Case study in system development - Notes

Lajos Kollár

Nóra Sterbinszky
# Table of Contents

1. Introduction  .............................................................. 2
   1.1. Software processes ...................................................... 2
   1.2. Process activities .................................................... 3
   1.3. Reuse-oriented software engineering .................................. 6
   1.4. Design and implementation ........................................... 7
   1.5. Specification .......................................................... 8
   1.6. Validation ............................................................ 9
   1.7. Evolution ............................................................ 9

2. UML ........................................................................ 9
   2.1. UML and metamodeling ................................................. 11
   2.2. Classification of UML 2.x diagrams .................................... 12
   2.2.1. Structural diagrams ................................................ 13
   2.2.2. Behavioral diagrams ............................................... 13
   2.2.3. Common elements of UML diagrams ............................... 14
   2.3. When to use UML diagrams? .......................................... 15

3. Requirements engineering .................................................. 16
   1. Classification of requirements ........................................... 18
   2. The Software Requirements Specification document .................. 20
   2.1. When is an SRS said to be good? .................................... 21
   2.2. Parts of an SRS ....................................................... 24

4. Analysis ...................................................................... 33
   4.1. Customer use cases .................................................... 39
   4.1.1. Browse/search ..................................................... 39
   4.1.2. Manage bookshelf ................................................ 41
   4.1.3. Move items to cart ................................................ 42
   4.1.4. Manage Shopping Cart ......................................... 43
   4.1.5. Move items to shelf .............................................. 44
   4.1.6. Place Order ........................................................ 45
   4.1.7. Feed Back ......................................................... 46
   4.1.8. Visit forums ....................................................... 47
   4.2. Administrator use cases ................................................ 48
   4.2.1. Manage Products .................................................. 49
   4.2.2. Update Order Status .............................................. 51
   4.2.3. Process Feedback ................................................ 52
   4.2.4. Compose/Send Newsletter .................................... 52
   4.2.5. Moderate Forums ............................................... 53
   4.3. Manager use cases ..................................................... 54
   4.3.1. Set Action Offers .................................................. 55
   4.3.2. Set Customer Discounts ....................................... 56
   4.3.3. View/Check Reports/Statistics ................................ 56
   4.4. Non-functional requirements .......................................... 63

3. System modeling ............................................................ 65
   1. Context models .......................................................... 66
   2. Interaction models ........................................................ 67
   3. Structural models ......................................................... 72
   3.1. Identification of domain classes ..................................... 85
   3.2. Refining the model .................................................... 88

4. Behavioral models .......................................................... 90
   4.1. Data-driven modeling ................................................. 90
   4.2. Event-driven modeling ............................................... 92

5. Robustness analysis ........................................................ 94

4. Architectural design .......................................................... 100
   1. Architecture of the case study application ............................. 101
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1. Logical architecture</td>
<td>101</td>
</tr>
<tr>
<td>1.2. Physical architecture</td>
<td>106</td>
</tr>
<tr>
<td>5. Detailed design</td>
<td>109</td>
</tr>
<tr>
<td>1. Criteria for modularization</td>
<td>109</td>
</tr>
<tr>
<td>1.1. Coupling</td>
<td>109</td>
</tr>
<tr>
<td>1.2. Cohesion</td>
<td>110</td>
</tr>
<tr>
<td>1.3. Open-Closed principle</td>
<td>111</td>
</tr>
<tr>
<td>2. Refinement and transformation of analysis-level design</td>
<td>111</td>
</tr>
<tr>
<td>2.1. Overview of attributes and methods</td>
<td>112</td>
</tr>
<tr>
<td>2.2. Shopping cart issues</td>
<td>116</td>
</tr>
<tr>
<td>2.3. What about Catalog?</td>
<td>117</td>
</tr>
<tr>
<td>2.4. Decouple OrderManagement from external dependencies</td>
<td>122</td>
</tr>
<tr>
<td>2.5. The show must go on</td>
<td>122</td>
</tr>
<tr>
<td>6. Using UML profiles</td>
<td>123</td>
</tr>
<tr>
<td>1. Profile</td>
<td>123</td>
</tr>
<tr>
<td>2. Metaclass</td>
<td>124</td>
</tr>
<tr>
<td>3. Stereotype</td>
<td>125</td>
</tr>
<tr>
<td>3.1. Stereotype application</td>
<td>126</td>
</tr>
<tr>
<td>3.2. Stereotype relationships</td>
<td>127</td>
</tr>
<tr>
<td>4. Tag definition</td>
<td>128</td>
</tr>
<tr>
<td>4.1. Tagged value</td>
<td>128</td>
</tr>
<tr>
<td>5. Extension</td>
<td>129</td>
</tr>
<tr>
<td>6. Reference</td>
<td>131</td>
</tr>
<tr>
<td>7. Profile application</td>
<td>131</td>
</tr>
<tr>
<td>8. UML profile diagram example</td>
<td>132</td>
</tr>
<tr>
<td>7. Service-oriented architecture</td>
<td>134</td>
</tr>
<tr>
<td>1. SoaML UML profile</td>
<td>134</td>
</tr>
<tr>
<td>2. Applying the SoaML profile using composite structure diagram of UML</td>
<td>136</td>
</tr>
<tr>
<td>References</td>
<td>140</td>
</tr>
</tbody>
</table>
List of Figures

1.1. Waterfall model [SOMMERVILLE2010] ........................................................................... 3
1.2. Incremental development [SOMMERVILLE2010] .............................................................. 5
1.3. Reuse-oriented development [SOMMERVILLE2010] ......................................................... 6
1.4. Requirements engineering process [SOMMERVILLE2010] .............................................. 7
1.5. Design process [SOMMERVILLE2010] ............................................................................ 8
1.6. Layered architecture of MOF . ..................................................................................... 11
1.7. UML diagram types . ..................................................................................................... 12
1.8. Framed class diagram with frame heading of long kind ................................................... 15
2.1. Readers of different types of requirements specification [SOMMERVILLE2010] ........... 18
2.2. Types of non-functional requirements [SOMMERVILLE2010] ....................................... 20
2.3. Users of a requirements document [SOMMERVILLE2010] ............................................. 21
2.4. Recommended structure for SRS documents ................................................................. 24
2.5. Actor Customer ........................................................................................................ 30
2.6. Generalization relationship between actors ................................................................. 30
2.7. Use case example ........................................................................................................... 31
2.8. An «extend» relationship between use cases. ............................................................... 31
2.9. Sample usage of extension points. .................................................................................. 32
2.10. Use cases B and C are extracted from larger use case A into separate use cases ([UML-
      DIAGRAMS.ORG]). ......................................................................................................... 32
2.11. Use case C is extracted from use cases A and B to be reused by both use cases using UML
      include relationship ([UML-DIAGRAMS.ORG]). ......................................................... 32
2.12. A use case including several use cases. ........................................................................ 33
2.13. Overview of use case packages. ................................................................................. 36
2.14. All identified use cases on a single diagram. ............................................................... 37
2.15. Overview of customer use cases. .................................................................................. 39
2.16. Use case UC001 ............................................................................................................ 39
2.17. Use case UC002 ............................................................................................................ 41
2.18. Use case UC003 ............................................................................................................ 42
2.19. Use case UC004 ............................................................................................................ 43
2.20. Use case UC005 ............................................................................................................ 44
2.21. Use case UC006 ............................................................................................................ 45
2.22. Use case UC007 ............................................................................................................ 46
2.23. Use case UC008 ............................................................................................................ 47
2.24. Overview of administrator use cases. ......................................................................... 48
2.25. Use case UC009 ............................................................................................................ 49
2.26. Use case UC011 ............................................................................................................ 51
2.27. Use case UC012 ............................................................................................................ 52
2.28. Use case UC013 ............................................................................................................ 53
2.29. Use case UC014 ............................................................................................................ 53
2.30. Overview of manager use cases. .................................................................................. 54
2.31. Use case UC015 ............................................................................................................ 55
2.32. Use case UC016 ............................................................................................................ 56
2.33. Use case UC017 ............................................................................................................ 56
2.34. Activity diagram for use cases Browse/Search and Purchase Order ......................... 57
2.35. Activity diagram sample with lot of features ([SPARXSYSTEMS.COM]) ................... 58
2.36. Sequence diagram of the activity Checkout .................................................................. 59
2.37. Call accept() if balance > 0, call reject() otherwise [UML-DIAGRAMS.ORG] ........ 62
2.38. Post comments if there were no errors [UML-DIAGRAMS.ORG] ............................. 62
2.40. Web customer and Bookshop use (reference) interaction Checkout [UML-DIAGRAMS.ORG] 62
2.41. Communication diagram of the online bookstore’s find book and checkout activities ... 63
3.1. Context model of the online store. ............................................................................... 66
3.2. Use cases describing interactions to Bank and Shipping system (external systems). ...... 67
3.3. Use cases describing interactions to Accounting system (external systems). .............. 69
3.4. Use cases describing interactions to Accounting system (external systems). .............. 70
3.5. Activity diagram of the use case Place Order and related activities ......................... 71
6.2. Profile Servers [UML-DIAGRAMS.ORG] ................................................................. 124
6.3. EJB Profile shown as a package with URI attribute [UML-DIAGRAMS.ORG] .......... 124
6.4. Metaclass Component [UML-DIAGRAMS.ORG] ....................................................... 125
6.5. Stereotype Computer extends metaclass Device [UML-DIAGRAMS.ORG] ............... 125
6.7. Stereotype Servlet with attached custom icon [UML-DIAGRAMS.ORG] .................. 125
6.8. Actor is extended by stereotype Web Client with attached custom icon [UML-DIAGRAMS.ORG] 125
6.9. Device extended by Server stereotype with tag definitions and custom icon [UML-DIAGRAMS.ORG] .. 126
6.10. Stereotype «Servlet» applied to the model element SearchServlet [UML-DIAGRAMS.ORG] 126
6.11. Servlet stereotype applied to the class SearchServlet [UML-DIAGRAMS.ORG] .......... 127
6.15. Abstract stereotype Session EJB is specialized by stereotypes Stateless EJB and Stateful EJB [UML-DIAGRAMS.ORG] ................................................................. 128
6.16. Stereotype Computer with tag definitions for vendor, CPU, and memory [UML-DIAGRAMS.ORG] ................................................................. 128
6.17. Stereotype Computer applied using “traditional” tag values notation [UML-DIAGRAMS.ORG] 129
6.18. Stereotype Computer applied with tag values in compartment [UML-DIAGRAMS.ORG] 129
6.20. Metaclass Class is extended by stereotype Customer [UML-DIAGRAMS.ORG] .......... 130
6.21. Required extension of metaclass Component by stereotype WebService [UML-DIAGRAMS.ORG] 130
6.22. Stereotype Provider extends either (or both?) Interface or Class metaclasses [UML-DIAGRAMS.ORG] ................................................................. 130
6.23. Metaclass Component is referenced (imported) by profile Servlets [UML-DIAGRAMS.ORG] 131
6.25. Simplified example of the unofficial Java EJB 3.0 Profile [UML-DIAGRAMS.ORG] .......... 132
7.1. SoaML UML Profile - Contracts .............................................................................. 135
7.2. SoaML UML Profile - Services ................................................................................ 135
7.3. SoaML UML Profile - Service Data ......................................................................... 136
7.4. Shipping service with two customers ..................................................................... 137
7.5. ShippingService service interface ......................................................................... 137
7.6. Service interfaces of a compound service ............................................................. 139
7.7. MessageTypes in purchase order processing ......................................................... 139
List of Tables

1.1. UML elements that can have frames ................................................................. 14
3.1. UML interpretations of classes ......................................................................... 73
3.2. Multiplicity examples ....................................................................................... 80
3.3. Optional modifiers for properties ...................................................................... 80
3.4. States of a shopping cart .................................................................................. 93
3.5. Stimuli of a shopping cart ................................................................................ 94
5.1. Class Order ........................................................................................................ 113
5.2. Class Manager ................................................................................................. 114
List of Examples

2.1. Non-functional requirements for the case study. ......................................................... 63
A projekt az Európai Unió támogatásával, az Európai Szociális Alap társsfinanszírozásával valósul meg.
Chapter 1. Introduction

This case study is intended to serve as a study material for the course Technology of System Development for Software IT BSc students at the Faculty of Informatics, University of Debrecen. The primary goal of the case study is to show the major steps of analysis and design activities of the software engineering lifecycle. The reader is expected to know the basics of the object-oriented (OO) paradigm and have some experience in programming in at least one OO language. However, the case study is intended to be independent of any concrete programming languages since it is out of the scope of this study material to deal with implementation details.

Knowledge about some guidelines, principles and patterns of software design might also be required. Students learn those pieces of background information in the prerequisite course.

Our running example will be a Book and DVD store. This example was chosen in order to be general enough for showing the most challenges that the specification and design of information systems have. The main purpose of the system is to allow the customers to browse and search products in an online warehouse, and they should also be able to place orders for those products. Detailed requirements will be discussed later, in the chapter entitled Requirements engineering.

The artifacts that are produced throughout the stages of development are documentations that are illustrated by models. Due to the rigid nature of some development documentations (i.e., requirements specification document or software design document) we will not include those documentations in their full details; instead, our focus is on the process of establishing the models describing the system of the case study. The models will predominantly be described using the Unified Modeling Language (UML) standard which is not only a de facto standard for describing models of software systems but has also become a de iure standard under reference numbers ISO/IEC 19505-1 and 19505-2. The current version of UML by the time of writing is 2.4.1 therefore we will use this version, however, version 2.5—which will have a major change in the organization of the specification—is under development.

This document is organized as follows. The rest of this chapter provides a very short overview of software process models and an outline of the UML 2.4.1 standard. Chapter 2 discusses the major issues of requirements engineering and provides a detailed view of the requirements of our case study. Chapter 3 introduces few high-level system modeling steps that will guide the reader to the architectural and detailed design covered in Chapter 4 and 5. The built-in extensibility mechanism of UML (called profiles) are detailed in Chapter 6 along with some examples of tailoring the design of the system to specific platforms. The last chapter is concerned with the emerging field of service-oriented architectures and discusses issues on how to expose system functionalities as services that can later be orchestrated into more complex services thus providing means of cross-organizational integration of software services.

1. Software processes

A software process can be defined as a set of related activities that will lead to the creation a software product. Some systems are developed from scratch while other systems are developed based on existing systems by extending them. Another group of systems are created when existing components are integrated and configured.

According to [SOMMERVILLE2010], the four fundamental activities of software engineering are:

1. **Software specification.** Software's functionality, required product attributes and operational constraints are defined.
2. **Software design and implementation.** The software is designed and programmed in order to meet the specification.
3. **Software validation.** Software must be validated to ensure that it delivers the required functionalities, meets the required product attributes and fulfills the constraints that users (customers) want so briefly it is checking whether the system fulfills its goals.
4. **Software evolution.** The development of software is not finished with deployment but the system should evolve to meet the changing needs.
These are those processes that you will inevitably meet regardless of which software process is applied in your project. Of course, these processes are complex processes: they include sub-processes or sub-activities like requirements management, architectural design, interface design, unit testing, acceptance testing, etc. Some supporting activities like configuration management or documentation is also needed when creating software.

When describing software processes, the activities involved in those processes should be discussed like designing the user interface or specifying a data structure. Ordering of activities are also important from process view so we can refine our previous definition on software processes that will include not only the set of activities that should be executed during development but the ordering of those activities are important, as well. Software process descriptions, however, are not limited to describe activities and ordering of them but they might also include information on

1. products or artifacts that are the results of a process activity,
2. roles that reflect the responsibilities of the people involved in the process, and
3. pre- and postconditions which define logical statements that must be true before or after a process activity has been performed or an artifact has been created.

A Software Requirements Specification document is an example of an artifact that is the result of the specification activity; programmer or project manager are examples of roles; while a precondition of integration testing might be that all the components that should be integrated are accessible and they are unit tested individually.

Important

Software processes are complex and, like all intellectual and creative processes, rely on people making decisions and judgments. There is no silver bullet that means that there is not an ideal process of developing software. Processes vary based on their domain, application type, developing organization, etc. Majority of organizations have developed their own software development processes that are standardized at an organizational level. Processes have evolved to get the best out of the people's capabilities within an organization and the specific characteristics of the systems to be developed.

Sommerville describes three categories of software process models (that can be, however, often used together). In fact, these are generic models that are not definitive descriptions of processes but are seen as abstractions. So you can think of them like software process frameworks that can and should be extended and/or adapted in order to create specific software engineering processes. These generic processes are:

1. **Waterfall model** which takes the four fundamental process activities described above and represents them as separate stages like requirements specification, software design, implementation, testing, and so on.
2. **Incremental development** which interleaves specification, development, and validation activities. The system is developed as a series of versions (increments), with each version adding functionality to the previous one.
3. **Reuse-oriented development** which is, instead of creating systems from scratch, focuses on the integration of existing components into a system so this is based on the existence of a significant number of reusable components.

Software processes can be categorized as either plan-driven or agile processes. Plan-driven processes are processes where all of the process activities are planned in advance and progress is measured against this plan while in agile processes, planning is incremental and it is easier to change the process to reflect changing customer requirements.

**1.1. Waterfall model**

The waterfall model is an example of a plan-driven process—in principle, you must plan and schedule all of the process activities before starting work on them. As it is seen on the figure, phases are cascaded from one phase to another, that is while it is known as the ‘waterfall model’.

**Figure 1.1. Waterfall model [SOMMERVILLE2010]**
The principal stages of the waterfall model directly reflect the fundamental development activities:

1. **Requirements analysis and definition.** The system’s services, constraints, and goals are established by consultation with system users. They are then defined in detail and serve as a system specification.

2. **System and software design.** The systems design process allocates the requirements to either hardware or software systems by establishing an overall system architecture. Software design involves identifying and describing the fundamental software system abstractions and their relationships.

3. **Implementation and unit testing.** During this stage, the software design is realized as a set of programs or program units. Unit testing involves verifying that each unit meets its specification.

4. **Integration and system testing.** The individual program units or programs are integrated and tested as a complete system to ensure that the software requirements have been met. After testing, the software system is delivered to the customer.

5. **Operation and maintenance.** Normally (although not necessarily), this is the longest life cycle phase. The system is installed and put into practical use. Maintenance involves correcting errors which were not discovered in earlier stages of the life cycle, improving the implementation of system units and enhancing the system’s services as new requirements are discovered.

At least in principle, the result (artifact) of each phase is one or more documents that should be approved and the subsequent phase should not be started until the previous phase has completely been finished. In practice, however, these stages overlap and feed information to each other. During design, problems with requirements can be identified, during coding some of the design problems can be found, etc. The software process therefore is not a simple linear model but involves feedback from one phase to another as it is depicted on the figure. Documents produced in each phase may then have to be modified to reflect the changes made.

In principle, the waterfall model should only be applied when requirements are well understood and unlikely to change radically during development as this model has a relatively rigid structure which makes it relatively hard to accommodate change when the process is underway.

An important variant of the waterfall model is formal system development, where the specification is described using a mathematical model which will later be refined into executable code (possibly through a series of other models) using consistency-preserving mathematical transformations. If the mathematical transformations applied are correct, we have a strong argument that a program generated in this way is consistent with its specification.

### 1.2. Incremental development
Incremental development is based on the idea of developing an initial implementation, exposing this to user feedback and evolving it through several versions until an acceptable system has been developed. Activities of specification, development and validation are not separated but interleaved. There are also lots of feedback across those activities.

**Figure 1.2. Incremental development [SOMMERVILLE2010]**

Incremental software development, which is a fundamental part of agile approaches, is better than a waterfall approach for most business, e-commerce, and personal systems. Incremental development reflects the way that we solve problems. We rarely work out a complete problem solution in advance but move toward a solution in a series of steps, backtracking when we realize that we have made a mistake. By developing the software incrementally, it is cheaper and easier to make changes in the software as it is being developed [SOMMERVILLE2010].

Each system increment captures a piece of the functionality that is needed by the customer. Generally, the early increments of the system should include the most important or most urgently required functionality. If this is the case then customers can evaluate the system at a relatively early stage in the development to see if it delivers what is required. If not, then only the current increment has to be changed and, possibly, new functionality defined for later increments. Compared to the waterfall model, incremental development has three important benefits:

1. There is a reduced cost of dealing with changing requirements since the amount of analysis and documentation activities that has to be redone is much less than is required when using the waterfall model.

2. It is much easier to get customer feedback on the work done. In waterfall-like processes (at least, theoretically) customers first meet the system when it is fully developed and tested and is deployed. If something is not acceptable than it is a real problem (that is the reason why waterfall should only be applied when requirements are well-understood). In iterative development, customers regularly receive a (not fully functional) version of the system that can be commented which provides valuable feedback.

3. More rapid delivery and deployment is possible so customers can get value out of the software earlier than is possible with the waterfall process. In incremental development it is allowed (and advised) to deliver useful functionalities even in early stages of the development.

Incremental development in some form is now the most common approach for the development of application systems. This approach can be either plan-driven, agile, or, more usually, a mixture of these approaches. In a plan-driven approach, the system increments are identified in advance; if an agile approach is adopted, only the early increments are identified but the development of later increments depends on progress and customer priorities.
1.3. Reuse-oriented software engineering

Reuse is one of the most important concepts of today's software engineering since it can not only save a given amount of work when existing components providing a given functionality are reused but existing components might have lots of testing received so far so we can possibly build more reliable systems based on them. Nowadays more and more software projects apply reuse to some extent, however, some of them relies more on reused components than others.

Reuse often happens in an informal way when people working on the project know of designs or code that are similar to what is required. They look for these, modify them as needed, and incorporate them into their system. This is basically the application of patterns in the development process.

Figure 1.3. Reuse-oriented development [SOMMERVILLE2010]

A general process model for reuse-oriented software engineering is shown in the above figure. Although the initial requirements specification stage and the validation stage are comparable with other software processes, the intermediate stages in a reuse-oriented process are different. According to [SOMMERVILLE2010], these stages are:

1. Component analysis. Based on the requirements specification, components that implement (some part of) the specification are looked for. In the most of the cases there is no exact match and the components that may be used only provide some of the functionality required.

2. Requirements modification. During this stage, the requirements are analyzed using information about the components that have been discovered. They are then modified to reflect the available components. Where modifications are impossible, the component analysis activity may be re-entered to search for alternative solutions.

3. System design with reuse. During this phase, the framework of the system is designed or an existing framework is reused. The architects will perform the design by taking into account the components that are reused and they will organize the framework accordingly. New pieces of software may have to be designed if reusable components are not available.

4. Development and integration. Software that cannot be externally procured is developed, and the components and commercial-off-the-shelf (COTS) systems are integrated to create the new system. System integration, in this model, may be part of the development process rather than a separate activity.

There are basically three types of software components that can be used in a reuse-oriented process:

1. Web services that are developed according to well-known service standards and which will become available for remote invocation.

2. Collections of objects that are developed as a package to be integrated with a component framework such like .NET or Java EE.

3. Standalone software systems that are configured for use in a particular environment.

One of the most important advantage of reuse-oriented software engineering is the reduced amount of software to be developed and therefore reduced cost and risks. As a consequence, it can also lead to faster delivery. However, compromises must be achieved on requirements which might lead to a system that does not meet the real needs of users. Some control over the system evolution might also be lost as new versions of the reusable
components are not under the control of the organization using them (no influence on what functionalities to include/exclude, etc.).

1.4. Process activities

In this section some of the most important activities and sub-activities of the four general process activities will be discussed very briefly. For a more detailed description of activities, the reader should read some general software engineering textbooks like [SOMMERVILLE2010].

The four basic process activities of specification, development, validation, and evolution are organized differently in different development processes. In the waterfall model, they are organized in sequence while in incremental development they are interleaved. How these activities are performed might depend on the type of software, people involved in development, etc.

1.4.1. Specification

Software specification or requirements engineering is the process of understanding and defining what services are required from the system and identifying the constraints on the system’s operation and development [SOMMERVILLE2010].

Figure 1.4. Requirements engineering process [SOMMERVILLE2010]

There are four main sub-activities in requirements engineering processes:

1. **Feasibility study** when an estimation of whether the identified requirements may be satisfied using current software and hardware technologies is made. A feasibility study should be relatively cheap and quick; it should inform the decision of whether or not to go ahead with a more detailed analysis.

2. **Requirements elicitation and analysis**, which is the process of deriving the system requirements through observation of existing systems, discussions with stakeholders, etc. This may involve the development of one or more system models and prototypes that can help us understanding the system to be specified.

3. **Requirements specification**, which is the activity of translating the information gathered during the analysis activity into a (formal or informal, depending on the underlying process used) document that defines a set of requirements. Two types of requirements may be included in this document:

   a. User requirements are abstract statements of the system requirements for the customer and end-user of the system;

   b. System requirements are a more detailed description of the functionality to be provided.
4. Requirement validation is the process of checking the requirements for realism, consistency and completeness. During this process our goal is to reveal errors in the requirements document. When an error is found, the requirements specification document needs to be modified to correct the problems.

Of course, the activities in the requirements process are not necessarily executed in a strict sequence but the activities of analysis, definition, and specification can be interleaved. In agile methods, such as extreme programming, requirements are developed incrementally according to user priorities and the elicitation of requirements comes from users who are part of the development team.

