitation, 2
  for architecture, 136
  functional, 3, 9, 16, 136, 146
  non-functional, 9
  planning, 2
  scope, 136
  UML based, 2
responsibilities
  identifying, 149
  responsibility, 149
  distribution, 3
  for activity, 115
responsibility domain, 150, 152
responsibility localization, see single responsibility principle
reusability, 150
role name, 38, see association
\[ S \]
scenario diagram, 102
scope
  of services, 3
scoping, see use case, 16, 24
sequence
  diagram, 13
sequence diagram, 12, 102
  activation bar, 104
  concurrency, 111
  contracts, 103
  duration constraint, 106
  interaction reference, 107
  interface based, 103
  levels of granularity, 104, 108
  life line, 104
  loop operator, 109
  objects on, 103
  par, 111
  return message, 104
  service requests, 104
  time axis, 103
  time constraint, 106
sequenceDiagram
messageTypes, 104
  asynchronous, 105
  create, 105
  destroy, 105
  found message, 106
  lost message, 106
  synchronous, 105
service, 3, 16, 37
  no return type, 39
  overloading, 40
  parameter, 37, 40
  in, 40
  inout, 40
  out, 40
  return type, 39
  compound, 39
service provider, see actor
services contract
  assign responsibilities to, 149
sequence diagram
  message to self, 104
  single responsibility principle, 142, 154
SLA, see contract
specialization
  multiple inheritance, 59
  notation for, 52
  substitutability, 53
specialization
  in class diagram, 52
stake holder, 16, 146
  for use case, 22
state chart, 13
stereotype, 19
  control, 19
  icon, 19
structured activity, 117
subject, 16
  scope of, 16
  use case diagram, 16, 18
synchronous message, see message
\[ T \]
thread
  of activity, 119
timing
  diagram, 13
traceability, 146, 154
transition
  automatic, 113
  in activity diagram, 113
\[ U \]
UML
  relationships
  summary, 96
URDAD, 145
  design phase, 149
  in context of MDA, 136
  in context of model driven development process, 138
  methodology, 145
  responsibility allocation, 149
responsibility identification, 149

usage scenarios, see use case

use case, 3, 17, 20, 22
  abstraction, 24, 25
  commonalities, 25
  concrete, 25
  diagram, 16, 20, 26
    scope, 16
    simple, 16
  extension, 23
    extension point, 23
  generalization, 24, 25
  relationship
    extend, 22
    include, 22

Use Case Responsibility Driven Analysis and Design, see URDAD

user, see actor
  role, 16, 18, 19

user work flow, 148

V
  view
    selective
      class diagram, 100
  visibility, see access levels
  vision, 3

W
  work flow, 3