1.4.2. Design and implementation

The implementation phase is the process of converting a system specification into an executable system. It always involves processes of software design and programming but, if an incremental approach to development is used, may also involve refinement of the software specification [SOMMERVILLE2010].

Figure 1.5. Design process [SOMMERVILLE2010]

The activities needs to be carried out during the design process vary depending on the type of the system needs to be developed. However, for the development of information systems, the main four activities are the following:

1. Architectural design, where the overall system structure is identified. It involves finding principal components (sometimes called sub-systems or modules) along with their relationships. Distribution of components (to hardware nodes and to development teams) should also be discussed.

2. Interface design, where the interfaces among various system components are defined. This interface specification must be unambiguous. With a precise interface, a component can be used without other components having to know how it is implemented. Once interface specifications are agreed, the components can be designed and developed concurrently.
3. Component design, where we take each system component and design how it will operate. This may be a simple statement of the expected functionality to be implemented, with the specific design left to the programmer.

4. Database design, where the data structures used by the system are designed and their representation in a database is dealt with. The sub-activities that need to be performed here depends on whether an existing database is to be reused or a new database is to be created.

### 1.4.3. Validation

Software validation or, more generally, verification and validation (V&V) is intended to show that a system both conforms to its specification and that it meets the real needs of the customer or user of the application. Testing, where the system is executed using simulated test data, is an important validation technique but some checking processes, such like inspections or reviews should also be applied during V&V processes.

Testing has three main stages [SOMMERVILLE2010]:

1. Development testing. The components that compose the system are tested by the people developing the system. Each component is tested independently, without other system components. Components may be simple entities such as functions or object classes, or may be coherent groupings of these entities. Test automation tools, such as JUnit are commonly used.

2. System testing. System components are integrated to create a complete system. This process is concerned with finding errors that result from unanticipated interactions between components and component interface problems. It is also concerned with showing that the system meets its functional and non-functional requirements, and testing the emergent system properties.

3. Acceptance testing. This is the final stage in the testing process before the system is accepted for operational use. The system is tested with data supplied by the system customer rather than with simulated test data. Acceptance testing may reveal errors and omissions in the system requirements definition, because the real data exercise the system in different ways from the test data.

### 1.4.4. Evolution

Requirements are always changing, even after the system has been put into its operating environment. Emerging systems with new interfaces that our software needs to communicate with, or business environment changes might result in changing needs.

Historically, there has always been a split between the process of software development and the process of software evolution (software maintenance). However, this distinction between development and maintenance is increasingly irrelevant. Hardly any software systems are completely new systems and it makes much more sense to see development and maintenance as a continuum. Rather than two separate processes, it is more realistic to think of software engineering as an evolutionary process where software is continually changed over its lifetime in response to changing requirements and customer needs [SOMMERVILLE2010].

### 2. UML

The Unified Modeling Language™ (UML®) is a standard visual modeling language intended to be used for

- modeling business and similar processes,
- analysis, design, and implementation of software-based systems.

UML is a common language used by various parties involved in the software engineering lifecycle: business analysts, software architects, developers, testers and system administrators. It is used to describe, specify, design, and document existing or new business processes, structure and behavior of artifacts of software systems.

UML can be applied to various application domains (e.g., banking, selling and buying product via internet, healthcare, finance, etc.) and it can be used together with all major object and component software development methods and for diverse implementation platforms (e.g., Java EE, .NET).
It is important to emphasize that UML is “only” a standard modeling language, not a software development process. A software development process is expected to

• provide guidance to the order of the activities performed by a group of people (the development team),

• specify what artifacts should be developed,

• control the tasks performed by both individual developers and the team as a whole, and

• offer criteria in order to monitor and measure the produced artifacts and the performed activities within a project.

UML is not about providing these pieces of information. It is not prescriptive but descriptive: as a modeling language, it does not prescribe any activities and does not require to produce any artifacts; however, it can be used to describe the artifacts that an applied software development process requires.

Therefore, UML is intended to be process independent and it can be applied in the context of different processes. However, it is most suitable for use case-driven, iterative and incremental development processes. An example of such process is Rational Unified Process (RUP).

Important

Our case study does not depend on the application of a given software process model. However, the presentation of the steps will resemble the reader similar to those of plan-driven approaches. The reason for that is very trivial: plan-driven approaches are more documentation-oriented than agile methods (which are much more code-oriented) and therefore, in our opinion, it is easier to describe (and understand) a system in a plan-driven way for learning purposes, especially because we focus on the analysis and design instead of actual programming.

UML is not completely visual which means that given some UML diagram, the described part or behavior of the system might not be understood from the diagram alone. Some pieces of information could be intentionally omitted from the diagram, some information represented on the diagram could have different interpretations, and some concepts of UML have no graphical notation at all, so there is no way to depict those on diagrams.

• For example, semantics of multiplicity of actors and multiplicity of use cases on use case diagrams is not defined precisely in the UML specification and could mean either concurrent or successive usage of use cases.

• Name of an abstract classifier is shown in italics while final classifier has no specific graphical notation, so there is no way to determine whether classifier is final or not from the diagram.

This, on one hand, provides a great flexibility when having a more or less informal system description intended to be read by human beings (some people involved in the process). On the other hand, the lack of proper semantics (that can lead to ambiguous interpretations) or missing notations are problematic when applying a model-driven style of development that would require precise statements about each models so that a model transformation or code generation might take place.

A UML diagram is a partial graphical representation (view) of a model of a system under design, implementation, or already in existence. UML diagrams contain graphical elements (symbols)—UML nodes connected with edges (also known as paths or flows)—that represent elements in the UML model of the designed system. Since it is uncommon to describe a system by using UML models only, system description might also contain other documentation such as use cases written as templated texts.

The UML offers a set of diagram types to capture the different aspect of software systems. The diagram type is defined by the primary graphical symbols shown on the diagram. For example, a diagram where the primary symbols are classes is a class diagram. A diagram showing actors and use cases is called a use case diagram. A sequence diagram shows sequence of message exchanges between lifelines. UML specification also allows to create a mixture of different kinds of diagrams, e.g. to combine structural and behavioral elements to show a state machine nested inside a use case. Consequently, the boundaries between the various kinds of diagrams are not strictly enforced. At the same time, some UML tools do restrict set of available graphical elements which could be used when working on specific type of diagram.
2.1. UML and metamodeling

Object Management Group (OMG) is the name of the organization responsible for developing and maintaining the UML standard. UML version 2.4.1 consists of two separate documents, namely, the UML Infrastructure ([UML2.4.1INFRA]) and the UML Superstructure ([UML2.4.1SUPER]).

Note

This will be changed from the 2.5 version of the UML since these documents will be merged together.

UML Infrastructure defines a core metamodel that is used to define the full MetaObject Facility (MOF), UML and Profiles. The key concept to understand is that in UML, reuse is the pervasive design consideration. The core of the UML Infrastructure is defined on an abstract level, and is extended by using object-oriented principles to define the UML Superstructure. Superstructure complements Infrastructure by defining user level constructs required for UML 2.4.1 (e.g., various digram types).

Note

The MetaObject Facility (MOF) Specification is the foundation of OMG's industry-standard environment where models can be exported from one application, imported into another, transported across a network, stored in a repository and then retrieved, rendered into different formats (including XMI, OMG's XML-based standard format for model transmission and storage), transformed, and used to generate application code.

MOF is designed as a four-layered architecture (see Figure 1.6, ―Layered architecture of MOF .‖). It provides a meta-meta model at the top layer, called the M3 layer. This M3-model is the language used by MOF to build metamodels, called M2-models. The most prominent example of a Layer 2 MOF model is the UML metamodel, the model that describes the UML itself. These M2-models describe elements of the M1-layer, and thus M1-models. These would be, for example, models written in UML. The last layer is the M0-layer or data layer. It is used to describe real-world objects.

Beyond the M3-model, MOF describes the means to create and manipulate models and metamodels by defining CORBA interfaces that describe those operations. Because of the similarities between the MOF M3-model and UML structure models, MOF metamodels are usually modeled as UML class diagrams. A supporting standard of MOF is the XML Metadata Interchange (XMI), which provides an exchange format based on XML for models of the various layers.

MOF is a closed metamodeling architecture; it defines an M3-model, which conforms to itself. MOF allows a strict meta-modeling architecture; every model element on each layer is strictly in correspondence with a model element of the layer above. MOF only provides a means to define the structure, or abstract syntax of a language or of data. For defining metamodels, MOF plays exactly the role that EBNF plays for defining programming language grammars. MOF is a Domain Specific Language (DSL) used to define metamodels, just as EBNF is a DSL for defining grammars. Similarly to EBNF, MOF could be defined in MOF. This is referred to as metacircular description.

Figure 1.6. Layered architecture of MOF [http://umlbase.com/learn/tag/meta-object-facility/].
2.2. Classification of UML 2.x diagrams

UML specification defines two major kinds of UML diagrams: structural diagrams and behavioral diagrams.

Structural diagrams show the static structure of the system and its parts on different abstraction and implementation levels and how they are related to each other. The elements in a structure diagram represent the meaningful concepts of a system, and may include abstract, real world and implementation concepts.

Behavioral diagrams show the dynamic behavior of the objects in a system, which can be described as a series of changes to the system over time.

UML 2.4 diagrams could be categorized hierarchically as shown below:

Figure 1.7. UML diagram types http://creately.com/blog/diagrams.uml-diagram-types-examples/.
2.2.1. Structural diagrams

Structural diagrams are not utilizing time related concepts, do not show the details of dynamic behavior. However, they may show relationships to the behaviors of the classifiers exhibited in the structure diagrams.

- **Class diagram** is a static structure diagram which describes structure of a system at the level of classifiers (classes, interfaces, etc.). It shows some classifiers of the system, subsystem or component, different relationships between classifiers, their attributes and operations, constraints.

- **Object diagram** can be defined as "a graph of instances, including objects and data values. A static object diagram is an instance of a class diagram; it shows a particular snapshot of the detailed state of a system at a point in time." It also stated that object diagram is "a class diagram with objects and no classes."

- **Package diagram** shows packages and relationships between the packages. A package is a namespace that is used to group together elements that are semantically related and might change together. It is a general purpose mechanism to organize elements into groups to provide better structure for system model.

- **Composite Structure Diagram** could be used to show the internal structure of a classifier, classifier interactions with the environment through ports, a behavior of a collaboration. The term "structure" for this type of diagrams is defined in UML as a composition of interconnected elements, representing run-time instances collaborating over communications links to achieve some common objectives.

- **Component diagram** shows components and their dependencies. This diagrams type plays an important role in modeling systems that are used for Component-Based Development (CBD), to describe systems with Service-Oriented Architecture (SOA).

- **Deployment diagram** shows architecture of the system as deployment (distribution) of software artifacts to deployment targets. In UML 2.x, artifacts are deployed to nodes, and artifacts could manifest (implement) components so components are deployed to nodes indirectly through artifacts. Deployment diagrams could be used to show logical or physical network architecture of the system.

- **Profile diagram** is an auxiliary UML diagram which allows defining custom stereotypes, tagged values, and constraints. The Profile mechanism has been defined in UML for providing a lightweight extension mechanism to the UML standard. Profiles allow to adapt the UML metamodel for different platforms (such as Java EE or .NET), or domains (such as real-time or business process modeling). Even though the concept of profiles are older, Profile diagrams were first introduced in UML 2.0.

2.2.2. Behavioral diagrams

Behavioral diagrams show the dynamic behavior of the objects in a system, which can be described as a series of changes to the system over time.
• **Use case diagrams** are behavior diagrams used to describe a set of actions (use cases) that some system or systems (subject) should or can perform in collaboration with one or more external users of the system (actors) to provide some observable and valuable results to the actors or other stakeholders of the system(s). Note, that UML 2.4 specification also defines use case diagrams as specialization of class diagrams (which are structure diagrams). Use case diagrams could be considered as a special case of class diagrams where classifiers are restricted to be either actors or use cases and the most used relationship is association.

• **Activity diagram** shows sequence and conditions for coordinating lower-level behaviors, rather than which classifiers own those behaviors. These are commonly called control flow and object flow models.

• **State machine diagram** is used for modeling discrete behavior through finite state transitions. In addition to expressing the behavior of a part of the system, state machines can also be used to express the usage protocol of part of a system. These two kinds of state machines are referred to as behavioral state machines and protocol state machines.

• **Interaction diagrams** include few different types of diagrams:
  
  • **Sequence diagram** is the most common kind of interaction diagrams, which focuses on the message interchange between lifelines (objects).

  • **Communication diagram** (previously known as Collaboration Diagram) is a kind of interaction diagram, which focuses on the interaction between lifelines where the architecture of the internal structure and how this corresponds with the message passing is central. The sequencing of messages is given through a sequence numbering scheme.

  • **Interaction overview diagram** defines interactions through a variant of activity diagrams in a way that promotes overview of the control flow. Interaction overview diagrams focus on the overview of the flow of control where the nodes are interactions or interaction uses. The lifelines and the messages do not appear at this overview level. Timing diagrams are used to show interactions when a primary purpose of the diagram is to reason about time.

  • **Timing diagrams** are used to show interactions when a primary purpose of the diagram is to reason about time. They focus on conditions changing within and among lifelines along a linear time axis.

2.2.3. Common elements of UML diagrams

Each UML diagram has a contents area. As an option, some diagrams may also have a frame (shown as rectangle) with frame heading. The frame could be used in the cases when the diagrammed element has some owned elements that are attached to the border, like ports for classes and components, and entry/exit points on state machines.

The heading of a frame represents the kind, name, and parameters of the element owning or namespace enclosing the elements represented in the contents area.

```
frame-heading ::= [ element-kind ] element-name [ parameters ]
element-kind ::= short-element-kind | long-element-kind
```

The following UML elements and namespaces could have frames:

**Table 1.1. UML elements that can have frames**

<table>
<thead>
<tr>
<th>Short kind</th>
<th>Long kind - owning element or enclosing namespace</th>
</tr>
</thead>
<tbody>
<tr>
<td>act</td>
<td>activity</td>
</tr>
<tr>
<td>class</td>
<td>class</td>
</tr>
<tr>
<td>cmp</td>
<td>component</td>
</tr>
<tr>
<td>dep</td>
<td>deployment</td>
</tr>
<tr>
<td>sd</td>
<td>interaction</td>
</tr>
<tr>
<td>pkg</td>
<td>package</td>
</tr>
<tr>
<td>stm</td>
<td>state machine</td>
</tr>
</tbody>
</table>
Short kind | Long kind - owning element or enclosing namespace
---|---
uc | use case

Figure 1.8. Framed class diagram with frame heading of long kind

Note

It is a common mistake to assume that frame kind specifies type of the depicted UML diagram. In fact, type of the diagram is defined by the primary graphical elements shown in the contents area of the diagram. Frame kind is kind of the UML element owning or namespace enclosing the elements shown in the frame contents area. For instance, an activity, sequence or state machine diagram belonging to a use case (i.e., providing more details on it) can still be of a use case type.

2.3. When to use UML diagrams?

UML diagrams are mostly useful if they express something in a higher level of abstraction than the code. Writing UML just for the sake of writing UML becomes unneeded bureaucracy and makes the project and the code less adaptable to changes with no benefit whatsoever. For example, a UML class diagram showing all the classes on a package, with all their attributes and methods—something that can be easily auto-generated—provides no value at all: it is at the same level of abstraction than the code. Plus, the code will most surely be a better source for that information because it will always be up to date, and it will probably be documented and organized in a way that is easier to know which methods/attributes/things are more important. On the other hand, if you have concepts of a higher level of abstraction than what can be expressed on the code, documenting those on a diagram is a good idea. For example, a diagram showing the higher level abstract modules on a complex system, with their dependencies and maybe a little description of their responsibilities and what package/namespace they map to in the source code can be a really useful for a new team member that needs to be introduced to the project, or can also be used to figure out where a new class/functionality should be thrown. Another example of a useful diagram could be a sequence diagram showing the high-level steps to be taken in a communication protocol. Maybe each step of those have it’s little quirks and complexities, but it is probably enough to describe them in the code itself. The higher level diagram can help a programmer to understand the “big picture” of things easily without needing to worry about the complexities of each interaction. Anyway, those are just some examples: there are lots of cases where a simple diagram can be of a lot of help. Just remember that you should be doing them only when you cannot express something in the code itself. If you find yourself using UML diagrams to explain the source code itself, make the source code more self-documenting instead. Finally, some of the general rules that apply to code can also apply to diagrams: avoid repeating yourself, keep it simple, don’t fear of changing things (just because something is documented on a UML diagram does not mean it cannot be changed) and always think of who will be reading/maintaining those diagrams in the future (probably your future-self) when writing them.

Important

It is not our goal to introduce the details of all diagram types of UML 2.x to the reader. It is the case study which drives which UML diagram types will be used: those that are needed to model our sample application. However, we will deal with the majority of diagram types (at a different level of detail), even if it is important to underline that not all diagram types are equally important. We use some of them much more frequently than others, that is the reason why particular importance will be given to Use case diagrams, Class diagrams and Sequence diagrams. Different application types might require the use of different diagram types.
Chapter 2. Requirements engineering

Requirements are the basis for every project, defining what the stakeholders—users, customers, suppliers, developers, businesses—in a potential new system need from it and also what the system must do in order to satisfy that need. To be well understood by everybody they are generally expressed in natural language and herein lies the challenge: to capture the need or problem completely and unambiguously without resorting to specialist jargon or conventions. Once communicated and agreed, requirements drive the project activity. However, the needs of the stakeholders may be many and varied, and may indeed conflict. These needs may not be clearly defined at the start, may be constrained by factors outside their control or may be influenced by other goals which themselves change in the course of time. Without a relatively stable requirements base, a development project can only flounder. It is like setting off on a sea journey without any idea of the destination and with no navigation chart. Requirements provide both the “navigation chart” and the means of steering towards the selected destination.

Agreed requirements provide the basis for planning the development of a system and accepting it on completion. They are essential when sensible and informed tradeoffs have to be made and they are also vital when, as inevitably happens, changes are called for during the development process. How can the impact of a change be assessed without an adequately detailed model of the prior system? Otherwise, what is there to revert to if the change needs to be unwound? Even as the problem to be solved and potential solutions are defined we must assess the risks of failing to provide a satisfactory solution. Few sponsors or stakeholders will support product or systems development without a convincing risk management strategy. Requirements enable the management of risks from the earliest possible point in development. Risks raised against requirements can be tracked, their impact assessed and the effects of mitigation and fallback plans understood long before substantial development costs have been incurred.

Requirements therefore form the basis for:

- project planning;
- risk management;
- acceptance testing;
- tradeoff;
- change control.

The most common reasons for project failures are not technical. A report created by the Standish Group based on a survey identified that incomplete requirements and the lack of user involvement are primary reasons for project failures. Other reasons include lack of resources, unrealistic expectations, lack of executive support, changing requirements, lack of planning, etc. More than half of those reasons (even the most influential two) are directly related to requirements.

The problems fall into three main categories:

- Requirements—either poorly organized, poorly expressed, weakly related to stakeholders, changing too rapidly or unnecessary; unrealistic expectations.

- Management problems of resources—failure to have enough money, and lack of support or failure to impose proper discipline and planning; many of these arise from poor requirements control.

- Politics—which contributes to the first two problems.

All these factors can be addressed at fairly low cost.

Since requirements are interconnected with other aspects of systems engineering and project management, it is quite challenging to find a satisfactory scope for a definition of requirements engineering.

First of all, what is a requirement? *IEEE Standard for Application and Management of the Systems Engineering Process* (IEEE Std 1220-2005) gives the following definition:
**Requirement:** a statement that identifies a product or process operational, functional, or design characteristic or constraint, which is unambiguous, testable or measurable, and necessary for product or process acceptability (by consumers or internal quality assurance guidelines).

Let us take a short look on the elements of this definition:

• **Statement.** That a requirement should be a statement is perhaps biased towards textual description, while they can also be captured in tabular form or in diagrammatic form (especially when using such a modeling language like UML), in formal notations, or even in domain-specific notations. The important concept, though, is to have a set of traceable, manageable elements identified as requirements.

• **Product or process.** Complete solutions contain varying mixtures of product (things that are built in response to requirements) and process (procedures for using the things that are built). Requirements may therefore define process as well as product. In addition to this, there may be requirements that constrain the way how the product should be developed, usually for quality control purposes.

• **Operational, functional, or design characteristic or constraint.** There are many different kinds of requirement, giving rise to different kinds of language, analysis, modelling, process and solution. Note that this definition has carefully avoided the term “non-functional”, because there is a debate about what this actually means. Design characteristics cover performance, usability, safety, maintainability and a host of other qualities.

• **Unambiguous.** A statement of requirement has desirable qualities that will be addressed in detail later. In brief, a requirement should lend itself to a clear, single understanding, common to all parties involved.

• **Testable or measurable.** Requirements are used to test that the design or solution is acceptable. For this to be possible, the requirement should be quantified, thus providing a means of “measuring” the solution against it.

• **Necessary for product or process acceptability.** Requirements play a multi-dimensional role: on one hand, they define what should be designed and developed, and, on the other hand, they also define how the solution should be tested and accepted. So they have an influence in the earliest stages of the development process as well as in the latest stages during acceptance.

• **By consumers or internal quality assurance guidelines.** Requirements come from many sources, including but not limited to customers, regulatory bodies, users and internal quality procedures.

Some other synonyms for requirements are: aims, aspirations, capabilities, criteria, constraints, directives, doctrines, duties, expectations, features, functions, goals, missions, needs, obligations, objectives, orders, regulations, rules, etc.

The term “stakeholder” has already been used without giving a definition:

**Stakeholder:** An individual, group of people, organisation or other entity that has a direct or indirect interest (or stake) in a system.

A stakeholder’s interest in a system may arise from using the system, benefiting from the system (in terms of revenue or other advantage), being disadvantaged by the system (in terms, for instance, of cost or potential harm), being responsible for the system, or otherwise being affected by it. Stakeholders are legitimate sources of requirements.

According to [HULL2010], requirements engineering is the subset of systems engineering concerned with discovering, developing, tracing, analyzing, qualifying, communicating and managing requirements that define the system at successive levels of abstraction.

This definition lists carefully selected key activities that are considered proper to requirements engineering. There are some activities closely related to requirements that are considered to be part of some other discipline. An example of this is system testing or verification; while requirements should have the qualities needed to ensure that the solution can be verified, the verification activity itself is another discipline. It also references the concept of requirements existing at various levels of development. Here are some notes on the definition:

• **Discovering.** This covers a number of terms often used, such as requirements elicitation and capture.
• **Tracing.** Tracing of requirements to other artifacts, including requirements at other layers, provides a means of validating requirements against real-world needs, of capturing rationale for the design, and of verifying the design against requirements.

• **Qualifying.** This refers to all kinds of testing activity, covering testing of the design and solution, including unit, component, integration, system, acceptance testing. There is considerable disagreement over the meaning of the terms “verification” and “validation”. The term “qualifying” is preferred, because it is about ensuring that the solution has the required “qualities.” In so much as the terms are used in this book, to validate requirements is to check a formal expression of requirements against informal needs as understood in the minds of stakeholders, and to verify requirements is to check their internal consistency within layers and between layers of abstraction.

• **Communicating.** Requirements are the means of communication through which customers, suppliers, developers, users and regulators can agree on what is to be achieved.

• **Levels of abstraction.** This is about the practice of organizing requirements into layers and tracing the satisfaction relationship between those layers. The requirements of the top layer define the system in terms of the problems to be solved as agreed by the stakeholders and validated against their real needs; requirements at subsequent layers define the whole or part of the system in terms of an implementable solution as verified against the requirements at the layer above; requirements at every layer provide a precise means of qualifying the solution. Some people refer to the relationship between requirements induced by recording satisfaction between layers as a requirements hierarchy, but in reality the many-to-many relationship forms a graph or heterarchy.

### 1. Classification of requirements

According to [SOMMERVILLE2010], requirements can be classified based on various viewpoints. A popular way of classifying requirements split them based on the level which is bound to how they communicate information about the system to different types of users. Different types of requirements are defined for different audiences. We can distinguish between these levels by using the term ‘user requirements’ to mean the high-level abstract requirements and ‘system requirements’ to mean the detailed description of what the system should do. User requirements and system requirements may be defined as follows:

1. **User requirements** are statements, in a natural language plus diagrams, of what services the system is expected to provide to system users and the constraints under which it must operate.

2. **System requirements** are more detailed descriptions of the software system’s functions, services, and operational constraints. The system requirements document (sometimes called a functional specification) should define exactly what is to be implemented. It may be part of the contract between the system buyer and the software developers.

Requirements should be written at different levels of detail because different readers use them in different ways. Figure Readers of different types of requirements specification [SOMMERVILLE2010] shows possible readers of the user and system requirements. The readers of the user requirements are not usually concerned with how the system will be implemented and may be managers who are not interested in the detailed facilities of the system. The readers of the system requirements need to know more precisely what the system will do because they are concerned with how it will support the business processes or because they are involved in the system implementation.

**Figure 2.1. Readers of different types of requirements specification [SOMMERVILLE2010]**
Software system requirements are often classified as functional requirements or nonfunctional requirements:

1. **Functional requirements** are statements of services the system should provide, how the system should react to particular inputs, and how the system should behave in particular situations. In some cases, the functional requirements may also explicitly state what the system should not do.

2. **Non-functional requirements** are constraints on the services or functions offered by the system. They include timing constraints, constraints on the development process, and constraints imposed by standards. Non-functional requirements often apply to the system as a whole, rather than individual system features or services.

However, the distinction between different types of requirements is not so clear as these simple definitions suggest. A user requirement concerned with security, such as a statement limiting access to authorized users, may appear to be a nonfunctional requirement. However, when developed in more detail, this requirement may generate other requirements that are clearly functional, such as the need to include user authentication facilities in the system.

This shows that requirements are not independent and that one requirement often generates or constrains other requirements. The system requirements therefore do not just specify the services or the features of the system that are required; they also specify the necessary functionality to ensure that these services/features are delivered properly.

The functional requirements for a system describe what the system should do. These requirements highly depend on the type of software being developed, the expected users, and the approach that is applied by the organization responsible for writing requirements together. When expressed as user requirements, functional requirements are usually described in an abstract way that can be understood by system users. However, more specific functional system requirements describe the system functions, its inputs and outputs, exceptions, etc., in detail.

Non-functional requirements, as the name suggests, are requirements that are not directly concerned with the specific services delivered by the system to its users. They may relate to emergent system properties such as reliability, response time, and store occupancy. Alternatively, they may define constraints on the system implementation such as the capabilities of I/O devices or the data representations used in interfaces with other systems.

Non-functional requirements (or, design characteristics), such as performance, security, or availability, usually specify or constrain characteristics of the system as a whole. Non-functional requirements are often more critical than individual functional requirements. System users can usually find some workarounds for a system function that does not really meet their expectations. However, failing to meet a non-functional requirement can mean that the whole system is unusable. For example, if an aircraft system does not meet its reliability requirements, it will not be certified as safe for operation; if an embedded control system fails to meet its performance requirements, the control functions will not operate correctly.

Although it is often possible to identify which system components implement specific functional requirements (e.g., there may be formatting components that implement reporting requirements), it is often more difficult to relate components to non-functional requirements. The implementation of these requirements may be diffused throughout the system. There are two reasons for this:

1. Non-functional requirements may affect the overall architecture of a system rather than the individual components. For example, to ensure that performance requirements are met, you may have to organize the system to minimize communications between components.
2. A single non-functional requirement, such as a security requirement, may generate a number of related functional requirements that define new system services that are required. In addition, it may also generate requirements that restrict existing requirements.

Non-functional requirements arise through user needs, because of budget constraints, organizational policies, the need for interoperability with other software or hardware systems, or external factors such as safety regulations or privacy legislation. Figure Types of non-functional requirements [SOMMERVILLE2010] shows a classification of non-functional requirements. You can see from this diagram that the non-functional requirements may come from required characteristics of the software (product requirements), the organization developing the software (organizational requirements), or from external sources:

**Figure 2.2. Types of non-functional requirements [SOMMERVILLE2010]**

1. *Product requirements.* These requirements specify or constrain the behavior of the software. Examples include performance requirements on how fast the system must execute and how much memory it requires, reliability requirements that set out the acceptable failure rate, security requirements, and usability requirements.

2. *Organizational requirements.* These requirements are broad system requirements derived from policies and procedures in the customer’s and developer’s organization. Examples include operational process requirements that define how the system will be used, development process requirements that specify the programming language, the development environment or process standards to be used, and environmental requirements that specify the operating environment of the system.

3. *External requirements.* This broad heading covers all requirements that are derived from factors external to the system and its development process. These may include regulatory requirements that set out what must be done for the system to be approved for use by a regulator, such as a central bank; legislative requirements that must be followed to ensure that the system operates within the law; and ethical requirements that ensure that the system will be acceptable to its users and the general public.

**2. The Software Requirements Specification document**

The software requirements specification (SRS) is an official statement of what the system developers should implement. It should include both the user requirements for a system and a detailed specification of the system requirements. In many cases, the user and system requirements are integrated into a single description. In other cases, the user requirements are defined in an introduction to the system requirements specification. As the number of requirements grow, it is more and more common to have the detailed system requirements presented in a separate document.

Requirements documents are essential when an outside contractor is developing the software system. However, some of the software development methodologies (i.e., agile methods) argue that requirements change so quickly that a requirements document is out of date as soon as it is written, therefore the effort spent with its
preparation is largely wasted. Rather than a formal document, approaches such as Extreme Programming (XP) or Scrum collect user requirements incrementally and write these on cards as so-called user stories. The user then prioritizes requirements for implementation in the next increment of the system.

For business systems where requirements are unstable, this is a viable approach. However, in our opinion, it is still useful to write a short supporting document that defines the business and dependability requirements for the system; it is easy to forget the requirements that apply to the system as a whole when focusing on the functional requirements for the next system release.

The requirements document has a diverse set of users (stakeholders), ranging from the senior management of the organization that is paying for the system to the engineers responsible for developing the software. Figure Users of a requirements document [SOMMERVILLE2010] shows possible users of the document and how they use it.

**Figure 2.3. Users of a requirements document [SOMMERVILLE2010]**

2.1. When is an SRS said to be good?

The IEEE Recommended Practice for Software Requirements Specifications ([IEEE830:1998]) discusses a number of issues that should be considered during the creation of an SRS. Here we only give a short overview of those considerations.

The SRS is a specification for a particular software product, program, or set of programs that performs certain functions in a specific environment. The SRS may be written by one or more representatives of the supplier, one or more representatives of the customer, or by both. The recommendation is that it should be prepared by both, together.

The basic issues that the SRS writer(s) shall address are the following:

a. **Functionality.** What is the software supposed to do?

b. **External interfaces.** How does the software interact with people, the system’s hardware, other hardware, and other software?

c. **Performance.** What is the speed, availability, response time, recovery time of various software functions, etc.?

d. **Attributes.** What are the portability, correctness, maintainability, security, etc. considerations?

e. **Design constraints imposed on an implementation.** Are there any required standards in effect, implementation language, policies for database integrity, resource limits, operating environment(s) etc.?

The SRS writer(s) should avoid placing either design or project requirements in the SRS.
An SRS has a specific role to play in the software development process, i.e., it plays a well-defined role in the whole project. It is a part of the total project plan and it should also conform to some higher-level documents like the System Requirements Specification which describes the requirements of a computer system (not only the software part). As a consequence, SRS writer(s) should be careful not to go beyond the bounds of that role. This means the SRS

a. Should correctly define all of the software requirements. A software requirement may exist because of the nature of the task to be solved or because of a special characteristic of the project.

b. Should not describe any design or implementation details. These should be described in the design stage of the project.

c. Should not impose additional constraints on the software. These are properly specified in other documents such as a software quality assurance plan.

Therefore, a properly written SRS limits the range of valid designs, but does not specify any particular design.

The Recommendation identifies 8 important characteristics which a good SRS should have. These are the following:

1. Correctness. An SRS is correct if, and only if, every requirement stated therein is one that the software shall meet. However, generally speaking there is no tool or procedure to ensure correctness. That’s why the SRS should be compared to superior documents (including the System Requirements Specification, if exists) during a review process in order to filter out possible contradictions and inconsistencies. Reviews should also be used to get a feedback from the customer side on whether the SRS correctly reflects the actual needs. This process can be made easier and less error-prone by traceability.

2. Unambiguity. An SRS is unambiguous if, and only if, every requirement stated therein has only one interpretation. As a minimum, this requires that each characteristic of the final product be described using a single unique term. In cases where a term used in a particular context could have multiple meanings, the term should be included in a glossary where its meaning is made more specific.

3. Completeness. An SRS is complete if, and only if, it includes the following elements:
   a. All significant requirements, whether relating to functionality, performance, design constraints, attributes, or external interfaces. In particular, any external requirements imposed by a system specification should be acknowledged and treated.
   b. Definition of the responses of the software to all realizable classes of input data in all realizable classes of situations. Note that it is important to specify the responses to both valid and invalid input values.
   c. Full labels and references to all figures, tables, and diagrams in the SRS and definition of all terms and units of measure.

However, completeness is very hard to achieve, especially when talking about business systems where the requirements always change and new requirements raise. Completeness is something that agile methodologies (which think that there is a continuous change in the requirements) cannot interpret. On the other hand, when following a plan-driven methodology, completeness might make sense, but probably in a bit modified manner: all those requirements should be described in the SRS which are known by the actual time. It also requires updating of the document if some new requirements emerge.

4. Consistency. Consistency refers to internal consistency. If an SRS does not agree with some higher-level document, such as a system requirements specification, then it is a violation of correctness. An SRS is internally consistent if, and only if, no subset of individual requirements described in it conflict. The three types of likely conflicts in an SRS are as follows:
   a. The specified characteristics of real-world objects may conflict. For example, the format of an output report may be described in one requirement as tabular but in another as textual or one requirement may state that all lights shall be green while another may state that all lights shall be blue.
   b. There may be logical or temporal conflict between two specified actions. For example, one requirement may specify that the program will add two inputs and another may specify that the program will multiply
them or one requirement may state that “A” must always follow “B”, while another may require that “A and B” occur simultaneously.

c. Two or more requirements may describe the same real-world object but use different terms for that object. For example, a program’s request for a user input may be called a “prompt” in one requirement and a “cue” in another. The use of standard terminology and definitions promotes consistency.

5. Ranking. An SRS is ranked for importance and/or stability if each requirement in it has an identifier to indicate either the importance or stability of that particular requirement. Typically, all of the requirements that relate to a software product are not equally important. Some requirements may be essential, especially for life-critical applications, while others may be desirable. There are many possibilities of ranking requirements, however, most notably we use ranking based on the degree of stability and/or degree of necessity.

   a. Stability can be expressed in terms of the number of expected changes to any requirement based on experience or knowledge of forthcoming events that affect the organization, functions, and people supported by the software system.

   b. Based on necessity, one can identify classes of requirements as essential, conditional, and optional. Essential requirements are those that imply that the software will not be acceptable unless these requirements are provided in an agreed manner. Conditional requirements imply that these are requirements that would enhance the software product, but would not make it unacceptable if they are absent. Last but not least, optional requirements imply a class of functions that may or may not be worthwhile. This gives the supplier the opportunity to propose something that exceeds the SRS.

6. Verifiability. An SRS is verifiable if, and only if, every requirement stated therein is verifiable. A requirement is verifiable if, and only if, there exists some finite cost-effective process with which a person or machine can check that the software product meets the requirement. In general any ambiguous requirement is nonverifiable. Nonverifiable requirements include statements such as “works well”, “good human interface”, and “shall usually happen”. These requirements cannot be verified because it is impossible to define the terms “good”, “well”, or “usually”. The statement that “the program shall never enter an infinite loop” is also nonverifiable because the testing of this quality is theoretically impossible. An example of a verifiable statement is “Output of the program shall be produced within 20 s of event x 60% of the time; and shall be produced within 30 s of event x 100% of the time”. This statement can be verified because it uses concrete terms and measurable quantities. If a method cannot be devised to determine whether the software meets a particular requirement, then that requirement should be removed or revised.

7. Modifiability. An SRS is modifiable if, and only if, its structure and style are such that any changes to the requirements can be made easily, completely, and consistently while retaining the structure and style. Modifiability generally requires an SRS to

   a. Have a coherent and easy-to-use organization with a table of contents, an index, and explicit crossreferencing;

   b. Not be redundant (i.e., the same requirement should not appear in more than one place in the SRS);

   c. Express each requirement separately, rather than intermixed with other requirements.

Redundancy itself is not an error, but it can easily lead to errors. Redundancy can occasionally help to make an SRS more readable, but a problem can arise when the redundant document is updated. For instance, a requirement may be altered in only one of the places where it appears. The SRS then becomes inconsistent. Whenever redundancy is necessary, the SRS should include explicit cross-references to make it modifiable.

8. Traceability. An SRS is traceable if the origin of each of its requirements is clear and if it facilitates the referencing of each requirement in future development or enhancement documentation. The following two types of traceability are recommended:

   a. Backward traceability (i.e., to previous stages of development). This depends upon each requirement explicitly referencing its source in earlier documents.

   b. Forward traceability (i.e., to all documents spawned by the SRS). This depends upon each requirement in the SRS having a unique name or reference number.
The forward traceability of the SRS is especially important when the software product enters the operation and maintenance phase. As code and design documents are modified, it is essential to be able to ascertain the complete set of requirements that may be affected by those modifications.

It is also important to note that the SRS may need to evolve as the development of the software product progresses. It may be impossible to specify some details at the time the project is initiated (e.g., it may be impossible to define all of the screen formats for an interactive program during the requirements phase). Additional changes may ensue as deficiencies, shortcomings, and inaccuracies are discovered in the SRS. Two major considerations in this process are the following:

a. Requirements should be specified as completely and thoroughly as is known at the time, even if evolutionary revisions can be foreseen as inevitable. The fact that they are incomplete should be noted.

b. A formal change process should be initiated to identify, control, track, and report projected changes.

Approved changes in requirements should be incorporated in the SRS in such a way as to

a. Provide an accurate and complete audit trail of changes;

b. Permit the review of current and superseded portions of the SRS.

A requirement specifies an externally visible function or attribute of a system. A design describes a particular subcomponent of a system and/or its interfaces with other subcomponents. The SRS writer(s) should clearly distinguish between identifying required design constraints and projecting a specific design. Note that every requirement in the SRS limits design alternatives. This does not mean, though, that every requirement is design. The SRS should specify what functions are to be performed on what data to produce what results at what location for whom. The SRS should focus on the services to be performed. The SRS should not normally specify design items such as the following:

a. Partitioning the software into modules;

b. Allocating functions to the modules;

c. Describing the flow of information or control between modules;

d. Choosing data structures.

However, in special cases, some requirements may severely restrict the design. For example, security or safety requirements may reflect directly into design such as the need to

a. Keep certain functions in separate modules;

b. Permit only limited communication between some areas of the program;

c. Check data integrity for critical variables.

Examples of valid design constraints are physical requirements, performance requirements, software development standards, and software quality assurance standards.

Therefore, the requirements should be stated from a purely external viewpoint. When using models to illustrate the requirements, remember that the model only indicates the external behavior, and does not specify a design.

### 2.2. Parts of an SRS

The *IEEE Recommended Practice for Software Requirements Specifications* (IEEE 830) document proposes a structure for SRS documents (see Figure Recommended structure for SRS documents). As this is a recommendation, companies are welcome to adapt, tune and tailor this structure to their actual needs. This means that in the most of the cases, organizational standards are developed on the basis of this quasi-standard. While an SRS does not have to follow this outline or use the names given here for its parts, a good SRS should include all the information discussed here.
3. Gathering the requirements of the case study

User requirements are determined first. They are written to the SRS and serve as a basis of analysis.

Here we start with a short overview of the application to be developed. Based on this description, more detailed requirements should be developed.

**Book & DVD Store outline**

The Book & DVD Store is an online warehouse. The purpose of the system is to make it possible for the customers to browse and search products (i.e. books and DVDs) and to place orders for these products online. Customers can choose their preferred payment method; new payment methods should be easy to add. The organization operating the online store does not have its own delivery solution therefore a courier company like FedEx or DHL will perform the delivery to the customer. All courier companies have their own software system which the online bookstore needs to exchange data with in order to provide information for the customers about the shipment of their newly acquired property if requested. The ordering system is responsible for issuing invoices but there is a separate accounting system at the organization which will receive detailed information on each sale.

The most important stakeholders along with a short outline of their most important functionalities are the following:

1. **User:** User of the system in three different roles: Customer, Administrator, Manager.
2. Customer:

2.1. Browse and search for products. Products can be either books or DVDs (films).

2.1.1. Browse based on the most frequently used categories. (Therefore, the system has to keep track of browsing activities to be able to determine the most frequently used categories by other customers.)

2.1.2. Browse based on categories. Each product has an associated category. Categories can be organized into forming hierarchies (so categories might have subcategories).

2.1.3. Browse among similar products. The system should ensure to identify similar products based on certain business rules.

2.1.4. In case of browsing or after a search process, the result screen should contain brief description about the product. The brief description has to contain some pieces of information to identify the underlying product (e.g., title, year of publishing, author/starring.)

2.1.5. Search by the most common attributes of a product (simple search).

2.1.6. Search by specific attributes of a product (advanced search).

2.1.7. Refine search result. Filtering the result items.

2.1.8. View product details.

2.2. Manage bookshelf and shopping cart.

2.2.1. Save selected products onto a bookshelf.

2.2.2. Delete selected products from the bookshelf.

2.2.3. Create own categories for the saved items on the bookshelf in order to be able to maintain certain products together.

2.2.4. Define and save attributes for own categories.

2.2.5. Browse/search among own categories.

2.2.6. Delete own categories.

2.2.7. Add item(s) to own categories.

2.2.8. Delete item(s) from own categories.

2.2.9. Move item(s)/item group(s) to shopping cart.

2.2.10. Remove item(s) from shopping cart.

2.2.11. Move item(s) to bookshelf from shopping cart.

2.2.12. Put selected item(s) directly into the shopping cart.

2.2.13. Save the shopping cart.

2.3. Manage orders.

2.3.1. Place an order based on the content of the shopping cart.

2.3.2. Finalize order.

2.3.3. Cancel order.

2.3.4. Get email notification about the order (with certain details).
2.3.5. Track order status.

2.3.6. Set rules on automatic orders. The system should send an email notification before placing an automatic order. Customer can finalize or cancel it.

2.3.7. Provide feedback about the products and/or the ordering process.

2.4. Perform other activities.

2.4.1. Visit forums about products and the shop.

3. **Administrator**:

3.1. Manage products.

3.1.1. Save/Record new products into the system.

3.1.2. Remove products from the system.

3.1.3. Modify products (product attributes).

3.1.4. Import products.

3.1.5. Export products.

3.2. Manage orders.

3.2.1. Update order status.

3.2.2. Process feedbacks.

3.3. Compose/Send newsletters.

3.4. Moderate forums.

4. **Manager**:

4.1. Manage rules for special offers.

4.1.1. Set categories on handling special customers. (Gold customer, diamond customer, etc.)

4.1.2. Modify categories on handling special customers.

4.1.3. Set special customer discounts.

4.1.4. Modify special customer discounts.

4.1.5. Set special action offers (for product categories).

4.1.6. Modify special action offers.

4.2. The system should generate/present reports/statistics about sales data.

These are high-level requirements of the system which are also called user requirements. After gathering them, an analysis process should be performed in order to develop a deeper understanding of the problem domain. During analysis, basic building blocks of the functionalities are identified and described as system requirements. The main difference between system and user requirements are that while the latter describes expectations using domain-specific concepts mainly, the former uses computer science terminology. During analysis, domain-specific concepts are "translated" onto a more specific language that various kinds of development team members (architects, programmers, testers, etc.) understand.

A user requirement might be described by several system requirements. An example for that is adding a new product to an existing order: it requires a lookup (or a form-based search) among existing products and the total price of the order should be re-calculated. Different user requirements might result in the same system.
requirement after analysis like in the case of sending a newsletter and sending a registration confirmation email: from the system's perspective, these two functionalities are quite the same (send an e-mail to a given address), only the text of e-mails will vary.

As it has been noted earlier, Unambiguity. [22] can be achieved by providing a glossary that defines the terms used in the specification. This is extremely important in such cases when some terms have a meaning in the vulgar tongue but we would like to use the same term in a more constraining manner. Using a glossary, terms can be precisely defined and then used in a well-defined and consistent manner throughout the whole documentation of the software. The glossary should be described in Section 1.3 of the SRS (see Figure 1.1 Recommended structure for SRS documents).

**Glossary**

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>User is an authenticated person who can place orders in the system. Browsing and searching operations do not need authentication. User of the system can be either Customer or Administrator or Manager.</td>
</tr>
<tr>
<td>Customer</td>
<td>Specific user of the system. Customer can perform browsing/searching among products and place an order.</td>
</tr>
<tr>
<td>Administrator</td>
<td>Specific user of the system. Administrator can maintain products or attributes of a product. He/she can modify the status of orders if necessary. User account management (including locking and unlocking users) also belongs to his/her responsibilities.</td>
</tr>
<tr>
<td>Manager</td>
<td>Specific administrator of the system. Managers make decisions about special action offers and discounts besides the activities they can perform as administrators.</td>
</tr>
<tr>
<td>Product</td>
<td>The purpose of the system is to sale products that can be either book or DVD.</td>
</tr>
<tr>
<td>Item</td>
<td>After selecting and moving to bookshelf or to shopping cart, a product becomes an item. Items can have additional attributes compared to products.</td>
</tr>
<tr>
<td>Category</td>
<td>Differentiating among products/items based on their one or more certain attributes. Products/items belong to given categories. A category contains similar products/items. Categories are hierarchical – a category may built upon one or more subcategories.</td>
</tr>
<tr>
<td>Item group</td>
<td>An item group can contain one or more items. Customer can create item groups to be able to add further attributes to one or more items. (These items can be (further) categorized on a refined way then can be handled together.)</td>
</tr>
<tr>
<td>Bookshelf</td>
<td>Customer’s bookshelf may contain one or more items or item groups. Bookshelf and its contents are saved automatically. Managing bookshelf requires authentication. One customer can have one bookshelf only.</td>
</tr>
<tr>
<td>Shopping cart</td>
<td>Customer’s shopping cart may contain one or more items. Based on the content of the shopping cart, customer can place an order. The order will contain items from the shopping cart. If the order is finalized by customer, the shopping cart will be erased. One customer can have one shopping cart in one time.</td>
</tr>
<tr>
<td>Order status</td>
<td>Each order has a status that describes different stages of the order in the ordering process. (E.g. &quot;new&quot;, &quot;packed&quot;, &quot;delivered&quot;, etc.) Customer can check the status of his order. Status can be changed by administrator. If an order is delivered, the system can close it and it will be no more relevant for the system.</td>
</tr>
</tbody>
</table>
Order tracking
Customer can track the stage of his order in the ordering process.
See Also Order status.

Discount
Rate of price lowering. Discounts can be product-based or customer-based.

Product-based discount
Rate of price lowering determined by product category.

Customer-based discount
Rate of price lowering determined by customer category.

Customer category
Differentiating among customers based on former orders. The higher category customer belongs to, the greater discounts (the lower prices) are assigned to the customer’s order.

Action offer
Special offer for one or more products that determines the extent/degree of discounts.

When a need for a new software emerges, the person (or people) responsible for gathering the requirements (based on the size of the software development organization and many other factors they can be business analysts (BA) or system architects) should consult with the future customers or end users of the system. They have imaginations and expectations about how they would like to make use of the system, about how the system should work.

For our case study (as we would like to keep our example simple), we do not have any opportunity to consult with end users, therefore, we will determine what functionalities a real customer would expect. Let us imagine that we are the end users of our software system to be developed. What would our expectations be against such a system? How would we like to use it? What functionalities do we need?

The system of our study is an online book and DVD store. What are the main scenarios that various system users might execute? We (as customers of an online shop) would like to browse among products. For example, the system should present us a list of products. We can scroll up and down and make selections. We would also like to perform various search operations, if we have in idea about one or more attributes or characteristics of the product. If the search result contains too many items, we would like to refine the search. If some more information needed about a certain product, we would like to view its details. If some of the products are fancy to us, we would like to save them to our own bookshelf or, rather, we would like to buy them. We would like to be able to place an order, confirm an order or cancel it. We would like to receive offers on discounts and be notified on some interesting titles via email. We would like to share our experiences about certain products, or we are curious about the people's opinion on certain items. Last but not least, we would like to give feedback on either the product or the process how we get it.

Now, we have all the customers’ imaginations and expectations in a nutshell. The system, however, will not work without any support. Under such a support, we mean someone who decides about products to be managed and sold by the system, someone who makes decisions about product prices and various kinds of discounts, someone who keeps the customers informed about action offers, someone who maintains products, someone who deals with customer relationship.

Besides customers, we identified two other groups of users from different aspects. One of them is the manager who makes business decisions. The other one is the administrator who should maintain the system itself and also the most important system elements besides customers—namely, the products for sale.

Documenting requirements using UML: use case diagrams

In the UML, use cases capture the requirements of a system (at least, for functional requirements). Use cases are a means of communicating with users and other stakeholders what the system is intended to do. A use case is a kind of behaviored classifier that specifies a complete unit of useful functionality performed by one or more subjects to which the use case applies in collaboration with one or more actors, and which (for complete use cases) yields an observable result that is of some value to those actors (or probably to other stakeholders) of each subject [UML-DIAGRAMS.ORG].
An actor is behaviored classifier which specifies a role played by an external entity that interacts with the system (e.g., by exchanging signals and data), a human user of the designed system, some other system or hardware using services of the system. An actor is always considered to be external to the system.

The term "role" is used informally as some type, group or particular facet of users that require specific services from the subject modeled with associated use cases. When an external entity interacts with the subject, it plays the role of a specific actor. That single physical entity may play several different roles, and a specific role may be played by single or multiple different instances.

All actors must have names according to the assumed role. Examples of actor names (user roles):

- Customer
- Manager
- Shipping System

The standard UML notation for actors is a "stick man" with the name of the actor above or (preferably) below of the icon.

**Figure 2.5. Actor Customer**

Actors can participate in generalization relationships with other actors. Generalization is rendered as a solid directed line with a large arrowhead (same as for generalization between classes). Actors can also be abstract. The names of abstract actors should be shown in italics. In the following figure, User is an abstract use case.

**Figure 2.6. Generalization relationship between actors**

Use cases are most frequently identified using a scenario-based requirements elicitation technique (at least, this is true for object-oriented systems). In this context, a use case is a single unit of meaningful
work. It provides a high-level view of behavior observable to someone or something outside the system. The notation for a use case is an ellipse.

**Figure 2.7. Use case example**

![Use case example diagram](image)

A use case definition typically includes the following parts:

- name and description,
- requirements,
- constraints, and
- scenarios.

Use cases are normally named as a verb phase. It is a good practice to give some name to that unit of functionality performed by a system which provides some observable and useful result to an actor. Example of use case names are: Make purchase, Place order, Manage Account, etc. A brief, informal textual description should also be provided along with the use case.

The requirements define the formal functional requirements that a use case must supply to the end user (or, to be more precise, the actor). A requirement is a contract or promise that the use case will perform an action or provide some value to the system.

These contracts are described by using constraints. Constraints are conditions or restrictions that a use case operates under and include pre-, post- and invariant conditions. A precondition specifies the conditions that need to be met before the use case can proceed. A post-condition is used to document the change in conditions that must be true after the execution of the use case. An invariant condition specifies the conditions that are true throughout the execution of the use case.

A scenario is a formal description of the flow of events that occur during the execution of a use case instance. It defines the specific sequence of events between the system and the external actors. It is normally described in text and corresponds to the textual representation of the sequence diagram.

Use cases can be related to each other. Three basic types of use case relationships are defined: as being a (behaviored) classifier, a generalization relationship can be understood such like between classes—child use case inherits properties and behavior of the parent use case and may override the behavior of the parent. Generalization is rendered as a solid directed line with a large arrowhead, the same as for generalization between any other kinds of classifiers.

Extend is a directed relationship that specifies how and when the behavior defined in usually supplementary (optional) extending use case can be inserted into the behavior defined in the extended use case. An extended use case is meaningful on its own, it is independent of the extending use case. The extending use case typically defines optional behavior that is not necessarily meaningful by itself. The extend relationship is owned by the extending use case. The same extending use case can extend more than one use case, and extending use case may itself be extended. The extension takes place at one or more extension points defined in the extended use case. Extend relationship is shown as a dashed line with an open arrowhead directed from the extending use case to the extended (base) use case. The arrow is labeled with the stereotype «extend».

**Figure 2.8. An «extend» relationship between use cases.**
An extension point is a feature of a use case which identifies (references) a point in the behavior of the use case where that behavior can be extended by some other (extending) use case, as specified by extend relationship. Extension points may be shown in a compartment of the use case oval symbol under the heading extension points. Each extension point must have a name, unique within a use case. Extension points are shown as a text string according to the syntax:

```
extension point ::= name [: explanation ]
```

The optional explanation is some description usually given as informal text (see Figure 2.9, “Sample usage of extension points.”). It could be in other forms, such as the name of a state in a state machine, an activity in an activity diagram, some precondition or postcondition.

**Figure 2.9. Sample usage of extension points.**

Use case include is a directed relationship between two use cases which is used to show that behavior of the included use case (the addition) is inserted into the behavior of the including (the base) use case. The include relationship could be used:

a. to simplify large use case by splitting it into several use cases,

b. to extract common parts of the behaviors of two or more use cases.

A large use case could have some behaviors which might be detached into distinct smaller use cases to be included back into the base use case using the UML include relationship. The purpose of this action is modularization of behaviors, making them more manageable.

**Figure 2.10. Use cases B and C are extracted from larger use case A into separate use cases ([UML-DIAGRAMS.ORG]).**

When two or more use cases have some common behavior, this common part could be extracted into a separate use case to be included back by the use cases with the UML include relationship.

**Figure 2.11. Use case C is extracted from use cases A and B to be reused by both use cases using UML include relationship ([UML-DIAGRAMS.ORG]).**
Execution of the included use case is analogous to a subroutine call or macro command in programming. All of the behavior of the included use case is executed at a single location in the including use case before execution of the including use case is resumed. Including use case depends on the addition of the included use case, which is required and not optional. Include relationship between use cases is shown by a dashed arrow with an open arrowhead from the including (base) use case to the included (common part) use case. The arrow is labeled with the stereotype «include».

**Figure 2.12. A use case including several use cases.**

### 4. Analysis

After collecting the overall requirements, an analysis phase takes place. During that part, not only the identified requirements will be elaborated but this is the stage when some preliminary modeling also takes place. The goal of the analysis activities is to establish a deep(er) understanding of the problem domain. It includes finding the details of the outline requirements and developing some system models that describe the system's context, structure, behavior and interactions. These models develop in parallel as newer and newer parts of the system are understood.

Now, let us return to the role of the BA and translate the former outline into a real system requirements specification that will be an important phase in system development before system modeling and designing the architecture. Documents describing the requirements and design are not text-only documents but the natural language description of the system is supplemented by some figures. In the context of our case study, UML diagrams will predominantly be used for system descriptions.

As it has been discussed above, use case diagram of the UML can portray different types of system users and the ways that they interact with the system. This type of diagram is typically used in conjunction with the textual use case and will often be accompanied by other types of diagrams as well. It covers functional requirements against the system almost entirely. Previously, we have identified three main user types: customer, administrator, and manager. They will emerge as actors in the use case diagram. We can also define relationships between actors. In most of the cases, these are generalization/specialization relationships. In our example, the most general actor is the user whose descendants are the customer and the administrator. Since the manager owns all attributes like an administrator but also additional ones, the manager actor specializes the administrator (i.e., the manager is descendant of the administrator).
After creating the specification, stakeholders should be consulted again in order to validate its contents. In our case, we make a revision of the diagram(s) ourselves. If it is finished, we obtain an overview of the structure of the system. Now, we know what the system should do for its users.

The following table shows a generic structure of documenting a single use case. This is what is called a structured natural language description of scenarios. We will use it as a template for documenting use cases in Sections 4.1, “Customer use cases” — Section 4.3, “Manager use cases”.

<table>
<thead>
<tr>
<th>Use case ID</th>
<th>Enter a unique numeric identifier for the Use Case. e.g. UC-1.2.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use case name</td>
<td>Enter a short name for the Use Case using an active verb phrase. e.g. Withdraw Cash</td>
</tr>
<tr>
<td>Actors</td>
<td>[An actor is a person or other entity external to the software system being specified who interacts with the system and performs use cases to accomplish tasks. Different actors often correspond to different user classes, or roles, identified from the customer community that will use the product. Name the actor that will be initiating this use case (primary) and any other actors who will participate in completing the use case (secondary).]</td>
</tr>
<tr>
<td>Description</td>
<td>[Provide a brief description of the reason for and outcome of this use case.]</td>
</tr>
<tr>
<td>Trigger</td>
<td>[Identify the event that initiates the use case. This could be an external business event or system event that causes the use case to begin, or it could be the first step in the normal flow.]</td>
</tr>
<tr>
<td>Precondition</td>
<td>[List any activities that must take place, or any conditions that must be true, before the use case can be started. Number each pre-condition. e.g.</td>
</tr>
<tr>
<td></td>
<td>1. Customer has active deposit account with ATM privileges</td>
</tr>
<tr>
<td></td>
<td>2. Customer has an activated ATM card.]</td>
</tr>
<tr>
<td>Postcondition</td>
<td>[Describe the state of the system at the conclusion of the use case execution. Should include both minimal guarantees (what must happen even if the actor’s goal is not achieved) and the success guarantees (what happens when the actor’s goal is achieved. Number each post-condition. e.g.</td>
</tr>
<tr>
<td></td>
<td>1. Customer receives cash</td>
</tr>
<tr>
<td></td>
<td>2. Customer account balance is reduced by the amount of the withdrawal and transaction fees]</td>
</tr>
<tr>
<td>Normal flow</td>
<td>[Provide a detailed description of the user actions and system responses that will take place during execution of the use case under normal, expected conditions. This dialog sequence will ultimately lead to accomplishing the goal stated in the use case name and description</td>
</tr>
<tr>
<td></td>
<td>1. Customer inserts ATM card</td>
</tr>
<tr>
<td></td>
<td>2. Customer enters PIN</td>
</tr>
<tr>
<td></td>
<td>3. System prompts customer to enter language performance English or Spanish</td>
</tr>
<tr>
<td></td>
<td>4. System validates if customer is in the bank network</td>
</tr>
<tr>
<td></td>
<td>5. System prompts user to select transaction type</td>
</tr>
<tr>
<td></td>
<td>6. Customer selects Withdrawal From Checking</td>
</tr>
<tr>
<td></td>
<td>7. System prompts user to enter withdrawal amount</td>
</tr>
<tr>
<td></td>
<td>8. ...]</td>
</tr>
<tr>
<td>Requirements engineering</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Use case ID</th>
<th>Enter a unique numeric identifier for the Use Case. e.g. UC-1.2.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use case name</td>
<td>Enter a short name for the Use Case using an active verb phrase. e.g. Withdraw Cash</td>
</tr>
</tbody>
</table>

9. System ejects ATM card

Alternative flows

[Document legitimate branches from the main flow to handle special conditions (also known as extensions). For each alternative flow reference the branching step number of the normal flow and the condition which must be true in order for this extension to be executed. e.g. Alternative flows in the Withdraw Cash transaction:

4a. In step 4 of the normal flow, if the customer is not in the bank network
   1. System will prompt customer to accept network fee
   2. Customer accepts
   3. Use Case resumes on step 5

4b. In step 4 of the normal flow, if the customer is not in the bank network
   1. System will prompt customer to accept network fee
   2. Customer declines
   3. Transaction is terminated
   4. Use Case resumes on step 9 of normal flow

Note: Insert a new row for each distinctive alternative flow. ]

Exceptions

[Describe any anticipated error conditions that could occur during execution of the use case, and define how the system is to respond to those conditions. e.g. Exceptions to the Withdraw Case transaction

2a. In step 2 of the normal flow, if the customer enters and invalid PIN
   1. Transaction is disapproved
   2. Message to customer to re-enter PIN
   3. Customer enters correct PIN
   Use Case resumes on step 3 of normal flow]

Includes

[List any other use cases that are included ("called") by this use case. Common functionality that appears in multiple use cases can be split out into a separate use case that is included by the ones that need that common functionality, e.g. steps 1-4 in the normal flow would be required for all types of ATM transactions- a Use Case could be written for these steps and "included" in all ATM Use Cases.]

Frequency of use

[How often will this Use Case be executed. This information is primarily useful for designers. e.g. enter values such as 50 per hour, 200 per day, once a week, once a year, on demand etc.]

Special requirements

[Identify any additional requirements, such as nonfunctional requirements, for the use case that may need to be addressed during design or implementation. These may include performance requirements or other quality attributes.]

Assumptions

[List any assumptions that were made in the analysis that led to accepting this use case into the product description and writing the use case description. e.g. For the Withdraw Cash Use Case, an assumption could be: The Bank]
<table>
<thead>
<tr>
<th>Use case ID</th>
<th>Enter a unique numeric identifier for the Use Case. e.g. UC-1.2.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use case name</td>
<td>Enter a short name for the Use Case using an active verb phrase. e.g. Withdraw Cash</td>
</tr>
<tr>
<td></td>
<td>Customer understands either English or Spanish language.</td>
</tr>
<tr>
<td>Notes and issues</td>
<td>List any additional comments about this use case or any remaining open issues or TBDs (To Be Determined) that must be resolved. e.g.</td>
</tr>
<tr>
<td></td>
<td>1. What is the maximum size of the PIN that a user can have?</td>
</tr>
</tbody>
</table>

Use cases are separated into three packages based on the primary actors. Packagis are used to group related elements together. The contents of each package will be detailed later.

**Figure 2.13. Overview of use case packages.**
A "big picture" of all use cases are shown in Figure 2.14, "All identified use cases on a single diagram." However, due to the relatively high number of scenarios this is a bit hard to capture therefore it is very common to have them separated on several smaller diagrams that show only a few use cases at a time.

Figure 2.14. All identified use cases on a single diagram.
The detailed description of the use cases with their tabular representation are given in the following 3 subsections that are followed by a short description of the non-functional requirements of the system.
4.1. Customer use cases

Figure 2.15. Overview of customer use cases.

4.1.1. Browse/search

Figure 2.16. Use case UC001
Use case ID | UC001  
---|---  
Use case name | Browse/Search  
Actors | Customer  
Description | Browse and/or search among products (books, DVDs).  
Trigger | The customer wants to browse among products or the customer would like to search for certain products.  
Precondition | The customer opens a browser on his/her machine.  
Postcondition | The system presents a result list of products (which meets with the given search conditions in case of searching) to the customer.  
Normal flow | 1. The customer opens a browser on his/her machine.  
2. The customer types a keyword in the search field and presses the "OK" button.  
3. The system finds products which meets with the search conditions and presents the result to the customer.  
Alternative flows | Perform Advanced Search  
Steps 2 is replaced with:  
1. The customer performs an advanced search operation. (S)He fills the input fields belonging to different attributes of products, then presses the "OK" button.  
Refine Search Result  
The following steps are added:  
1. The customer refines the result by typing new keywords into the search fields.  
2. The system executes a new search operation on the previous result list and presents a new result list to the customer.
### Use case ID UC001

**Use case name**: Browse/Search

View Product Details

Steps 2-4 are replaced with the following:

1. The customer selects a product from the presented list.
2. The customer presses the "Details" button.
3. A new popup window opens containing some more information about the selected product.

**Exceptions**

The format or number of the search keywords is incorrect.
The system cannot load products (the database is not available).

**Includes**

None

**Notes and issues**

If none of the products meets the given search conditions, the system returns an empty list.

### 4.1.2. Manage bookshelf

**Figure 2.17. Use case UC002**

<table>
<thead>
<tr>
<th>Use case ID</th>
<th>UC002</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Use case name</strong></td>
<td>Manage bookshelf</td>
</tr>
<tr>
<td><strong>Actors</strong></td>
<td>Customer</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>Handling content of a customer’s bookshelf.</td>
</tr>
<tr>
<td><strong>Trigger</strong></td>
<td>The customer wants to save some products for a future order.</td>
</tr>
</tbody>
</table>
| **Precondition** | 1. The customer executes a browsing or a searching operation.  
2. The customer must be logged in. |
| **Postcondition** | The system saves products chosen by the customer to his/her bookshelf. |
| **Normal flow** | 1. The customer selects one or more products from the list.  
2. The customer moves the selected product(s) to the bookshelf. |
### Use case UC002

**Use case name**: Manage bookshelf

**Alternative flows**
1. The customer creates item group(s) or, with another expression, own categories inside the bookshelf which categories can be also hierarchical.
2. The customer selects one or more products from the list.
3. Product(s) will be added to or removed from the own categories by the customer. Each item on the shelf belongs to a certain category which items can be managed together this way.

**Exceptions**
The system is not able to save the items due to a database failure.

**Includes**
1. Browse/Search
2. Manage Item Groups (Own Categories)

**Notes and issues**
In case of deleting a category, all included items will also be deleted.

### 4.1.3. Move items to cart

**Use case UC003**

**Actors**
Customer

**Description**
Move certain items from the bookshelf to the customer’s shopping cart.

**Trigger**
The customer decides to move selected items to the shopping cart.

**Precondition**
1. The customer’s bookshelf contains at least one item.
2. The customer must be logged in.

**Postcondition**
1. Selected items are removed from the bookshelf.
Use case ID | UC003
---|---
Use case name | Move items to cart

Normal flow
1. The customer selects one or more items from the bookshelf.
2. The customer moves the selected item(s) to the shopping cart.

Alternative flows | None

Exceptions | The system cannot delete the selected items from the bookshelf due to a database failure.

Includes | 1. Browse/Search
2. Manage Item Groups (Own Categories)

Notes and issues | In case of moving a complete category to the shopping cart, the bookshelf will not contain this category from now on. The items moved into the shopping cart will lose their category-marker attribute.

4.1.4. Manage Shopping Cart

Figure 2.19. Use case UC004

Use case ID | UC004
---|---
Use case name | Manage Shopping Cart
Actors | Customer
Description | Preparation to an ordering process with adding items to or removing items from the shopping cart.
Trigger | The customer wants to perform an order and therefore chooses items for it.
Precondition | 1. The customer executes a browsing or a searching operation.
### Use case ID
UC004

### Use case name
Manage Shopping Cart

2. The customer must be logged in.

### Postcondition
None

#### Normal flow
1. The customer picks up one or more products from the list.
2. The customer clicks the "Show Shopping Cart" button.
3. The customer adds products to or removes products from the shopping cart.

#### Alternative flows
4. The customer changes the amount of the products. The system re-calculates the total price in this case.

### Exceptions
None

### Includes
Browse/Search

### Notes and issues
The contents of the shopping cart are not saved. There is no possibility to permanently store the customer’s shopping cart (but he/she can use the bookshelf in order to permanently store items).

### 4.1.5. Move items to shelf

**Figure 2.20. Use case UC005**

<table>
<thead>
<tr>
<th>Use case ID</th>
<th>UC005</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use case name</td>
<td>Move items to shelf</td>
</tr>
<tr>
<td>Actors</td>
<td>Customer</td>
</tr>
<tr>
<td>Description</td>
<td>Move certain items from the shopping cart to the customer’s bookshelf.</td>
</tr>
<tr>
<td>Trigger</td>
<td>The customer decides to move selected items to the bookshelf.</td>
</tr>
</tbody>
</table>
### Use case UC005
**Use case name** Move items to shelf

**Precondition**
1. The customer’s shopping cart contains at least one item.
2. The customer must be logged in.

**Postcondition**
1. Selected items are removed from the shopping cart.
2. The customer’s bookshelf will contain these items from now on.

**Normal flow**
1. The customer selects one or more items from the shopping cart.
2. The customer moves the selected item(s) to the bookshelf.

**Exceptions**
The system cannot insert the selected items to the bookshelf due to a database failure.

**Includes**
Browse/Search

**Notes and issues**
The items that are moved onto the bookshelf will own a default category-marker attribute from now on.

#### 4.1.6. Place Order

**Figure 2.21. Use case UC006**

**Use case ID** UC006

**Use case name** Place Order

**Actors** Customer

**Description** The customer creates an order to purchase certain products from the shop.

**Trigger** The customer wants to order certain products from the shop.
<table>
<thead>
<tr>
<th>Use case ID</th>
<th>UC006</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use case name</td>
<td>Place Order</td>
</tr>
</tbody>
</table>
| Precondition | 1. The customer must be logged in.  
2. The customer’s shopping cart contains at least one item. |
| Postcondition | The system saves the new order and performs further processing on it. (Obtaining items from the inventory; packing them together; handing the package over to an external delivery system.) |
| Normal flow | 1. The customer selects one or more items from the shopping cart.  
2. The customer clicks the "Buy" button.  
3. The customer fills in the required personal data (name, phone number, email, shipping address, billing address, etc.) on the "Order" dialog.  
4. The customer chooses the payment method.  
5. The customer confirms the order. |
| Alternative flows | None |
| Exceptions | 1. The system cannot save the order due to a database failure.  
2. The customer can cancel the order at any time before confirming it. |
| Includes | Manage Shopping Cart |
| Notes and issues | 1. The customer is unable to move items back into the shopping cart.  
2. If the order is cancelled, its items will also be lost.  
3. The system calculates and re-calculate the total price for the order according to the amount changes. This price includes not only the value of the price attribute of each item but also the discounts for certain products and for special customers. |

**4.1.7. Feed Back**

**Figure 2.22. Use case UC007**
Use case ID | UC007
---|---
Use case name | Feed Back
Actors | Customer
Description | The customer sends a feedback about products or the ordering process.
Trigger | The customer wants to provide feedback about products or processes.
Precondition | 1. The customer must be logged in.
2. The customer must own at least one order for which he/she wants to provide a feedback.
Postcondition | The system saves and forwards feedback for further processing.
Normal flow | 1. The customer types in a message on the "Feedback" dialog.
2. The customer sends the feedback.
Alternative flows | Instead of sending, it can be cancelled.
Exceptions | The system cannot save the feedback due to a database failure.
Includes | Place Order
Notes and issues | The feedback will not be available by the customer, only by the administrator.

4.1.8. Visit forums

Figure 2.23. Use case UC008
Use case ID | UC008  
---|---  
Use case name | Visit forums  
Actors | Customer  
Description | The customer visits forums to share his/her opinion about the products or the system.  
Trigger | The customer wants to get a line on the products and share his/her opinion or experiences.  
Precondition | None  
Postcondition | The system saves the newly created comments.  
Normal flow | 1. The customer visits the forum site.  
2. The customer reads or writes comments about products.  
Alternative flows | 1a. The customer can create a new topic.  
Exceptions | The system cannot save comments due to a database failure.  
Includes | None  
Notes and issues | A new comment will not automatically be visible to other users, only after moderation.  

### 4.2. Administrator use cases

**Figure 2.24. Overview of administrator use cases.**
4.2.1. Manage Products

Figure 2.25. Use case UC009
Use case ID | UC009
---|---
Use case name | Manage Products
Actors | Administrator
Description | Adding new products to the system, modifying attributes of products, and removing products from the system.
Trigger | The use case is activated when there is a need for adding, modifying, or removing some products in the system.
Precondition | 1. The administrator must be logged in.
2. There must be some products present either in external files (in case of addition) or in the system (in case of modification or deletion).
Postcondition | Product changes will be saved in the system.
Normal flow | 1. The administrator opens the "Manage Products" dialog.
2. The administrator clicks the "Import Products from File" button.
3. He/she chooses one or more files on the dialog.
4. The "Open" button is pressed. The system imports products from the selected files and present them in a list to the administrator.
5. The administrator selects one or more products from the list.
6. The administrator clicks the "Save" button. The system saves the selected products in the database.
7. [[adds new products to the system. He fills in a form with attributes of new products.]]
8. The administrator selects one product from the list.
9. The "Modify" button is clicked. A dialog window opens.
10. Administrator edits the details of the selected product.
11. Upon pressing the "Save" button, changes will be saved into the database.
12. Administrator selects one or more products from the list.
Use case ID | UC009  
---|---
Use case name | Manage Products  
13. He/she clicks the "Delete" button. The system will remove the selected products from the database.  
Alternative flows  
Steps 1—4 might be replaced with:  
1. Administrator opens the "Manage Products" dialog.  
2. Administrator clicks the "Record New Product" button. A dialog window opens.  
3. Administrator fills in the attributes of the new product on the form.  
4. He/she presses the "Save" button. The system saves the newly created product in the database.  
Exceptions | The changes cannot be saved permanently due to a database failure.  
Includes | None  
Notes and issues | Adding, deleting and updating operations cannot be undone.  

4.2.2. Update Order Status

Figure 2.26. Use case UC011

Use case ID | UC011  
---|---
Use case name | Update Order Status  
Actors | Administrator  
Description | The administrator updates the status of pending orders.  
Trigger | The status of an order needs manual intervention.  
Precondition | 1. The administrator must be logged in.  
2. There must be orders recorded in the system.  
Postcondition | New order status will be saved in the system.  
Normal flow | 1. Administrator opens the "Update Order Status" dialog.
Use case ID | UC011
---|---
Use case name | Update Order Status

2. He/she selects the order which needs to be updated from the list.
3. The new status to be set is chosen.
4. Administrator saves the changes to make them permanent.

Alternative flows | None
Exceptions | The system cannot save the new states due to a database failure.
Includes | None
Notes and issues | Changes after saving cannot be made undone.

4.2.3. Process Feedback

Figure 2.27. Use case UC012

Use case ID | UC012
---|---
Use case name | Process Feedback
Actors | Administrator
Description | The administrator processes incoming feedbacks from customers.
Trigger | Administrator wants to evaluate the feedbacks given by the customers.
Precondition | Administrator must be logged in.
Postcondition | None
Normal flow | 1. The administrator opens the "Process Feedbacks" dialog.
2. He/she reads the feedbacks.
Alternative flows | None
Exceptions | None
Includes | None
Notes and issues | None

4.2.4. Compose/Send Newsletter
**Figure 2.28. Use case UC013**

![Use case UC013 Diagram](image)

<table>
<thead>
<tr>
<th>Use case ID</th>
<th>UC013</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use case name</td>
<td>Compose/Send Newsletter</td>
</tr>
<tr>
<td>Actors</td>
<td>Administrator</td>
</tr>
<tr>
<td>Description</td>
<td>The administrator composes and sends newsletters to customers signed up for newsletters.</td>
</tr>
<tr>
<td>Trigger</td>
<td>There is a new action offer created by the management.</td>
</tr>
</tbody>
</table>
| Precondition | 1. Administrator must be logged in.  
2. Action offers must be set in the system by the manager. |
| Postcondition | The newsletter is sent to the customers. |
| Normal flow | 1. The administrator obtains descriptions of the newest action offers.  
2. He/she composes a newsletter holding information about action offers mentioned previously.  
3. The newsletter is sent to the customers signed up. |
| Alternative flows | None |
| Exceptions | The mailing system is not available. |
| Includes | Set Action Offers |
| Notes and issues | None |

**4.2.5. Moderate Forums**

**Figure 2.29. Use case UC014**
Use case ID | UC014  
Use case name | Moderate Forums  
Actors | Administrator  
Description | Administrator moderates comments before they become available to other customers.  
Trigger | Administrator wants to moderate comments arrived since the last review.  
Precondition | 1. Administrator must be logged in.  
2. There is a comment to be moderated.  
Postcondition | The moderated comment will be saved and presented to the other customers.  
Normal flow | 1. The administrator opens the "Moderating Comments" dialog.  
2. He/she reads a new comment.  
3. A decision is made by the administrator whether the comment is acceptable.  
   a. If it is acceptable then the message will be updated with this information so it will become readable by anyone else.  
   b. Otherwise, the comment will be rejected.  
Alternative flows | None  
Exceptions | None  
Includes | None  
Notes and issues | None  

### 4.3. Manager use cases

Figure 2.30. Overview of manager use cases.
4.3.1. Set Action Offers

Figure 2.31. Use case UC015

<table>
<thead>
<tr>
<th>Use case ID</th>
<th>UC015</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use case name</td>
<td>Set Action Offers</td>
</tr>
<tr>
<td>Actors</td>
<td>Manager</td>
</tr>
<tr>
<td>Description</td>
<td>The manager sets action offers in the system.</td>
</tr>
<tr>
<td>Trigger</td>
<td>A management decision about a new action is made.</td>
</tr>
<tr>
<td>Precondition</td>
<td>Manager must be logged in</td>
</tr>
<tr>
<td>Postcondition</td>
<td>The newly created action offers and rules are saved.</td>
</tr>
<tr>
<td>Normal flow</td>
<td>1. The manager sets rules on the form for certain products. During the order process, these rules will be applied together with the customer discounts at</td>
</tr>
</tbody>
</table>
### 4.3.2. Set Customer Discounts

**Figure 2.32. Use case UC016**

<table>
<thead>
<tr>
<th>Use case ID</th>
<th>UC016</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use case name</td>
<td>Set Customer Discounts</td>
</tr>
<tr>
<td>Actors</td>
<td>Manager</td>
</tr>
<tr>
<td>Description</td>
<td>The manager sets rules for special customer discounts.</td>
</tr>
<tr>
<td>Trigger</td>
<td>Management decision is made about customer discounts.</td>
</tr>
<tr>
<td>Precondition</td>
<td>Manager must be logged in</td>
</tr>
<tr>
<td>Postcondition</td>
<td>The newly created rules are saved.</td>
</tr>
</tbody>
</table>
| Normal flow | 1. The manager sets certain rules on the form for each customer category about discounts.  
2. The manager saves the rules. |
| Alternative flows | None |
| Exceptions | The rules cannot be saved in the system due to a database failure. |
| Includes | None |
| Frequency of use | Special |
| Special requirements | Assumptions |
| Notes and issues | The manager must be familiar with the syntax and functionality of special rules. Customers belong to different categories based on the amount of purchased products so far. During the ordering process, these rules will be applied together with action offers for products at calculating total price for a given order. |

### 4.3.3. View/Check Reports/Statistics

**Figure 2.33. Use case UC017**
Use case ID | UC017
---|---
Use case name | View/Check Reports/Statistics
Actors | Manager
Description | The manager can check different reports and/or statistics about the sales.
Trigger | Manager wants to some reports or statistics.
Precondition | The manager must be logged in.
Postcondition | Generated reports are presented to the manager.
Normal flow | 1. The manager opens the "Reports and Statistics" dialog.
| 2. The appropriate report type is selected plus values for required parameters (like time window for the report) are provided.
Alternative flows | None
Exceptions | None
Includes | None
Notes and issues | None

In many processes, the detailed description of the use cases are developed in line with some preliminary models of the system. This side by side development means that we often perform some iterations and as we develop deeper and deeper understanding of the problem, more and more accurate system models that capture the problem space are established. Examples of such models that can be developed at this stage are context models, detailed interaction and behavioral models (described by sequence and activity diagrams and statecharts) and domain models that make use of class diagrams in order to capture the main concepts of the particular domain. (More examples will be shown in Chapter 3, System modeling soon.)

Based on the detailed description of use cases, activity and/or sequence diagrams might be created in order to show how the control flows when a use case is executed. However, it is uncommon to provide such interaction diagrams for each use case: some of the use cases can be so simple that it does not need further elaboration. For complex scenarios it might be reasonable to provide an activity or sequence diagram as it helps the reader to understand easier.

**Figure 2.34. Activity diagram for use cases Browse/Search and Purchase Order**
In UML, an activity diagram is used to display the sequence of activities. Activity diagrams show the workflow from a start point to the finish point detailing the many decision paths that exist in the progression of events contained in the activity. They may be used to detail situations where parallel processing may occur in the execution of some activities. Activity diagrams are useful for business modelling where they are used for detailing the processes involved in business activities.

A sample activity diagram that uses lots of the possible features are shown below.

**Figure 2.35. Activity diagram sample with lot of features ([SPARXSYSTEMS.COM])**

The following elements can occur on an activity diagram:

- **Activity**: An activity is the specification of a parameterized sequence of behaviour. An activity is shown as a round-cornered rectangle enclosing all the actions, control flows and other elements that make up the activity. In this example, Process Order is an example of an activity.

- **Activity parameter**: activities can have parameters which are displayed on the border. Requested order (on the left border of the activity) is an example for such a parameter.
• **Action**: Action is a named element which represents a single atomic step within activity, i.e. that is not further decomposed within the activity. Activity represents a behavior that is composed of individual elements that are actions.

• **Control flow** (or activity edge): A control flow shows the flow of control from one action to the next. Its notation is a line with an arrowhead. (Each arrow is an example.)

• **Initial node**: a filled circle, this is where the execution of the activity starts.

• **Final node**: there are two types of final nodes, activity final (denotes the end of all control flows in the activity) and flow final (used for showing the end of a single control flow). The first is shown in the sample diagram with a dot inside the circle; the second is denoted by a cross inside the circle.

• **Decision and merge nodes**: both denoted by a diamond. They can both be named. The control flows coming away from a decision node will have guard conditions which will allow control to flow if the guard condition is met. The following diagram shows use of a decision node and a merge node. In the example, the left diamond is a decision one while the right one depicts a merge node.

• **Fork and join nodes**: denoted by either a horizontal or a vertical bar (the orientation is dependent on whether the control flow is running left to right or top to bottom). They indicate the start and end of concurrent threads of control. In the example diagram, the left side of the two bars is a fork (concurrent execution of actions start here) while the right side is a join (there is a synchronization).

• **Accept event action**: it is an action that waits for a specific event to occur, notated as a concave pentagon. This action handles asynchronous messages, including asynchronous calls. It cannot be used with synchronous calls (except accept call action). Check shopping cart and Proceed to checkout are both examples for it.

• **Send signal action**: notated as convex pentagon, it creates a signal from its inputs.

This activity diagram shows how a customer will perform the most important scenario of the system, namely, purchasing an order. This process has multiple entries: normally, it starts with either browsing or searching for a product, however, it is also possible to check the actual contents of the shopping cart, or, one can directly proceed to checkout phase in case he or she has a non-empty cart. Note the guards on the outgoing edges of decisions. The [new search] guard should include a refined search, as well, since the normal execution flow of searching requested the possibility of refining search criteria.

The described process still stands on a relatively high level of abstraction: it shows how some of the use cases work together to provide system functionalities but the underlying objects that will interact each other are not revealed. To illustrate how various objects will communicate with each other in run-time, we can create sequence diagrams for some activities. Again, there is no need to assign a separate sequence diagram to each activities as some of the activities work in a very simple way: for example, Browse items only visits Products, there is no further processing. On the other hand, some activities capture more difficult logic that worth depicting on a separate diagram as seen on the following figure.

**Figure 2.36. Sequence diagram of the activity Checkout**
Of course, this is the Customer actor, who initiates the checkout process. When the actual contents of the shopping cart needs to be checked out, the customer will click on the Checkout button which then will result in the invocation of the shopping cart domain object which has the responsibility of the price calculation. It will first ask the customer domain object whether the given customer has some customer-based discounts to calculate with. After retrieving this information, a loop will iterate over each product put into the shopping cart and their individual prices along with their quantities accessed from the associated ProductSelection object will be retrieved. Based on that information, the shopping cart will send itself a self-message (invoke its own method) to calculate the price.

Note

This diagram still omits a couple of details. Of course, it is not the Customer actor itself who will send the checkout message to the shopping cart domain object but there should be a GUI object (a class with the boundary stereotype). According to the Model—View—Controller pattern, an intermediary controller object is also needed since the views are required to notify the controllers about GUI events but these objects and the delegations performed by them are not shown in the figure in order to keep the example small.

Important

Technically, there are two customer lifelines on this diagram, an actor and an object. The former is not an object of the system but a human being using the application and cause various scenarios to be executed while the latter is an object of the system which can therefore receive messages. So this is not an error but a preferred way of how you should model situations when an actor initiates some activities that involve sending messages to the "digital representation" of the actor.

UML 2 Sequence diagram

Sequence diagram is the most common kind of interaction diagram, which focuses on the message interchange between a number of lifelines. Sequence diagram describes an interaction by focusing on the sequence of messages that are exchanged, along with their corresponding occurrence specifications on the lifelines. The following nodes and edges are typically drawn in a UML sequence diagram (this is not an exhaustive list):
• **lifeline**. Lifeline is a named element which represents an individual participant in the interaction. While parts and structural features may have multiplicity greater than 1, lifelines represent only one interacting entity. If the referenced connectable element is multivalued (i.e., has a multiplicity > 1), then the lifeline may have an expression (selector) that specifies which particular part is represented by this lifeline. If the selector is omitted, this means that an arbitrary representative of the multivalued connectable element is chosen.

A lifeline is shown using a symbol that consists of a rectangle forming its "head" followed by a vertical line (which may be dashed) that represents the lifetime of the participant.

• **execution specification**. Execution (full name: execution specification, informally called *activation*) is an interaction fragment which represents a period in the participant's lifetime when it is

• executing a unit of behavior or action within the lifeline,
• sending a signal to another participant,
• waiting for a reply message from another participant.

Note that the execution specification includes the cases when behavior is not active, but just waiting for reply. The duration of an execution is represented by two execution occurrences—the start occurrence and the finish occurrence.

Execution specifications can overlap. They are represented by overlapping rectangles on the same lifeline.

• **message**. Message is a named element that defines one specific kind of communication between lifelines of an interaction. The message specifies not only the kind of communication, but also the sender and the receiver. Sender and receiver are normally two occurrence specifications (points at the ends of messages).

Syntax for the message is:

```plaintext
message ::= [ attribute '=' ] signal-or-operation-name [ arguments ] [ ':' return-value ] | '*'
arguments ::= '(' [argument [',' argument]* ')'
argument ::= [ parameter-name '=' ] argument-value | attribute '=' out-parameter-name [ ':' argument-value ] | '*'
```

Arguments of a message could only be:

• attributes of the sending lifeline,
• constants,
• symbolic values (which are wildcard values representing any legal value),
• explicit parameters of the enclosing interaction,
• attributes of the class owning the interaction.

A message is shown as a line from the sender message end to the receiver message end. The line must be such that every line fragment is either horizontal or downwards when traversed from send event to receive event. The send and receive events may both be on the same lifeline. The form of the line or arrowhead reflects properties of the message.

A message could be one of the following:

• **synchronous call**. Synchronous call typically represents operation call—send message and suspend execution while waiting for response. Synchronous call messages are shown with filled arrow head.
• **asynchronous call**. Asynchronous call—send message and proceed immediately without waiting for return value. Asynchronous messages have an open arrow head.
• **asynchronous signal.** Asynchronous signal message sends a signal.

• **create.** Create message is sent to a lifeline to create itself. It is shown as a dashed line with open arrowhead (looks the same as reply message), and pointing to the created lifeline's head.

• **delete.** Delete message (called stop in previous versions of UML) is sent to terminate another lifeline. The lifeline usually ends with a cross in the form of an X at the bottom denoting destruction occurrence. The notation that should be used is a filled arrow head (like in case of synchronous call) with the stereotype «destroy».

• **reply.** Reply message to an operation call is shown as a dashed line with open arrow head (looks similar to creation message).

• **combined fragment.** Combined fragment is an interaction fragment which defines a combination (expression) of interaction fragments. A combined fragment is defined by an interaction operator and corresponding interaction operands. Through the use of combined fragments the user will be able to describe a number of traces in a compact and concise manner.

Combined fragment may have interaction constraints also called guards. These are boolean expressions shown in square brackets covering the lifeline where the first event occurrence will occur, positioned above that event, in the containing interaction or interaction operand.

Some examples of the interaction operator could be:

• **alt - alternatives.** It represents a choice or alternatives of behavior.

Figure 2.37. Call accept() if balance > 0, call reject() otherwise [UML-DIAGRAMS.ORG]

• **opt - option.** Optional execution (either the operand happens or nothing).

Figure 2.38. Post comments if there were no errors [UML-DIAGRAMS.ORG]

• **loop - iteration.** The loop operand will be repeated a number of times.

Figure 2.39. Potentially infinite loop [UML-DIAGRAMS.ORG]

Number of executions can be controlled by putting a number into parenthesis right after the keyword loop.

• **interaction use.** Interaction use is an interaction fragment which allows to use (or call) another interaction. Large and complex sequence diagrams could be simplified with interaction uses. It is also common reusing some interaction between several other interactions.

Figure 2.40. Web customer and Bookshop use (reference) interaction Checkout [UML-DIAGRAMS.ORG]

Communication diagrams provide almost an alternative notation for sequence diagrams. A communication diagram also shows the messages that are passed between objects but the lifelines are depicted as objects (rectangles) and the order of messages are described by numbering the arrows that represent messages.
**Figure 2.41. Communication diagram of the online bookstore's find book and checkout activities**

![Communication diagram of the online bookstore's find book and checkout activities](image)

**UML 2 Communication diagram**

Communication diagram (called collaboration diagram in UML 1.x) is a kind of UML interaction diagram which shows interactions between objects and/or parts using sequenced messages in a free-form arrangement. Communication diagram corresponds (i.e. could be converted to/from or replaced by) to a simple sequence diagram without structuring mechanisms such as interaction uses and combined fragments. It is also assumed that message overtaking (i.e., the order of the receptions are different from the order of sending of a given set of messages) will not take place or is irrelevant.

On communication diagrams, objects are shown with association connectors between them. Messages are added to the associations and show as short arrows pointing in the direction of the message flow. The sequence of messages is shown through a numbering scheme.

**4.4. Non-functional requirements**

Besides functional requirements, it is important to gather and document non-functional requirements. Such requirements might be product, organizational or external requirements that need to be worded in a verifiable manner. Examples of non-functional requirements (with their categories denoted in parenthesis) for the bookstore are shown in Example 2.1, “Non-functional requirements for the case study.”.

**Example 2.1. Non-functional requirements for the case study.**

**NF01:** At peak periods, the online store is expected to receive 50 search requests per second. (Product)

**NF02:** The system should respond within 2 seconds to 95% of the requests but all valid requests needs to be answered in no more than 4 seconds. (Product)

**NF03:** The application has to be multi-platform. Supported operating systems are: Windows XP or higher and Linux. (Organizational)

**NF04:** During the operation of the system, the operative Act of Accounting of the country of operation must be observed. (External)
Non-functional requirements are in many cases more important than functional ones, especially from the architectural design point of view since non-functional requirements can heavily affect architectural design decisions.
Chapter 3. System modeling

System modeling is the process of developing abstract models of a system, with each model presenting a different view or perspective of that system [SOMMERVILLE2010]. System modeling has generally come to mean representing the system using some kind of graphical notation, which is now almost always based on notations in the Unified Modeling Language (UML). However, it is also possible to develop formal (mathematical) models of a system, usually as a detailed system specification.

Models are used during the requirements engineering process to help derive the requirements for a system, during the design process to describe the system to engineers implementing the system and after implementation to document the system’s structure and operation. You may develop models of both the existing system and the system to be developed:

1. Models of the existing system are used during requirements engineering. They help clarify what the existing system does and can be used as a basis for discussing its strengths and weaknesses. These then lead to requirements for the new system.

2. Models of the new system are used during requirements engineering to help explain the proposed requirements to other system stakeholders. Engineers use these models to discuss design proposals and to document the system for implementation. In a model-driven engineering process, it is possible to generate a complete or partial system implementation from the system model.

The most important aspect of a system model is that it leaves out detail. A model is an abstraction of the system being studied rather than an alternative representation of that system. Ideally, a representation of a system should maintain all the information about the entity being represented but unfortunately, the real world (also known as the universe of discourse) is utterly complex so we need to simplify. An abstraction consciously simplifies and picks out the most evident characteristics.

You may develop different models to represent the system from different perspectives [SOMMERVILLE2010]. For example:

1. An external perspective, where you model the context or environment of the system.

2. An interaction perspective where you model the interactions between a system and its environment or between the components of a system.

3. A structural perspective, where you model the organization of a system or the structure of the data that is processed by the system.

4. A behavioral perspective, where you model the dynamic behavior of the system and how it responds to events.

These perspectives have much in common with Krutchken’s 4 + 1 view of system architecture, where he suggests that you should document a system’s architecture and organization from different perspectives.

When developing system models, you can often be flexible in the way that the graphical notation is used. You do not always need to stick rigidly to the details of a notation. The detail and rigor of a model depends on how you intend to use it. There are three ways in which graphical models are commonly used:

1. As a means of facilitating discussion about an existing or proposed system.

2. As a way of documenting an existing system.

3. As a detailed system description that can be used to generate a system implementation.

In the first case, the purpose of the model is to stimulate the discussion amongst the software engineers involved in developing the system. The models may be incomplete (so long as they cover the key points of the discussion) and they may use the modeling notation informally. This is how models are normally used in so-called ‘agile modeling’.
When models are used as documentation, they do not have to be complete as you may only wish to develop models for some parts of a system. However, these models have to be correct—they should use the notation correctly and be an accurate description of the system.

In the third case, where models are used as part of a model-based development process, the system models have to be both complete and correct. The reason for this is that they are used as a basis for generating the source code of the system. Therefore, you have to be very careful not to confuse similar symbols, such as stick and block arrowheads, that have different meanings.

[SOMMERVILLE2010] identifies four important types of system models, namely, context models, interaction models, structural and behavioral models are introduced.

1. Context models

At an early stage in the specification of a system, you should decide on the system boundaries. This involves working with system stakeholders to decide what functionality should be included in the system and what is provided by the system’s environment. You may decide that automated support for some business processes should be implemented but others should be manual processes or supported by different systems. You should look at possible overlaps in functionality with existing systems and decide where new functionality should be implemented. These decisions should be made early in the process to limit the system costs and the time needed for understanding the system requirements and design.

In some cases, the boundary between a system and its environment is relatively clear. For example, where an automated system is replacing an existing manual or computerized system, the environment of the new system is usually the same as the existing system’s environment. In other cases, there is more flexibility, and you decide what constitutes the boundary between the system and its environment during the requirements engineering process.

Once some decisions on the boundaries of the system have been made, part of the analysis activity is the definition of that context and the dependencies that a system has on its environment. Normally, producing a simple architectural model is the first step in this activity.

Our case study does not deal with all aspects of passing an ordered item to its customer. The proposed system deals with neither the details of payment (e.g., for allowing payment with credit card, the issuing bank’s system needs to be accessed) nor the shipment of the purchased product. It is outside the responsibilities of the system to maintain up-to-date information on the stock items. This information is contained in an inventory database that can be updated if new items arrive to stock separately from the online store application. Context models are often depicted as box and line diagrams since such simple diagrams are enough to put the system into context with other systems. The context model for our case study is shown in the following figure.

Figure 3.1. Context model of the online store.

If we take a look at this picture, it is very easy the figure out where the boundaries of this system are. We can tell which functionalities belong to the system and which are those that are external to it. Due to the fact that we have users with quite different characteristics (and, hopefully, number) the system was split into two parts: the
online store itself (this is what customers meet) which contains all the functionalities required by customers and an admin application that captures the administrative duties. The latter does not need to be used so frequently and there will be a lot less users with administrative privileges therefore it seems to be a wise idea to separate them. It is also possible that different technologies will be used for these parts. However, this is not the systems modeling phase where such decisions need to be made but we wanted to give a rationale of this separation. The separated representation was the result of an analysis activity (i.e., we have found that parts with different characteristics exist) that does not tell anything about implementation technologies.

Context models normally show that the system's environment contains other systems but the types of relationships between the systems of the environment and the system that is being specified are not shown. That is why it is common to create some models describing business processes showing what activities humans or automated processes will execute. UML provides activity diagrams in order to describe those high-level business processes.

2. Interaction models

All systems involve interaction of some kind. This can be user interaction, which involves user inputs and outputs, interaction between the system being developed and other systems or interaction between the components of the system. Modeling user interaction is important as it helps to identify user requirements. Modeling system to system interaction highlights the communication problems that may arise.

Modeling component interaction helps us understand if a proposed system structure is likely to deliver the required system performance and dependability.

Sommerville covers two related (complementary) approaches to interaction modeling:

1. Use case modeling, which is mostly used to model interactions between a system and external actors (users or other systems).

2. Sequence diagrams, which are used to model interactions between system components, although external agents may also be included.

Use case models and sequence diagrams present interaction at different levels of detail and so may be used together. The details of the interactions involved in a high-level use case may be documented in a sequence diagram.

During interaction modeling, additional use cases might be developed (besides those that we have created in the requirements engineering phase). The context model can help us in finding what other systems should our application interact with. Therefore we can conclude and understand that our system should include requirements that allow customers to pay or check the actual delivery status (even if delivery is handled outside our system). Of course, we should provide the detailed description of the newly found use cases using a structured natural language description as we saw for our previous use cases. However, this task is left to the reader as an exercise.

Note

Actors may represent roles played by human users, external systems or hardware, or other subjects.

Figure 3.2. Use cases describing interactions to Bank and Shipping system (external systems).
In this picture one can see that external systems are denoted by actors. One of our use cases, Pay, can be extended when customer selects to pay with credit card. Not all payments require a Bank: it depends on what payment methods are allowed. For example, it might be possible to pay with coupons which does not need a bank to participate in the process. However, payments with credit cards will always require interaction to the credit card issuer. Customers can check the actual delivery status of their purchased products which naturally involves the Shipping system as this is where shipment information are maintained. Customers can also indicate their preferred time slot for delivery (when they are at home) which will let the courier know when to go (or when not to go). For unambiguosity reasons we would like to draw the reader’s attention to the fact that having the Set preferred time range use case implies that this functionality is included by the online bookstore system. It is not a functionality of the shipping system (however, it may offer such a function), or, at least we do not know whether there exists such a useful functionality or not. During modeling our application, we make decisions about the functionalities of the system under design.

Let us consider the scenario which describes when customer transactions needs to be exchanged with the Accounting system. The best solution would be to automate this task. We suppose that on the first day of each week, all the transactions that have been executed during the previous week will be processed and sent to the Accounting system. A use case for the administrator to correct potential problems (claims, failed delivery, whatever) is needed since it is not allowed to account a purchase that has not been successfully delivered.

But the question arises: given a use case that is automatically triggered by the system clock or timer, what actor should be used? In such cases, basically we have three alternatives.

1. **System timer as a primary actor.**

   Since a scheduled task is run internally by the system, it does not provide any visible outcome from a black box perspective. However, since this task matches a functionality, it can be considered as being in the system scope.
With this option, the system timer (or scheduler) is an internal feature that is taken outside of the system (from a black box perspective) by representing it as a primary actor based on the following: the system timer triggers this use case on a periodic basis (e.g. daily, weekly, etc.). It may be argued that the system timer does not completely fulfill the primary actor definition since it does not have any goal to achieve from the use case. However, when there is no trigger defined, the first action from the primary actor can be considered as the triggering event.

This supports the choice of modeling the system timer as the primary actor, which is common practice in UML, providing the advantage of specifying unambiguously that all use cases associated with the System Timer primary actor are triggered on a scheduled basis.

A “system timer” is suitable in the use case analysis since it does not define how this functionality will be implemented. At a later design stage, it will be possible to choose a system clock or any other mean to achieve the role held by the system timer actor.

2. A primary actor must have a goal to achieve.

An alternative involves identifying the primary actor that has a goal in executing the use case.

In the customers’ transaction processing example, the Administrator or the Manager is the actor who has a goal in executing the “Process customer transactions” use case, i.e., getting an up-to-date list of the processed transactions for all customer accounts.

What makes this use case diagram different from most use cases is the lack of interaction between the primary actor and the studied system. If applicable, it is also possible to model the use case’s triggering event through a secondary actor, the system timer. The use of a secondary actor easily communicates via the use case diagram through the use case details that the associated use case runs on a periodic basis.

If the system timer is not used, it is recommended to specify as use case trigger, e.g., “the system timer starts this use case”.

3. The use case has no actor.

The last option involves specifying a use case without any actor. This is UML compliant since we specify a use case that has not got any interaction with the external environment.

However, so we do not miss the “scheduled nature” of our use case, a trigger can be defined with a reference to the internal system timer. If, on the contrary, no trigger is set, a use case on its own may be more difficult to understand for some, or even lead to some questions such as “is this diagram complete? I thought you hadn’t finished your analysis”.

As a conclusion, we suggest of applying the first option, i.e., identification of the system timer as the primary actor, as it is shown in Figure 3.3, “Use cases describing interactions to Accounting system (external systems).”.

However, our goal was to discuss and compare the identified options, which should help weighting the impact from choosing either option or any other that may exist. An important element to be taken into account should be the impact on the communication with the stakeholders and the team through the use case specifications, and how well they will be understood. Personal views and each context will, of course, affect the final decision.

**Figure 3.3. Use cases describing interactions to Accounting system (external systems).**
It is also desirable to isolate and illustrate the operations that an external actor requests of a system, because they play a crucial role in understanding system behavior. The UML uses sequence diagrams as a notation that can illustrate interactions between objects and actors, and the operations initiated by them.

A system sequence diagram (SSD) is a figure that shows the events generated by external actors (along with their order) for a particular scenario of a given use case. Systems are treated as black boxes; the main goal of this diagram is to show events that cross the system boundary from actors to systems.

**Tip**

An SSD should be done for the main success scenario (also known as the normal flow) of the use case, and frequent or complex alternative scenarios.

The UML does not define a separate diagram type for "system" sequence diagrams but we use ordinary sequence diagrams in a special way: one of the lifelines are devoted to the system as a whole (black box).

**Figure 3.4. Use cases describing interactions to Accounting system (external systems).**
For each step of the Place Order scenario, the actor Customer sends messages to the system. In fact, these messages are provided via a user interface, and it is also the GUI that represents the system's responses (note the display message: it is the abstraction of showing a GUI element).

Activity diagrams can have swimlanes that are very useful when capturing a detailed interaction of different systems in the environment. The following diagram shows the whole business process of placing an order, making the payment and providing both invoices and delivery. The responsibilities of each stakeholder (system) are separated by the help of swimlanes. Each swimlane contains only those activities that belong to the responsibilities of the stakeholder provided in the heading of the swimlane.

**Figure 3.5. Activity diagram of the use case Place Order and related activities**
Note

UML allows the swimlanes to draw either horizontally (like here) or vertically.

This diagram uses a different style of final node than we had in Figure 2.34, “Activity diagram for use cases Browse/Search and Purchase Order”. That was the activity final node (a circle with a dot inside) while here we used the flow final node (a circle with a cross inside). The difference between the two node types is that the flow final node denotes the end of a single control flow; the activity final node denotes the end of all control flows within the activity.

3. Structural models

Structural models of software display the organization of a system in terms of the components that make up that system and their relationships. Structural models may be static models, which show the structure of the system design or dynamic models, which show the organization of the system when it is executing. These are not the same things—the dynamic organization of a system as a set of interacting threads may be very different from a static model of the system components.

You create structural models of a system when you are discussing and designing the system architecture. Architectural design is a particularly important topic in software engineering and UML component, package, and deployment diagrams may all be used when presenting architectural models. Deeper details on architectural modeling will be covered in the next chapter. Here we focus on the creation of the analysis-level class diagrams only that are useful for better understanding of the problem domain.

Domains represent the different subject matters that we need to understand to build a system. A domain is an autonomous, real, hypothetical or abstract world inhabited by a set of conceptual entities that behave according to characteristic rules and policies. (Definition from [MELLOR2002].) We abstract like things and call them classes. In forming such abstractions, we ignore most of the things as our aim is to leave out details and concentrate only on the important aspects. The remaining things are are grouped according to some perceptions about what is means to be “like”.
Important

A domain model is a representation of real-world conceptual classes, not of software components. It is not a set of diagrams describing software classes, or software objects with responsibilities. A domain model serves as a visual dictionary of abstractions.

UML uses class diagrams at all abstraction levels to describe the static structure of a system. Starting from domain classes and abstract concepts through design patterns till the constructs of programming language(s) used for the implementation, we use classes. However, you should not think that the same name means the same construct: a UML class and a Java class, for example, might be quite similar and very different, as well. This is because UML uses 4 interpretations when talking about classes:

<table>
<thead>
<tr>
<th>Table 3.1. UML interpretations of classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class as a concept</td>
</tr>
<tr>
<td>Class as a data type</td>
</tr>
<tr>
<td>Class as a set of objects</td>
</tr>
<tr>
<td>Class as implementation</td>
</tr>
</tbody>
</table>

Class diagrams appear in different contexts. It is very convenient to classify them based on the stages of the software engineering lifecycle where they are used.

Analysis-level classes are the elements of the problem domain. They help us to understand and document the problem space.

Design-level classes are about how to transform domain structures into technical structures so we work in the solution space instead of the problem space. Design-level classes combine domain motivations with technical aspects. Some of the classes are closer to the domain elements while others are closer to the elements of the solution.

Implementation-level classes show how a solution is given. These classes map directly to the corresponding construct of the implementation language (Java, C# or C++, etc.).

First we need to establish an analysis-level class diagram as this is what is needed to describe the problem domain itself. To achieve this, an abstraction process is needed: we need to find out what elements (constructs) of the real world are important for our universe of discourse, how can the concepts of the problem domain be revealed.

We choose attributes that support the ideas of likeness we have in mind when abstracting the class. Relationships exist between the things of the domain that are abstracted to associations between classes. The result of this abstraction process is an analysis-level class diagram that will serve as a good source for further refinements (to evolve it into design-level and probably implementation-level diagrams).

UML 2 Class diagrams [UML-DIAGRAMS.ORG]

The class diagram shows the basic building blocks of any object-orientated system. Class diagrams depict a static view of the model, or part of the model, describing what attributes and behavior it has rather than detailing the methods for achieving operations. Class diagrams are most useful in illustrating relationships between classes and interfaces. Generalizations, aggregations, and associations are all valuable in reflecting inheritance, composition or usage, and connections respectively.

- A **class** is a classifier which describes a set of objects that share the same
  - **features**
  - **constraints**
• semantics (meaning).

A class is shown as a solid-outline rectangle containing the class name, and optionally with compartments separated by horizontal lines containing features or other members of the classifier.

As class is the most widely used classifier, there is no need to add the "class" keyword in guillemets («») above the class name. Class name should be centered and in bold face, with the first letter of class name capitalized (if the character set supports upper case).

**Figure 3.6. Class Customer—details suppressed**

Features of a class are *attributes* and *operations*.

When class is shown with three compartments, the middle compartment holds a list of attributes and the bottom compartment holds a list of operations. Attributes and operations should be left justified in plain face, with the first letter of the names in lower case.

**Figure 3.7. Class Customer—analysis level details.**

Characteristics represented by feature may be of the classifier’s instances considered individually (not static) or of the classifier itself (static). The same feature cannot be static in one context and non static in another.

With regard to static features, two alternative semantics are recognized. Static feature may have:

• different values for different featuring classifiers, or

• the same value for all featuring classifiers.

In accordance with this semantics, inheritance of values for static features is permitted but not required by UML 2.

Static features are underlined—but only the names. An ellipsis (...) as the final element of a list of features indicates that additional features exist but are not shown in that list.

Attributes of a class are represented by instances of property that are owned by the class. Some of these attributes may represent the navigable ends of binary associations.

Objects of a class must contain values for each attribute that is a member of that class, in accordance with the characteristics of the attribute, for example its type and multiplicity.

Additional compartments may be provided to show other details, such as constraints, or just to divide features.

A class in the UML could be used as a namespace for other classifiers including classes, interfaces, use cases, etc. Nested classifiers are visible only within the namespace of the containing class.

**Abstract class.** Abstract class exists only for other classes to inherit from and to support reuse of the features declared by it. No object may be a direct instance of an abstract class, although an object may be an indirect instance of one through a subclass that is nonabstract.

The name of an abstract class is shown in italics.

**Figure 3.8. Class Product is an abstract class**
An abstract classifier can also be shown using the keyword «abstract» before the name.

**Standard Class Stereotypes.** Standard UML class stereotypes include:

- **«Focus»** is class that defines the core logic or control flow for one or more supporting classes. The supporting classes may be defined either explicitly using auxiliary classes or implicitly by dependency relationships. Focus classes are typically used for specifying the core business logic or control flow of components during design phase.

- **«Auxiliary»** is class that supports another more central or fundamental class, typically by implementing secondary logic or control flow. The class that the auxiliary supports may be defined either explicitly using a focus class or implicitly by a dependency relationship. Auxiliary classes are typically used for specifying the secondary business logic or control flow of components during design phase.

- **«Type»** is class that specifies a domain of objects together with the operations applicable to the objects, without defining the physical implementation of those objects. Type may have attributes and associations. Behavioral specifications for type operations may be expressed using, for example, activity diagrams. An object may have at most one implementation class, however it may conform to multiple different types.

- **«Utility»** is class that has only class scoped static attributes and operations. As such, utility class usually has no instances.

**Note**

Naming convention for stereotypes and applied stereotypes was changed in UML 2.4.1 to have the first letter in upper case. You will still see stereotypes in lower case, e.g. «focus», «type», everywhere as it will take some time to switch to the new notation.

**Figure 3.9. Math is utility class having static attributes and operations (underlined)**

```
Math
([class]
 + E :: double = 2.71828182845904523536...
 + PI :: double = 3.14159265358979323846...
 + random().nextInt().nextDouble()   Random.com
 - negativeZeroDoubleBits_ long = Float.bitToLong...
 - negativeZeroDoubleBits_ long = Double.bitToLong...

 - Math()
 + sinh(double) : double
 + cosh(double) : double
 + tanh(double) : double
 + atanh(double) : double
 + asinh(double) : double
```

**Nonstandard Class Stereotypes.** There are several nonstandard but routinely used class stereotypes available in several UML tools including IBM Rational Software Architect (RSA) and Sparx Enterprise Architect:

- **«Boundary»**
- **«Control»**
- **«Entity»**
These class stereotypes are part of the so-called Robustness Diagrams which are nonstandard diagrams that often act as bridge from use cases to other models. The stereotypes roughly correspond to the parts of the Model-View-Controller (MVC) design pattern, where Boundary represents View, Control is Controller, and Entity corresponds to the Model.

«Boundary» is a stereotyped class or object that represents some system boundary, e.g. a user interface screen, system interface or device interface object. It could be used in the analysis or conceptual phase of development to capture users or external systems interacting with the system under development. It is often used in sequence diagrams which demonstrate user interactions with the system.

Boundary is drawn as a circle connected with a short line to a vertical line to the left. It could be also shown as a class with the «Boundary» stereotype.

«Control» is a stereotyped class or object that is used to model flow of control or some coordination in behavior. One or several control classes could describe use case realization. System controls represent the dynamics of the designed system and usually describe some "business logic".

Control is drawn as a circle with embedded arrow on the top. It could be also shown as a class with the «Control» stereotype.

Note

UML offers the standard «Focus» stereotype applicable to classes which could be used for specifying the core business logic or control flow of components during design.

«Entity» is a stereotyped class or object that represents some information or data, usually but not necessarily persistent.

Entity is drawn as a circle with line attached to the bottom of the circle. It could be also shown as a class with the «Entity» stereotype.

Business entities represent some "things", items, documents, or information handled, used or processed by business workers while they do business. Examples of business entities are Prescription at the doctor's office, Menu at the restaurant, Ticket at the airport.

System entities represent some information or data, usually but not necessarily persistent, which is processed by the system.

Note

UML has standard «Entity» stereotype applicable to components and representing a business concept of a persistent information.

• An interface is a classifier that declares a set of coherent public features and obligations. An interface specifies a contract. Any instance of a classifier that realizes (implements) the interface must fulfill that contract and thus provides services described by contract.

An interface may be shown using a rectangle symbol with the keyword «interface» preceding the name.

Figure 3.10. Interface Entity

The obligations that may be associated with an interface are in the form of various kinds of constraints (such as pre- and postconditions) or protocol specifications, which may impose ordering restrictions on interactions through the interface.
Since interfaces are declarations, they are not instantiable. Instead, an interface specification is implemented by an instance of an instantiable classifier, which means that the instantiable classifier presents a public facade that conforms to the interface specification.

Any given classifier may implement more than one interface. Interface may be implemented by a number of different classifiers.

An association between an interface and any other classifier implies that a conforming association must exist between any implementation of that interface and that other classifier. In particular, an association between interfaces implies that a conforming association must exist between implementations of the interfaces.

Interfaces realized by a classifier are its provided interfaces, and represent the obligations that instances of that classifier have to their clients. They describe the services that the instances of that classifier offer to their clients.

Interface participating in the interface realization dependency is shown as a circle or ball (also known as the "lollipop notation"), labeled with the name of the interface and attached by a solid line to the classifier that realizes this interface.

**Figure 3.11. Interface ProductDetails is realized (implemented) by Product.**

Required interface specifies services that a classifier needs in order to perform its function and fulfill its own obligations to its clients. It is specified by a usage dependency between the classifier and the corresponding interface.

The usage dependency from a classifier to an interface is shown by representing the interface by a half-circle or socket, labeled with the name of the interface, attached by a solid line to the classifier that requires this interface.

**Figure 3.12. Interface Product is used (required) by Order.**

It is often the case in practice that two or more interfaces are mutually coupled through application-specific dependencies. In such situations, each interface represents a specific role in a multi-party “protocol.” These types of protocol role couplings can be captured by associations between interfaces.

An interface in the UML could be used as a namespace for other classifiers including classes, interfaces, use cases, etc. Nested classifiers are visible only within the namespace of the containing interface.

- A data type is a classifier—similar to a class—whose instances are identified only by their value.

A typical use of data types would be to represent value types from business domain, primitive types or structured types of a programming language. For example, date/time, gender, currency, address could be defined as data types. All copies of an instance of a data type and any instances of that data type with the same value are considered to be equal instances.

A data type is shown using rectangle symbol with keyword «dataType».
Figure 3.13. DateTime data type

A data type may contain attributes and operations to support the modeling of structured data types. Instances of a structured data type are considered to be equal if the structure is the same and the values of the corresponding attributes are equal.

Figure 3.14. Structured data type Address

When data type is referenced by, e.g., as the type of a class attribute, it is shown simply as the name of the data type.

Figure 3.15. Attributes of the Product contains an attribute of data type Money.

Primitive Type. A primitive type is a data type which represents atomic data values, i.e. values having no parts or structure. A primitive data type may have precise semantics and operations defined outside of UML, for example, mathematically.

Standard UML primitive types include:

- Boolean,
- Integer,
- UnlimitedNatural,
- String,
- Real.

Instances of primitive types do not have identity. If two instances have the same representation, then they are indistinguishable.

A primitive type has the keyword «primitive» above or before the name of the primitive type.

Figure 3.16. Primitive data type PersonalName.

Enumeration. An enumeration is a data type whose values are enumerated in the model as user-defined enumeration literals.

An enumeration may be shown using the classifier notation (a rectangle) with the keyword «enumeration». The name of the enumeration is placed in the upper compartment. A compartment listing the attributes for the enumeration is placed below the name compartment. A compartment listing the operations for the enumeration is placed below the attribute compartment.

A list of enumeration literals may be placed, one to a line, in the bottom compartment. The attributes and operations compartments may be suppressed, and typically are suppressed if they would be empty.
A property is a structural feature which could represent

- an attribute of a classifier, or
- a member end of association, or
- a part of a structured classifier.

Here, we will only deal with attributes. A property owned by a classifier represents an attribute of the classifier. Context of the attribute is the owning classifier.

As a structural feature, property represents some named part of the structure of a classifier. For example, Product class could have name, description, unit price, etc. as its properties.

When instance of a classifier is created, each non-static property becomes a part of the state of that instance, and is implemented by some mapping of the name of the property to a specific value or values that the state of the instance is composed of. Values of each property have specific type and are allocated in the slots of the instance or associated with that instance.

The general syntax for properties is shown below:

```
property ::= [ visibility ] [ '/' ] property-name [ ':' property-type ] [ '[' multiplicity ']' ] [ '=' default-value ] [ property-modifiers ]
visibility ::= '+' | '~' | '#' | '-'
property-modifiers ::= '{' property-modifier [ ',' property-modifier ] '}'
property-modifier ::= 'id' | 'readOnly' | 'ordered' | ( 'seq' | 'sequence' ) | 'unique' | 'nonunique' | 'union' | 'redefines' property-name | 'subsets' property-name | property-constraint
```

Optional visibility is the visibility of the property. Note, that there is no default visibility. Also, visibility may be suppressed from being displayed on a diagram, even if it has some value in the model (e.g. stored by UML tool). So, if visibility is not shown on a diagram, it was either not specified or suppressed.

Forward slash '/' means that the property is derived. UML 2.x provides some lazy definition of derived property—property is derived when its value or values can be computed from other information.

The property-type is the type of the property represented by the name of classifier (in case of an attribute this is the attribute type).

Property could have multiplicity. The multiplicity bounds constrain the size of the collection of property values. By default the maximum bound is 1. Multiplicity is a definition of cardinality—i.e. number of elements—of some collection of elements by providing an inclusive interval of non-negative integers to specify the allowable number of instances of described element. Multiplicity interval has some lower bound and (possibly infinite) upper bound:

```
multiplicity-range ::= [ lower-bound '...' ] upper-bound
lower-bound ::= natural-value-specification
upper-bound ::= natural-value-specification | '*'
```
Lower and upper bounds could be natural constants or constant expressions evaluated to natural (non-negative) number. Upper bound could be also specified as asterisk ('*') which denotes unlimited number of elements. Upper bound should be greater than or equal to the lower bound. If the multiplicity is associated with an element whose notation is a text string (such as a class attribute), the multiplicity range is placed within square brackets as part of that text string (as it can be seen in next to the Subtitle and Author attributes of class Book). This means that a book instance might or might not have a subtitle (optional attribute) and it has at least one author but it is possible that a book has multiple authors (without an upper bound).

Figure 3.19. Attribute multiplicities

![Diagram of Book class with attributes:ISBNNumber, Title, Subtitle (0..1), Author (0..*)]

Some typical examples of multiplicity:

<table>
<thead>
<tr>
<th>Multiplicity</th>
<th>Cardinality</th>
</tr>
</thead>
<tbody>
<tr>
<td>0..0 (or briefly 0)</td>
<td>Collection must be empty</td>
</tr>
<tr>
<td>0..1</td>
<td>No instances or one instance</td>
</tr>
<tr>
<td>1..1 (or briefly 1)</td>
<td>Exactly one instance</td>
</tr>
<tr>
<td>0..* (or briefly *)</td>
<td>Zero or more instances</td>
</tr>
<tr>
<td>1..*</td>
<td>At least one instance</td>
</tr>
<tr>
<td>4..4 (or briefly 4)</td>
<td>Exactly 4 instances</td>
</tr>
<tr>
<td>m..n</td>
<td>At least m but no more than n instances</td>
</tr>
</tbody>
</table>

The default-value option is an expression for the default value or values of the property.

Property may have optional modifiers:

Table 3.3. Optional modifiers for properties

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>Property is part of the identifier for the class which owns the property. The id attribute of class Patient provides an example application of this modifier.</td>
</tr>
<tr>
<td>readOnly</td>
<td>Property is read only (isReadOnly = true).</td>
</tr>
<tr>
<td>ordered</td>
<td>Property is ordered (isOrdered = true).</td>
</tr>
<tr>
<td>unique</td>
<td>Multi-valued property has no duplicate values (isUnique = true).</td>
</tr>
<tr>
<td>nonunique</td>
<td>Multi-valued property may have duplicate values (isUnique = false).</td>
</tr>
<tr>
<td>sequence (or briefly seq)</td>
<td>Property is an ordered bag (isUnique = false and isOrdered = true).</td>
</tr>
<tr>
<td>union</td>
<td>Property is a derived union of its subsets.</td>
</tr>
<tr>
<td>redefines property-name</td>
<td>Property redefines an inherited property named property-name.</td>
</tr>
<tr>
<td>subsets property-name</td>
<td>Property is a subset of the property named property-name.</td>
</tr>
<tr>
<td>property-constraints</td>
<td>A constraint that applies to the property.</td>
</tr>
</tbody>
</table>

*Association is a relationship between classifiers which is used to show that instances of classifiers could be either linked to each other or combined logically or physically into some aggregation.
UML specification categorizes association as semantic relationship. Association could be used on different types of UML structure diagrams:

- class diagram associations
- use case diagram associations
- deployment diagram artifact associations
- deployment diagram communication path

The various kinds of association relationships are shown in the following figure:

**Figure 3.20. Association relationship overview diagram** [UML-DIAGRAMS.ORG]

An association is usually drawn as a solid line connecting two classifiers or a single classifier to itself. Name of the association can be shown somewhere near the middle of the association line but not too close to any of the ends of the line. Each end of the line could be decorated with the name of the association end.

Association end is a connection between the line depicting an association and the icon depicting the connected classifier. Name of the association end may be placed near the end of the line. The association end name is commonly referred to as role name. The role name is optional and suppressible.

The idea of the role is that the same classifier can play the same or different roles in other associations. For example, Customer could be the purchaser of some Orders but he/she can be a reviewer of them if this customer is a responsible person for checking orders on behalf of a company when disposing bulk orders.

**Figure 3.21. Association between Customer and Order with association ends is placed by and places.**
Navigability. End property of association is navigable from the opposite end(s) of association if instances of the classifier at this end of the link can be accessed efficiently at runtime from instances at the other ends of the link.

UML specification does not dictate how efficient this access should be or any specific mechanism to achieve the efficiency. It is implementation specific.

**Notation:**

- navigable end is indicated by an open arrowhead on the end of an association
- not navigable end is indicated with a small $\times$ on the end of an association
- no adornment on the end of an association means unspecified navigability (however, it is often considered as two-way navigation, just as a shorthand to the notation with arrowheads on both association ends).

**Figure 3.22. Both ends of association have unspecified navigability** [UML-DIAGRAMS.ORG]

![Figure 3.22](image)

**Figure 3.23. A2 has unspecified navigability while B2 is navigable from A2** [UML-DIAGRAMS.ORG]

![Figure 3.23](image)

**Figure 3.24. A3 is not navigable from B3 while B3 has unspecified navigability** [UML-DIAGRAMS.ORG]

![Figure 3.24](image)

**Figure 3.25. A4 is not navigable from B4 while B4 is navigable from A4** [UML-DIAGRAMS.ORG]

![Figure 3.25](image)

**Figure 3.26. A5 is navigable from B5 and B5 is navigable from A5** [UML-DIAGRAMS.ORG]

![Figure 3.26](image)

**Figure 3.27. A6 is not navigable from B6 and B6 is not navigable from A6** [UML-DIAGRAMS.ORG]

![Figure 3.27](image)
Arity. Each association has specific arity as it could relate two or more items.

- **Binary Association**

  Binary association relates two typed instances. It is normally rendered as a solid line connecting two classifiers, or a solid line connecting a single classifier to itself (the two ends are distinct). The line may consist of one or more connected segments.

- **N-ary Association**

  Any association may be drawn as a diamond (larger than a terminator on a line) with a solid line for each association end connecting the diamond to the classifier that is the end’s type. N-ary association with more than two ends can only be drawn this way.

**Shared and Composite Aggregation.** Aggregation is a binary association representing some whole/part relationship. Aggregation type could be either:

- shared aggregation (also known as aggregation), or
- composite aggregation (also known as composition).

**Aggregation.** Aggregation (shared aggregation) is a "weak" form of aggregation when part instance is independent of the composite:

- the same (shared) part could be included in several composites, and
- if composite is deleted, shared parts may still exist.

  Shared aggregation is shown as binary association decorated with a hollow diamond as a terminal adornment at the aggregate end of the association line. The diamond should be noticeably smaller than the diamond notation for N-ary associations.

**Figure 3.28. Search Service has a Query Builder using shared aggregation**

**Composition.** Composition (composite aggregation) is a "strong" form of aggregation. Composition requirements/features listed in UML specification are:

- it is a whole/part relationship,
- it is binary association,
- part could be included in at most one composite (whole) at a time, and
- if a composite (whole) is deleted, all of its composite parts are "normally" deleted with it.

  Note, that UML does not define how, when and specific order in which parts of the composite are created. Also, in some cases a part can be removed from a composite before the composite is deleted, and so is not necessarily deleted as part of the composite.

  Composite aggregation is depicted as a binary association decorated with a filled black diamond at the aggregate (whole) end.

**Figure 3.29. Composition example: Order is composed of OrderItems**
Order could contain many items, while each OrderItem has exactly one Order that it belongs to. If Order is deleted, all contained OrderItems are deleted as well. The \{ordered\} constraint tells that OrderItems have a well-defined order in an Order.

**Association Class.** An association may be refined to have its own set of features; that is, features that do not belong to any of the connected classifiers but rather to the association itself. Such an association is called an association class. It is both an association, connecting a set of classifiers and a class, and as such could have features and might be included in other associations.

An association class can be seen as an association that also has class properties, or as a class that also has association properties.

An association class is shown as a class symbol attached to the association path by a dashed line. The association path and the association class symbol represent the same underlying model element, which has a single name. The association name may be placed on the path, in the class symbol, or on both, but they must be the same name.

**Figure 3.30. Association class ProductSelection with details suppressed**

![Association class ProductSelection](image)

**UML 2 Object diagram**

An object diagram can be considered a special case of a class diagram. Object diagrams use a subset of the elements of a class diagram in order to emphasize the relationship between instances of classes at some point in time (runtime). They are useful in understanding class diagrams. They do not show anything architecturally different to class diagrams, but reflect multiplicity and roles.

*Object* is an instance of a class or an interface. Object is not a UML element by itself. Objects are rendered on object diagrams as instance specifications.

**Figure 3.31. Anonymous instance of the Customer class.**

![Anonymous instance of the Customer class](image)

In some cases, class of the instance is unknown or not specified. When instance name is also not provided, the notation for such an anonymous instance of an unnamed classifier is simply underlined colon (:).

If an instance has some value, the value specification is shown either after an equal sign (\(=\)) following the instance name, or without the equal sign below the name.

*Slots* are shown as structural features with the feature name followed by an equal sign (\(=\)) and a value specification. Type (classifier) of the feature could be also shown.

**Figure 3.32. Instance book of the Book class has slots with values specified.**

![Instance book of the Book class has slots with values specified](image)
A link is an instance of an association. It is a tuple with one value for the each end of the association, where each value is an instance of the type of the end. Association has at least two ends, represented by properties (end properties).

Link is rendered using the same notation as for an association. Solid line connects instances rather than classifiers. Name of the link could be shown underlined though it is not required. End names (roles) and navigation arrows can be shown.

The following object diagram shows a particular state of an order.

**Figure 3.33. An object diagram that shows runtime instances of some classes**

3.1. Identification of domain classes

The domain model illustrates conceptual classes or vocabulary in the domain. Informally speaking, a conceptual class is an idea, thing, or object. More formally, a conceptual class may be considered in terms of its symbol, intension, and extension.

- **Symbol**: words or images representing a conceptual class.
- **Intension**: the definition of a conceptual class.
- **Extension**: the set of examples to which the conceptual class applies.

Consider the conceptual class for a product as an example. Its symbol is Product, its intension is that is represents something that can be sold or bought which has a name and a unit price. The Product's extension is the set of all products.
Important

It is better to overspecify a domain model with lots of fine-grained conceptual classes than to underspecify it. Do not think that a domain model is better if it has fewer conceptual classes; quite the opposite tends to be true.

It is common to miss conceptual classes during the initial identification step, and to discover them later during the consideration of attributes or associations, or during design work. That's why this work should be performed in iterations. When a new conceptual class is found, the domain model should be augmented. Relational database design principles are probably not the best to follow since when modeling the domain, even attributeless classes might be important to include as they describe a purely behavioral role in the domain instead of playing an informational role.

A common technique used for class identification is to look for the noun phrases in the overall project description. In our case study, we had nouns like customer, order, payment, category, book, DVD, product, name, address, etc. These nouns will be potential classes and attributes.

In order to decide whether a given noun describes a class or an attribute we can apply the following guideline: if it has an identity in the domain then it is a class, otherwise it is an attribute. In this latter case, of course, we need to find its class, as well. That is the reason why we execute this process in iterations. First, we only concentrate on classes, then in a new iteration we assign attributes to classes. The third phase should be finding additional attributes that did not come up in the textual description.

By following these guidelines, we can find a set of initial classes that belong to the domain.

**Figure 3.34. Initial domain classes**

Name, address and category are such nouns that does not describe a real world entity but instead they provide some descriptive information about real world object.

**Note**

Of course, the actual domain highly influences such decisions. Would this system be used for the registry of real estates, address of a property might become a first-class citizen of this system and should have been modeled as a class instead of an attribute.

Name can be associated with users, customers, manages, administrators but products might also have a name. Category is something that allows grouping of various products. Address is belonging to humans (a home address, for example), however, an order can also have associated addresses in multiple roles: a billing address and a shipping address are both addresses. From the domain's perspective these belong to the order not the customer since without an order it would be pointless to talk about a customer's billing or shipping address. That's why it is not considered as an attribute of the customer but something that describes an order.

After that, we need to identify additional attributes of our domain classes. Even if they were not mentioned in the description, customer's phone number (that can be used to contact the customer), the order's date or the actual price of a product are examples of such descriptive information that can easily be identified when investigating the domain deeper. Even we might think that orders have an attribute called customer which represents the person from whom this order has been created.

Later, we need to find the relationships among our domain classes. As a first step of this identification, we should look for the possibility of generalization. Are there any classes that describe more general (or more specific) concepts than others? We should take care of the substitutability. We should introduce generalization/specialization relationships only when the instances of the subclass always belong to the
superclass, as well. From our domain, we can state that a book is a product, and the same holds also for DVDs. An administrator and a customer are users and a manager is special case of an administrator (based on our system's glossary).

Besides generalization/specialization, containment is also an important relationship type. We need to figure out whether we have concepts in our domain that are members (parts) of other concept. In our domain, we can say that a shopping cart contains products, or, contrary, a product is a part of a shopping cart. We should decide how strong this containment relationship is, to find out if it is an aggregation or an even stronger composition. The question to ask is: if a shopping cart is deleted, will products (the cart's members) be also deleted? Of course, products remain products, regardless if shopping carts exist or not, therefore it is an (shared) aggregation.

**Figure 3.35. Product is part of ShoppingCart**

We now look for whether there are attributes that have types of some other classes. As we stated, orders have an attribute Customer but we also have such a class that describes a customer. Therefore, this attribute will be transformed to an association between those classes in our domain model.

If we consider that product categories might be hierarchical then we can conclude that maintaining category as a product attribute is not flexible enough. That is why we create an independent class for product categories: each category might be in a hierarchical relationship with other categories. Of course, this will also result in an association between classes `Product` and `ProductCategory`.

A shopping cart might contain the same product multiple times (if the customer wants to purchase multiple instances of the product). In order to model this issue, the simple containment (as identified above) might not be appropriate. Therefore we can convert it into an ordinary association between `ShoppingCart` and `Product`. Still it misses the quantity which is a descriptive information about the association itself and not about any of the participating classes. Therefore we can introduce an association class `ProductSelection` that can capture this information. By analysing the domain, we can also find that (especially when some customers have discounts) the price one buys a product actually might differ from the list price of the product so it is reasonable to include the actual price into `ProductSelection`, as well. Important to note that if the domain allows discounts then this is still a reflection of a domain concept not a design decision.

Finally, we need to decide on the multiplicities of association, as well. For many cases, it is very straightforward to determine by examining the domain. However, we would like to draw your attention to the fact that multiplicities are very important and they influence subsequent design activities. Let us take a look at the association between `ShoppingCart` and `Order`! The multiplicities we selected (and depicted on the following diagram) state that for each shopping cart there might or might not exist an associated order. This is very natural as it is possible to populate a shopping cart and later abandon the whole process (so decide to not to buy the contents of the cart). On the other hand, there is exactly one (1) shopping cart that to each order. What does it mean? An order instance can only be created when there is an associated shopping cart. It is also easy to understand that this model does not allow placing an order in another way than using a shopping cart as an order object cannot exist without an associated cart. If the domain would allow orders to created by other means than using a shopping cart then 0..1 will be a more convenient decision.

**Important**

When modeling an application, you will always have lots of options. Many of those might result in a good design (even if it is very hard to tell what good is). However, you should always be aware of the consequences of your decisions. They will form constraints on subsequent design activities.

Let us consider the association between `Product` and `ShoppingCart`! The same product can be selected in multiple (or none) carts, this is very easy to understand. It is hard to think of requirements that would think it differently. From the other direction, however, this model shows that a shopping cart instance contains at least one product. This has a consequence that a shopping cart can only be created when there is a product to be placed in it so an empty cart is not allowed. We should only make such decisions if they are dictated by the domain (or at least if we have good reasons for that). Otherwise it will not allow the architect who performs the detailed design to pick a solution that deals with empty carts.
That said, we can draw our initial domain model.

**Figure 3.36. Initial domain model**

In analysis-level class diagrams classes and attributes reflect domain concepts. Attribute names do not need to follow the well-known conventions of some programming languages. They can be capitalized and they can even contain spaces (this also emphasises that we concentrate on the domain, independently of any computer system).

Attribute details are often omitted. Some of them does not even have a type assigned yet but even those attributes whose types have been identified reflect concepts in the domain: PersonalName, Count, Money and Address are examples for it. Not all attributes have domain-specific types: even in this early phase we can decide (based on the domain) whether some attributes are textual (of type String) or numeric.

### 3.2. Refining the model

Even if enough care is taken when designing our domain models, there might be some concepts that were missing. When they are found (regardless which stage of the development cycle we are in) the model should be revised. In many cases, this is not about forgotten concepts but either the deeper understanding results in the need for some classes, attributes or relationships or we left out some concepts on purpose for subsequent iterations.

The latter has happened with our initial model. There we omitted details on type of attributes, however, UML can use stereotypes primitive and dataType for defining special classes that represent data types. Primitive types are those for which identity is not used in checking equality. PersonalName, TelephoneNumber and Count are examples of primitive types. Of course, one might think that simply using Strings instead of first two or integer instead of Count would be enough since this is how we will implement them finally. However, it would be a bad idea to make such a decision now, at analysis level. Of course, not all of the strings would qualify as a name of a person or as a phone number. They must follow some patterns that are known from the domain. Using String in place of PhoneNumber would mean that we do not want to check the format of phone numbers for sure, just as we do not do any checks on other textual objects (like on product description).
We can also apply analysis patterns. Pattern “P of EAA 488” describes the Money pattern to represent monetary values. This pattern was described by Martin Fowler in page 488 of his excellent book entitled Patterns of Enterprise Application Architecture [FOWLER2002] (a similar pattern, Quantity, has been described earlier in [FOWLER1996]). This pattern advocates the coupling of an amount and a currency type together. For the sake of completeness, the primitive types can be seen as a manifestation of Value object pattern (“P of EAA 486”).

We can also notice that the class Order did not contain any information on the status of the order (which needs to be traceable according to the requirements) or the payment type how the order is paid. Most systems offer a set of possible order states and payment methods that are allowed. Therefore we can create few enumerations that list those acceptable values. These are illustrated in Figure 3.37, “Revised domain model with supporting data types”.

**Figure 3.37. Revised domain model with supporting data types**

![Revised domain model with supporting data types](image)

However, even this revised model does not contain information on the concept of bookshelf which is something like a wishlist where customers can save those products that they are interested in. A bookshelf is associated with a customer and contains products. Do you remember the reasons why we switched to represent the relationship between ShoppingCart and Products from aggregation to association with an association class? A sloppy reader would think that we face the same problems now. However, this is not true. There is no reason to put the same product more than once onto a bookshelf (contrary to the case of shopping cart) so no additional pieces of information are needed so using aggregation seems to be a good choice. Products put onto the bookshelf are ordered: however, it is up to the customer to reorder its elements. But since there is an order that exists, we can use the {ordered} constraint to denote it.

**Figure 3.38. Domain model after introducing Bookshelf**

![Domain model after introducing Bookshelf](image)
4. Behavioral models

Behavioral models are models of the dynamic behavior of the system as it is executing. They show what happens or what is supposed to happen when a system responds to a stimulus from its environment. You can think of these stimuli as being of two types:

1. **Data.** Some data arrives that has to be processed by the system.

2. **Events.** Some event happens that triggers system processing. Events may have associated data but this is not always the case.

Many business systems are data processing systems that are primarily driven by data. They are controlled by the data input to the system with relatively little external event processing. Their processing involves a sequence of actions on that data and the generation of an output. For example, our bookstore system will accept information about orders made by a customer, calculate the costs of these orders, and using another system, it will generate an invoice to be sent to that customer.

### 4.1. Data-driven modeling

Data-driven models show the sequence of actions involved in processing input data and generating the associated output. This is very useful during the analysis stage since they show end-to-end processing in a system which means that they show the entire action sequence of how input data become output data. In other words, it shows the response of the system to particular input.

In UML, activity and sequence diagrams can be used to describe such data flows. Note that these are the same diagram types we used for interaction modeling but now the emphasis is put on the processing itself, not on the objects that will participate in processing (interactions). That is why activity diagrams are better used for that purpose since the lifelines of sequence diagrams depict objects and actors therefore some attention must be paid to responsibility allocation when using sequence diagrams.

The basic processes of the online store (starting from the insertion of a new product through browsing and selecting products to buy till tracking the order’s delivery and provide feedback) is shown on the following activity diagram.
The first action in the system is populating it with some products. This is the administrator’s job. The next step in the flow is a fork indicating that the following activities can be executed parallelly. The manager can set discounts for customers and products in an arbitrary order. The next step joins the two branches of the flow. Based on the information set up, the administrator can compose and send newsletters. The customer receives the newsletter sent and visits the site. He/she performs a browsing or searching action. Then, he/she selects one or more products from the list that can be placed either onto the bookshelf or into the shopping cart. If items got onto the shelf and the customer wants to place an order later, the first step is to move selected items into the shopping cart. In the other case, when items got into the shopping cart and the customer wants to save them for later, the first step is to move the selected items onto the shelf. Then, the customer can continue with browsing/searching or can place an order. If the items got earlier onto the bookshelf or into the shopping cart and the customer does not want to either place an order or save items, he/she can continue with browsing/searching products. Steps described previously can be repeated as many times as needed.

If the customer finishes with browsing/searching/selecting and would like to place an order, items of the shopping cart are used to create the new order. The system calculates the total price for the order. At this point, the customer can cancel the order process. If he/she continues with ordering, some pieces of order data (e.g. shipping and billing address) should be filled and the payment mode must be selected. Then if the customer does not cancel the process, the system validates the entered information and as a result, the order is actually created. Customers can check the status of their latest order (pending order) later. The status can be updated by the administrator. Then, customers can send a feedback that is processed also by the administrator. Thereafter, the manager can generate reports about sales data in order to support decision making.

### 4.2. Event-driven modeling

Event-driven modeling shows how a system responds to external and internal events (stimuli). It is based on the assumption that a system has a finite number of states and that an event (stimulus) may cause a transition from one state to another.

The UML supports event-based modeling using state machine diagrams

<table>
<thead>
<tr>
<th>UML 2 State machine diagrams</th>
</tr>
</thead>
<tbody>
<tr>
<td>A state machine diagram models the behaviour of a single object, specifying the sequence of events that an object goes through during its lifetime in response to events. It contains the following elements:</td>
</tr>
<tr>
<td>• <strong>State.</strong> A state is denoted by a round-cornered rectangle with the name of the state written inside it. There are two special states:</td>
</tr>
<tr>
<td>• <strong>Initial state.</strong> The initial state is denoted by a filled black circle and may be labeled with a name.</td>
</tr>
<tr>
<td>• <strong>Final state.</strong> The final state is denoted by a circle with a dot inside and may also be labeled with a name.</td>
</tr>
<tr>
<td>• <strong>Transitions.</strong> Transitions from one state to the next are denoted by lines with arrowheads. A transition may have a trigger, a guard and an effect, as below. <strong>Trigger</strong> is the cause of the transition, which could be a signal, an event, a change in some condition, or the passage of time. <strong>Guard</strong> is a condition which must be true in order for the trigger to cause the transition. <strong>Effect</strong> is an action which will be invoked directly on the object that owns the state machine as a result of the transition.</td>
</tr>
<tr>
<td>• <strong>State action.</strong> State actions describe effects associated with a state. A state action is an activity label/behavior expression pair. The activity label identifies the circumstances under which the behavior will be invoked. There are three reserved activity labels:</td>
</tr>
<tr>
<td>• entry: the behavior is performed upon entry to the state,</td>
</tr>
<tr>
<td>• do: ongoing behavior, performed as long as the element is in the state,</td>
</tr>
<tr>
<td>• exit: a behavior that is performed upon exit from the state.</td>
</tr>
<tr>
<td>• <strong>Self-transition.</strong> A state can have a transition that returns to itself. This is most useful when an effect is associated with the transition.</td>
</tr>
</tbody>
</table>
• **Entry point.** If we do not enter the machine at the normal initial state, we can have additional entry points.

• **Exit point.** In a similar manner to entry points, it is possible to have named alternative exit points.

Besides these main elements, it is important to note that UML state machine diagrams support the notion of superstates that encapsulate a number of separate states. This superstate can be used as a single state on a higher-level model but is then expanded to show more details.

The following state machine diagram shows the internal states and transitions of a shopping cart.

**Figure 3.39. State machine diagram of the class ShoppingCart**

The state diagram shows the internal states and transitions of a shopping cart. When the shopping cart is initiated, it will be in an Empty state. Whenever products are added, a transition to Collecting state is performed. When the shopping cart is in the Collecting state and a Product deleted stimulus happens then based on the guards (if the number of products in cart is equal to 1 or greater) a transition to Empty state might happen.

One might argue that Empty and Collecting are very similar that it does not worth separating them. This can be a valid point, however, the reason why we separated is that when converting a bookshelf’s contents (like a wishlist) to shopping cart contents will not allow having an empty cart so the inclusion of the entry point resulted in the separation.

**Note**

When the cart is in the Empty state, it cannot receive a Product deleted stimuli (there is no such transition starting from Empty) therefore this model disallows deleting products from an empty cart (of course, it is impossible). This is a second reason why states Empty and Collecting has been separated.

The state “Show summary” contains an effect that is executed upon entering the state: total price should be counted as it should be presented to the customer.

The UML notation lets you indicate the activity that takes place in a state. In a detailed system specification you have to provide more detail about both the stimuli and the system states. These are shown in the following tables with a tabular description of each state and how the stimuli that force state transitions are generated.

**Table 3.4. States of a shopping cart**

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty</td>
<td>The shopping cart does not contain any products.</td>
</tr>
<tr>
<td>Collecting</td>
<td>The cart contains some products.</td>
</tr>
<tr>
<td>Show summary</td>
<td>Overview of the shopping cart contents are shown.</td>
</tr>
<tr>
<td>State</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>------------------------------------</td>
</tr>
<tr>
<td>Checkout</td>
<td>Cart contents are ordered.</td>
</tr>
</tbody>
</table>

**Table 3.5. Stimuli of a shopping cart**

<table>
<thead>
<tr>
<th>Stimulus</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product added</td>
<td>The user pressed the Add product button.</td>
</tr>
<tr>
<td>Product deleted</td>
<td>The user selected a product and pressed the Remove product button.</td>
</tr>
<tr>
<td>Finish</td>
<td>The user pressed the Checkout button.</td>
</tr>
<tr>
<td>Cancel</td>
<td>The user pressed the Continue shopping button.</td>
</tr>
<tr>
<td>Accept</td>
<td>The user pressed the Accept button so the contents of the cart are ordered.</td>
</tr>
<tr>
<td>Abandon order</td>
<td>The user pressed the Abandon order button.</td>
</tr>
</tbody>
</table>

A less detailed state machine diagram showing a possible set of order statuses is shown below:

**Figure 3.40. State machine Order**

An order is in NEW state when being created. It changes state when its items are obtained from the inventory and the assembled package is ready for dispatching. Its state becomes to DISPATCHED when the package is under delivering. When the customer receives the package, the state of the order will be DELIVERED. After processing the customer’s feedback, the state can be CLOSED.

5. **Robustness analysis**

Rosenberg and Stephen [ROSENBERG2007] introduced robustness analysis as a way for filling the gap between analysis (the what) and design (the how). From that point of view robustness analysis is a preliminary design when designers make assumptions on the design and start thinking of the possible technical solutions.

For supporting robustness analysis, they use robustness diagrams. This is a nonstandard diagram type in the manner that it is not described by the UML specification, however, it uses UML concepts. It is a specialized communication diagram that uses stereotyped objects. These stereotypes are defined in UML:

- **boundary**. The interface between the system and the outside world. Boundary objects are typically screens or web pages (i.e., the presentation layer that the actor interacts with).
- **entity**. Entity objects are usually objects from the domain model.
- **control**. Control objects are the “glue” between boundary and entity objects.
A robustness diagram is somewhat of a hybrid between a class diagram and an activity diagram. It visually represents a use case's behavior, showing both participating classes and software behavior. Nevertheless, it does not describe which class is responsible for which parts of the behavior. A robustness diagram is probably easier to read than an activity diagram since objects talks to each other. This flow of action is represented by a line between the two objects that are talking to each other.

It is a specialized communication diagram since not all object can talk to any other object. It is useful to think of boundary objects and entity objects as being nouns, and controllers as being verbs. Then the following rules apply in robustness diagrams:

- Nouns can talk to verbs (and vice versa).
- Nouns can’t talk to other nouns.
- Verbs can talk to other verbs.

These rules help to enforce a noun-verb-noun pattern in the text of use cases. This is useful as sequence diagrams have the very same nature: the objects are the nouns, and the messages that go between them are the verbs. So by following that pattern it will be easier to do the detailed design task later. Robustness analysis provides a sanity check for our use cases.

By revisiting our use cases, the first thing that we can find out that browsing and searching products need to be separated into two use cases since they have different operations. Here we only focus on searching which is the more complex of them.

Figure 3.42. Robustness diagram for Search product

Based on the robustness diagram we can rephrase the use case as follows:

<table>
<thead>
<tr>
<th>Use case ID</th>
<th>UC001A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use case name</td>
<td>Search Products</td>
</tr>
</tbody>
</table>
Use case ID | UC001A  
---|---
Use case name | Search Products  
Actors | Customer  
Description | Search for products based on some criteria.  
Trigger | The customer wants to browse among products or the customer would like to search for certain products.  
Precondition | Customer starts a web browser.  
Postcondition | Search results meeting the criteria are displayed.  
2. Customer clicks Search button.  
3. Search page is displayed by the system.  
4. Customer enters search criteria.  
5. The system validates the criteria provided.  
6. The system looks up the catalog to find the products that meet the criteria.  
7. Search results page is displayed with the products fulfilling the criteria.  
Alternative flows | Refine Search Results  
The following steps are added:  
8. Customer refines search results by providing additional criteria.  
9. Steps 5–7 are re-executed.  
Exceptions | In Step 5, if search criteria is invalid or even missing then Step 3 (display search page) will be executed along with some hints on valid criteria.  
In Step 6, if no products meet the criteria then Step 3 (display search page) will be executed along with providing the error message "Product not found".  
Includes | None  
Notes and issues | None  

Based on that we can also introduce a new class to our domain model: Catalog. The Catalog class is an abstraction of a container of all products. This is not to be confused with the inventory that has been mentioned earlier. The inventory contains information about the actually available pieces of products (those that are stocked) while catalog contains product metadata about each existing product (regardless of that there are any pieces of them on stock). Inventory was concerned as implemented by an external application that provides an interface to retrieve information on products in warehouses, however, catalog should be an integral part of this application since the whole lifecycle of a product is covered by our system's functionalities.

We have also created a robustness diagram for one of the most important use cases of the system, Place order.

**Figure 3.43. Robustness diagram for Place order**
The revised use case is the following.

<table>
<thead>
<tr>
<th>Use case ID</th>
<th>UC006A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use case name</td>
<td>Place Order</td>
</tr>
<tr>
<td>Actors</td>
<td>Customer</td>
</tr>
<tr>
<td>Description</td>
<td>Customer places an order to purchase certain products from the shop.</td>
</tr>
<tr>
<td>Trigger</td>
<td>Customer wants to order certain products from the shop.</td>
</tr>
<tr>
<td>Precondition</td>
<td>Customer is logged in.</td>
</tr>
<tr>
<td>Postcondition</td>
<td>The system saves the new order.</td>
</tr>
</tbody>
</table>

**Normal flow**

1. Customer visits the Shopping cart page.
2. Customer clicks the Buy button.
3. The system displays the Order details form where the required data to complete the order (name, phone number, email, shipping and billing addresses, payment method) needs to be provided.
   3.a. The system will retrieve information of the customer in order to populate a list of default values.
4. Customer fills in data and submits Order details form by clicking the Confirm button.
5. The system records the order.
   5.a. It retrieves shopping cart elements to add them to the order.
6. The system displays the Order created page.

**Alternative flows**

None

**Exceptions**

1. The system cannot save the order due to a database failure.
2. The customer can cancel the order at any time before confirming it.

**Includes**

Manage Shopping Cart
Use case ID  | UC006A
--- | ---
Use case name  | Place Order
Notes and issues  
1. The customer is unable to move items back into the shopping cart.  
2. If the order is cancelled, its items will also be lost.  
3. The system calculates and re-calculates the total price for the order according to the amount changes. This price includes not only the value of the price attribute of each item but also the discounts for certain products and for special customers.

For the rest of the use cases, the task of revisiting the use cases are left to the reader.

The ordering process can now illustrated with some sequence diagrams. Short explanations can be read below. These two diagrams reveal the differences found in the ordering phase if the ordering process is performed based on the selected bookshelf items or the shopping cart is used directly for the order.

**Figure 3.44. Sequence diagram with bookshelf**

On this diagram, eight elements can be found. Each element is placed one after another in a horizontal manner. Each of them owns a lifeline (dashed vertical line) that indicates the existence of an element, i.e. the participation of an element in different interactions. The customer element as an actor calls the graphical user interface (GUI) element to establish a connection with the system. (S)He browse and search the list offered by the interface and, probably, views detailed information of certain products. The GUI sends a message to the product entity element to get appropriate information from it. After receiving information, the customer, with the help of the GUI, acquires the bookshelf through a message then (s)he can put selected products onto it. The bookshelf creates an item object that gains the appropriate product and wraps it. The item element returns to the bookshelf i.e. the bookshelf will hold the given product wrapped in an item object from now on. An item object is existing only during the period of time while a bookshelf is using it. Sequences from viewing product details to moving products as items to shelf, can be repeated many times (when the customer places not only one product on the shelf). Thereafter, the customer launches a request through the bookshelf to move certain (or all) items from it to the shopping cart. The customer can place an order through the shopping cart that calls the order entity. The system calculates total price for the order that, thereafter, can be confirmed by the customer who will be informed about this process through the graphical user interface.
This diagram is very similar to the previous one, it is almost the same except that the bookshelf object cannot be found here. The customer adds product items to the shopping cart directly, not through a bookshelf object. The adding steps can be repeated many times. Then, the customer places an order through the shopping cart. An explanation for remaining steps can be found in the previous section.
Chapter 4. Architectural design

Architectural design is concerned with understanding how a system should be organized and designing the overall structure of that system. In the model of the software development process, as shown in Chapter 2, architectural design is the first stage in the software design process. It is the critical link between design and requirements engineering, as it identifies the main structural components in a system and the relationships between them. The output of the architectural design process is an architectural model that describes how the system is organized as a set of communicating components.

In agile processes, it is generally accepted that an early stage of the development process should be concerned with establishing an overall system architecture. Incremental development of architectures is not usually successful. While refactoring components in response to changes is usually relatively easy, refactoring a system architecture is likely to be expensive.

In practice, there is a significant overlap between the processes of requirements engineering and architectural design. Ideally, a system specification should not include any design information. This is unrealistic except for very small systems.

Architectural decomposition is usually necessary to structure and organize the specification. Therefore, as part of the requirements engineering process, you might propose an abstract system architecture where you associate groups of system functions or features with large-scale components or sub-systems. You can then use this decomposition to discuss the requirements and features of the system with stakeholders.

You can design software architectures at two levels of abstraction, which Sommerville calls architecture in the small and architecture in the large:

1. **Architecture in the small** is concerned with the architecture of individual programs. At this level, we are concerned with the way that an individual program is decomposed into components.

2. **Architecture in the large** is concerned with the architecture of complex enterprise systems that include other systems, programs, and program components. These enterprise systems are distributed over different computers, which may be owned and managed by different companies.

The three main advantages of explicitly designing and documenting a software architecture are:

1. **Stakeholder communication.** The architecture is a high-level presentation of the system that may be used as a focus for discussion by a range of different stakeholders.

2. **System analysis.** Making the system architecture explicit at an early stage in the system development requires some analysis. Architectural design decisions have a profound effect on whether or not the system can meet critical requirements such as performance, reliability, and maintainability.

3. **Large-scale reuse.** A model of a system architecture is a compact, manageable description of how a system is organized and how the components interoperate. The system architecture is often the same for systems with similar requirements and so can support large-scale software reuse.

System architectures are often modeled using simple block diagrams. Each box in the diagram represents a component. Boxes within boxes indicate that the component has been decomposed to sub-components. Arrows mean that data and or control signals are passed from component to component in the direction of the arrows.

Block diagrams present a high-level picture of the system structure, which people from different disciplines, who are involved in the system development process, can readily understand. However, in spite of their widespread use, several professionals dislike informal block diagrams for describing an architecture. They claim that these informal diagrams are poor architectural representations, as they show neither the type of the relationships among system components nor the components’ externally visible properties.

The apparent contradictions between practice and architectural theory arise because there are two ways in which an architectural model of a program is used:

1. As a way of facilitating discussion about the system design. A high-level architectural view of a system is useful for communication with system stakeholders and project planning because it is not cluttered with
detail. Stakeholders can relate to it and understand an abstract view of the system. They can then discuss the system as a whole without being confused by detail. The architectural model identifies the key components that are to be developed so managers can start assigning people to plan the development of these systems.

2. As a way of documenting an architecture that has been designed. The aim here is to produce a complete system model that shows the different components in a system, their interfaces, and their connections. The argument for this is that such a detailed architectural description makes it easier to understand and evolve the system.

Block diagrams are an appropriate way of describing the system architecture during the design process, as they are a good way of supporting communications between the people involved in the process. In many projects, these are often the only architectural documentation that exists. However, if the architecture of a system is to be thoroughly documented then it is better to use a notation with well-defined semantics for architectural description.

1. Architecture of the case study application

During architectural design the system should be decomposed onto a set of communicating subsystems. These subsystems will work together in order to provide overall system functionalities and also to meet the non-functional requirements. As the context model clarified, we have a couple of systems that we supposed to be present: an inventory system provides information about the available products, the shipping system is responsible for handling the delivery of products that our customers purchased.

This means that we need legacy systems that we need to communicate with. We did not emphasize it yet but existing systems will also influence the requirements of our systems. It is because they have some existing interfaces how they can be used so it is not up to our architects’ design decisions. Some of the systems in the ecosystem might be replaced from time to time (new releases, change of supplier, etc.), or sometimes we need to prepare our system to work with several existing systems of the same domain. A good example for that is when delivery is performed by several companies which have systems with diverse interfaces but we want to support our customers with the possibility of tracking the status of the order regardless which provider delivers the package.

When discussing architecture, we should not only focus on the decomposition. Issues of allocation of the decomposed entities to resources are also belong to the field of architectural design. Basically, we will have at least two viewpoints when dealing with architectural design: a logical one, describing how the system is structured, how to assign responsibilities to various parts; and a physical perspective which deals with how this system will be executed, which logical parts need to be deployed on which nodes (for example, computers). Actually, we would have more viewpoints as allocation of developer resources is an important issue but here we will only focus on the logical and physical layout.

1.1. Logical architecture

Designing the logical architecture is possibly the stage of the software development lifecycle (SDLC) which leans on patterns the most. It is uncommon to build a system that is not based on some well-known architectural styles. Famous examples of such patterns include the client-server pattern, the broker pattern, the layered architecture. Since our application is intended to be a web application therefore we can apply a layered architecture combined with MVC where we need a data management layer, a business logic layer and a presentation layer.

Data management layer will contain those parts of our application that need to be persisted. For that need the use of an entity management framework is encouraged.

Note

Notice that we still did not make a decision on the implementation platform and technology. An MVC-style layered architecture can be implemented in various platforms, for example, .NET offers the Entity Framework for persistence management and ASPs can be used to implement the web frontend. However, in the Java technology stack we can apply Hibernate or JPA for entity management and servlets, JSPs or JSFs can be used for frontend implementation.
It is not considered to be an error if the architect decides on the implementation technology when making a decision on the architecture, however, it is not always necessary. As far as they can, such decisions might be deferred.

On the other hand, we would like to introduce you into the design of the physical architecture soon. By the time the physical architecture is decided, the implementation platforms and technologies must be fixed, otherwise you would not be able to tell what artifacts are created and where to deploy them. Therefore, we state that Java EE has been selected for implementation and a Java EE-enabled application server like Glassfish has been picked as an execution environment. Persistence is supported by the Java Persistence API (JPA) while servlets, JavaServer Pages (JSPs) and JavaServer Faces (JSFs) will provide the frontend. The middleware will include stateful and stateless EJBs, as well. It is out of the scope of this material to provide detailed information on Java EE development but [GONCALVES2013] and [JAVAEE7TUTORIAL] provides detailed information about this field with lots of examples.

The basic architecture is shown in Figure 4.1, “Overall MVC-based architecture”.

**Figure 4.1. Overall MVC-based architecture**

https://netbeans.org/kb/docs/javaee/ecommerce/design.html

A servlet as a controller will be used to handle incoming requests. The pages from the robustness diagrams (boundary objects) can be mapped to views. Finally, the business data, which will be maintained in a database, can be accessed and modified in the application using EJB session beans with JPA entity classes. These components represent the model.

This is, however, a basic architecture for Java EE applications and does not tell too much about our case. Therefore we need a more fine-grained representation of our system elements.

We need to find which objects will become session beans or entity classes (which are Plain Old Java Objects, i.e., POJOs). For identifying our controller and view classes, we can use the robustness diagrams created earlier since they apply the same concepts. A boundary object is something that users meet so they are the views. User interaction will be handed to control classes (controllers) which might perform data validations and decide what to do. The following modules can be identified:

- **User management**: tracks information about various types of users and their accounts. This module will encapsulate the domain classes User, Customer, Administrator and Manager, along with classes that will be used to manage user accounts (create, delete, lock or unlock accounts).

- **Product management**: provides information about various products, allows searching them, management of product categories (create or delete category, assign categories to products). Domain classes Product, Book, DVD and ProductCategory belong here for sure, even Catalog should be here.

- **Shopping cart management**: allows users to manipulate the contents of shopping cart and bookshelf, place products into/onto them. Shopping cart and Bookshelf are so strongly related to each other that Shopping cart can be populated using a bookshelf. This coupling justifies the decision to put them into the same module.
• Order management: performs order processing when user buys the items is cart. This module will also contain a stub to the payment system (which is outside our scope) and the shipping system to gather information on the actual state of delivery.

• Control: controller classes will be placed here that are responsible for controlling the interaction between users and the system. Creation of subpackages might be needed as there might be diverse set of controllers.

• GUI: placeholder for boundary classes.

These are illustrated on Figure 4.2, “Architecture described using a package diagram”

Figure 4.2. Architecture described using a package diagram

UML 2 Package diagram

Package diagrams are used to reflect the organization of packages and their elements. When used to represent class elements, package diagrams provide a visualization of the namespaces. The most common use for package diagrams is to organize use case diagrams and class diagrams, although the use of package diagrams is not limited to these UML elements.

Package. Package is a namespace used to group together elements that are semantically related and might change together. It is a general purpose mechanism to organize elements into groups to provide better structure for system model. Elements contained in a package share the same namespace. Therefore, the elements contained in a specific namespace must have unique names. Packages can be built to represent either physical or logical relationships. When choosing to include classes in specific packages, it is useful to assign the classes with the same inheritance hierarchy to the same package. There is also a strong argument for including classes that are related via composition, and classes that collaborate with them, in the same package.

Packages can be nested, this is denoted by either the physical containment of packages or there is a special nesting connector that can be used.

Package merge. A «merge» connector between two packages defines an implicit generalization between elements in the source package, and elements with the same name in the target package. The source
element definitions are expanded to include the element definitions contained in the target. The target element definitions are unaffected, as are the definitions of source package elements that don't match names with any element in the target package.

Package merge can be viewed as an operation that takes the contents of two packages and produces a new package that combines the contents of the packages involved in the merge.

Package import. The «import» connector indicates that the elements within the target package, which in this example is a single class, use unqualified names when being referred to from the source package. The source package's namespace gains access to the target classes; the target's namespace is not affected.

This diagram shows the internals of the online bookstore application. Packages internal to the system have been stereotyped with «subsystem». This has a graphical representation on the diagram. Systems that are external to our system have been denoted with the stereotype «system» but we also assign them the non-standard stereotype called «external» in order to denote that these are not parts of the system under design. Dependencies have been analyzed: it is a directed relationship showing that if a target component changes than the dependent component (source) might be affected. Many dependencies here are bi-directional: this is because these packages contains classes that are related to each other using a bi-directional association.

This is something that we can easily reduce by investigating these associations. Do they need to be bi-directional? Let us consider ProductManagement and ShoppingCartManagement first: here Product is associated with both Bookshelf and ShoppingCart. But is it needed for a Product object to access those bookshelves or carts they are put onto? It does not seem too realistic therefore we can change the associations from bi-directional to uni-directional: from the ShoppingCartManagement classes a Product must be navigable. Making this association one-way also results in that one of the directions between the packages has been lost. (The same holds for ShoppingCartManagement and OrderManagement.)

Figure 4.3. Architecture described using a package diagram, revised

Please note that the external systems provide dependencies in our application. Whenever the shipping system (used here for keeping track of order delivery) or the inventory system changes, the dependent packages are subject to change, as well.

UML 2 Component diagram
Component diagrams illustrate the pieces of software, embedded controllers, etc., that will make up a system. A component diagram has a higher level of abstraction than a Class Diagram - usually a component is implemented by one or more classes (or objects) at runtime. They are building blocks so a component can eventually encompass a large portion of a system.

Components in UML could represent logical components (e.g., business components, process components), and physical components (e.g., CORBA components, EJB components, COM+ and .NET components, WSDL components, etc.), along with the artifacts that implement them and the nodes on which they are deployed and executed. It is anticipated that profiles based around components will be developed for specific component technologies and associated hardware and software environments. The following nodes and edges are typically drawn in a component diagram:

- **component.** A component is a structured class representing a modular part of a system with encapsulated content and whose manifestation is replaceable within its environment. A component has its behavior defined in terms of provided interfaces and required interfaces (potentially exposed via ports).

- **interface.** An interface is a classifier that declares a set of coherent public features and obligations. An interface specifies a contract. Any instance of a classifier that realizes (implements) the interface must fulfill that contract and thus provides services described by contract.

- **provided interface.** Interfaces realized by a classifier are its provided interfaces, and represent the obligations that instances of that classifier have to their clients. They describe the services that the instances of that classifier offer to their clients. Interface participating in the interface realization dependency is shown as a circle or ball, labeled with the name of the interface and attached by a solid line to the classifier that realizes this interface. The notation for provided interfaces is the so-called ball notation (also known as lollipop notation).

  **Figure 4.4. Interface ProductDetails is provided by Product.**

- **required interface.** Required interface specifies services that a classifier needs in order to perform its function and fulfill its own obligations to its clients. Required interfaces are denoted by sockets.

  **Figure 4.5. Interface Product is used (required) by Order.**

- **class.** The same as on class diagrams.

- **port.** Using ports with component diagrams allows for a service or behavior to be specified to its environment as well as a service or behavior that a component requires. Ports may specify inputs and outputs as they can operate bi-directionally. Ports are denoted by small squares on the border of the component.

- **connector.** Connector specifies a link that enables communication between two or more instances. Connector was extended in the components to include contracts and specific notation. Connector linking components could be either:

  - **delegation connector.** A delegation connector is a connector that links the external contract of a component (as specified by its ports) to the realization of that behavior. It represents the forwarding...
of events (operation requests and events): a signal that arrives at a port that has a delegation connector to one or more parts or ports on parts will be passed on to those targets for handling.

**Figure 4.6. Delegation connector from the delegating port to the simple port of SearchEngine**

- **assembly connector.** An assembly connector is a connector between two or more parts or ports on parts that defines that one or more parts provide the services that other parts use.

**Figure 4.7. Assembly connector between simple ports of Customer and Shipper and Scheduler components**

- **artifact.** An artifact is a product of the software development process. More details will be given when discussing deployment diagrams.

- **component realization.** Component Realization is specialized realization dependency used to (optionally) define classifiers that realize the contract offered by a component in terms of its provided interfaces and required interfaces.

**Figure 4.8. Component UserService realized by both UserServlet and UserDAO**

- **dependency.** Dependency is a directed relationship which is used to show that some UML element or a set of elements requires, needs or depends on other model elements for specification or implementation. Because of this, dependency is called a supplier—client relationship, where supplier provides something to the client, and thus the client is in some sense incomplete while semantically or structurally dependent on the the supplier element(s). Modification of the supplier may impact the client elements.

- **usage.** A usage is a dependency relationship in which one element (client) requires another element (or a set of elements, supplier) for its full implementation or operation.

**1.2. Physical architecture**
The physical layout of our applications can be depicted by deployment diagrams.

**UML 2 Deployment diagram**

A deployment diagram models the run-time architecture of a system. It shows the configuration of the hardware elements (nodes) and shows how software elements and artifacts are mapped onto those nodes.

**Node.** A Node is either a hardware or software element. It is shown as a three-dimensional box shape. As a deployment target it represents computational resource upon which artifacts may be deployed for execution. A node can contain other elements, such as components or artifacts, or other nodes. Nodes can be specialized into:

- **device.** A device is a node which represents a physical computational resource with processing capability upon which artifacts may be deployed for execution. Examples are: application server, database server, client workstation, mobile device, embedded device, etc.

- **execution environment.** An execution environment is a (software) node that offers an execution environment for specific types of components that are deployed on it in the form of executable artifacts. Components of the appropriate type are deployed to specific execution environments. Examples of execution environments are: OS (operating system), workflow engine, database system, Java EE Container, web server, web browser, etc.

**Node instance.** A node instance can be shown on a diagram. An instance can be distinguished from a node by the fact that its name is underlined and has a colon before its base node type. An instance may or may not have a name before the colon. This is useful when describing a real physical architecture with existing device names (machine names).

**Artifact.** An artifact is a product of the software development process. That may include process models (e.g. use case models, design models etc), source files, executables, design documents, test reports, prototypes, user manuals, etc. An artifact is denoted by a rectangle showing the artifact name, the «artifact» keyword and a document icon. Artifact is a source of a deployment to a node. Examples of artifacts are: text documents, source files, binary executables, archives, etc.

**Communication path.** A communication path is association between two deployment targets, through which they are able to exchange signals and messages. Communication path is notated as association.

**Figure 4.9. Deployment diagram**

The Book & DVD Store system is deployed onto a Glassfish 4 application server. Besides the packaged application (Java EE containers use jars, wars and ears for different purposes) itself, also the JSF library for the graphical user interfaces, and the JDBC driver can be found here. The latter provides a connectivity to the
Oracle DB 11g database server where the database schema for our application is set up. The browser of the client’s machine communicates via HTTP requests and responses with the application server.

This kind of diagram is very useful for system administrators as well because it clarifies which packages should be installed on which nodes. This holds not only for the packages we have developed but third-party libraries, reusable components can (and should) also be shown.
Chapter 5. Detailed design

An object-oriented system is made up of interacting objects that maintain their own local state and provide operations on that state. The representation of the state is private and cannot be accessed directly from outside the object. Object-oriented design processes involve designing object classes and the relationships between these classes. These classes define the objects in the system and their interactions. When the design is realized as an executing program, the objects are created dynamically from these class definitions.

Object-oriented systems are easier to change than systems developed using functional approaches. Objects include both data and operations to manipulate that data. They may therefore be understood and modified as stand-alone entities. Changing the implementation of an object or adding services should not affect other system objects. Because objects are associated with things, there is often a clear mapping between real-world entities (such as hardware components) and their controlling objects in the system. This improves the understandability, and hence the maintainability, of the design.

To develop a system design from concept to detailed, object-oriented design, there are several things that you need to do:

1. Understand and define the context and the external interactions with the system.

2. Design the system architecture.

3. Identify the principal objects in the system.

4. Develop design models.

5. Specify interfaces.

Like all creative activities, design is not a clear-cut, sequential process. You develop a design by getting ideas, proposing solutions, and refining these solutions as information becomes available. You inevitably have to backtrack and retry when problems arise. Sometimes you explore options in detail to see if they work; at other times you ignore details until late in the process. Consequently, I have deliberately not illustrated this process as a simple diagram because that would imply design can be thought of as a neat sequence of activities. In fact, all of the above activities are interleaved and so influence each other.

The design of a system is correct if a system built precisely according to the design satisfies the requirements of that system. Clearly, the goal during the design phase is to produce correct designs. However, correctness is not the sole criterion during the design phase, as there can be many correct designs. The goal of the design process is not simply to produce a design for the system. Instead, the goal is to find the best possible design within the limitations imposed by the requirements and the physical and social environment in which the system will operate. To evaluate a design, we have to specify some evaluation criteria. We will focus on modularity of a system, which is decided mostly by design, as the main criterion for evaluation. A system is considered modular if it consists of discrete modules so that each module can be implemented separately, and a change to one module has minimal impact on other modules. Modularity is clearly a desirable property. Modularity helps in system debugging—isolating the system problem to a module is easier if the system is modular; in system repair—changing a part of the system is easy as it affects few other parts; and in system building—a modular system can be easily built by “putting its modules together.” Nevertheless, modularity allows the development of system parts carried out (almost) independently of each other, therefore reducing development time.

A software system cannot be made modular by simply chopping it into a set of modules. For modularity, each module needs to support a well-defined abstraction and have a clear interface through which it can interact with other modules. To produce modular designs, some criteria must be used to select modules so that the modules support well-defined abstractions and are solvable and modifiable separately. Coupling and cohesion are two modularization criteria, which are often used together. We also discuss the open-closed principle, which is another criterion for modularity.

1. Criteria for modularization

1.1. Coupling
Two modules are considered independent if one can function completely without the presence of the other. Obviously, if two modules are independent, they are solvable and modifiable separately. However, all the modules in a system cannot be independent of each other, as they must interact so that together they produce the desired external behavior of the system. The more connections between modules, the more dependent they are in the sense that more knowledge about one module is required to understand or solve the other module. Hence, the fewer and simpler the connections between modules, the easier it is to understand one without understanding the other.

**Coupling** between modules is the strength of interconnections between modules or a measure of interdependence among modules. In general, the more we must know about module A in order to understand module B, the more closely connected A is to B. “Highly coupled” modules are joined by strong interconnections, while “loosely coupled” modules have weak interconnections. Independent modules have no interconnections. To solve and modify a module separately, we would like the module to be loosely coupled with other modules. The choice of modules decides the coupling between modules. Because the modules of the software system are created during system design, the coupling between modules is largely decided during system design and cannot be reduced during implementation.

Coupling increases with the complexity and obscurity of the interface between modules. To keep coupling low we would like to minimize the number of interfaces per module and the complexity of each interface.

### 1.2. Cohesion

We have seen that coupling is reduced when the relationships among elements in different modules are minimized. That is, coupling is reduced when elements in different modules have little or no bonds between them. Another way of achieving this effect is to strengthen the bond between elements of the same module by maximizing the relationship between elements of the same module.

**Cohesion** is the concept that tries to capture this intramodule. With cohesion, we are interested in determining how closely the elements of a module are related to each other. Cohesion of a module represents how tightly bound the internal elements of the module are to one another. Cohesion of a module gives the designer an idea about whether the different elements of a module belong together in the same module. Cohesion and coupling are clearly related. Usually, the greater the cohesion of each module in the system, the lower the coupling between modules is. This correlation is not perfect, but it has been observed in practice.

Cohesion in object-oriented systems has three aspects:

- **Method cohesion**
- **Class cohesion**
- **Inheritance cohesion**

**Method cohesion** is the same as cohesion in functional modules. It focuses on why the different code elements of a method are together within the method. The highest form of cohesion is if each method implements a clearly defined function, and all statements in the method contribute to implementing this function.

**Class cohesion** focuses on why different attributes and methods are together in this class. The goal is to have a class that implements a single concept or abstraction with all elements contributing toward supporting this concept. In general, whenever there are multiple concepts encapsulated within a class, the cohesion of the class is not as high as it could be, and a designer should try to change the design to have each class encapsulate a single concept. One symptom of the situation where a class has multiple abstractions is that the set of methods can be partitioned into two (or more) groups, each accessing a distinct subset of the attributes. That is, the set of methods and attributes can be partitioned into separate groups, each encapsulating a different concept. Clearly, in such a situation, by having separate classes encapsulating separate concepts, we can have modules with improved cohesion (this is also strongly related to the violation of the so-called single responsibility principle).

**Inheritance cohesion** focuses on the reason why classes are together in a hierarchy. The two main reasons for inheritance are to model generalizationspecialization relationship, and for code reuse. Cohesion is considered high if the hierarchy supports generalization-specialization of some concept (which is likely to naturally lead to reuse of some code). It is considered lower if the hierarchy is primarily for sharing code with weak conceptual
relationship between superclass and subclasses. In other words, it is desired that in an OO system the class hierarchies should be such that they support clearly identified generalization-specialization relationship.

1.3. Open-Closed principle

This principle of object-oriented designed has been coined by Bertrand Meyer, who stated that “Software entities should be open for extension, but closed for modification.” A module being “open for extension” means that its behavior can be extended to accommodate new demands placed on this module due to changes in requirements and system functionality. The module being “closed for modification” means that the existing source code of the module is not changed when making enhancements.

Then how does one make enhancements to a module without changing the existing source code? This principle restricts the changes to modules to extension only, i.e. it allows addition of code, but disallows changing of existing code. If this can be done, clearly, the value is tremendous. Code changes involve heavy risk and to ensure that a change has not “broken” things that were working often requires a lot of regression testing. This risk can be minimized if no changes are made to existing code. But if changes are not made, how will enhancements be made? This principle says that enhancements should be made by adding new code, rather than altering old code.

There is another side benefit of this. Programmers typically prefer writing new code rather than modifying old code. But the reality is that systems that are being built today are being built on top of existing software. If this principle is satisfied, then we can expand existing systems by mostly adding new code to old systems, and minimizing the need for changing code. This principle can be satisfied in OO designs by properly using inheritance and polymorphism. Inheritance allows creating new classes that will extend the behavior of existing classes without changing the original class. And it is this property that can be used to support this principle.

2. Refinement and transformation of analysis-level design

We should not forget that one of the main advantages of the object-oriented design approach was that we have the same concept of class in each SDLC phase (even if different characteristics of classes are emphasized at different phases). So the domain classes we identified during analysis can and should be revisited from the design perspective. Recall that now we should deal with not the problem but the solution space so we need a design that can tell how to achieve the requested functionalities (after analysing the what part earlier). A design-level model should still be independent of concrete implementation platforms since this will ensure that the design is reusable.

As a reminder, here is our latest figure that shows the domain classes (augmented with the class Catalog that has been discussed but never shown on a diagram):

**Figure 5.1. Domain model**
Refinement of the analysis-level model to design-level model will include the following steps (this is not intended to be an exhaustive list):

- check of classes to have a more precise set of attributes with proper types,
- identification of the most important methods that describe the object’s behavior (along with finding names and types of arguments, and, in an ideal case, providing contracts for those methods),
- introduction, deletion or rearrangement of classes as needed: some classes that are good in describing a domain might not be useful in providing a solution, and, which is more often the case, additional classes are needed,
  - especially when thinking of controller classes or classes of the UI,
- introduction of interfaces when possible and reasonable,
- assign solution classes to the packages identified during architectural design (separation of concerns),
- refactor the model to patterns,
- develop sequence diagrams that show how the identified objects will communicate with each other (using the methods that we identified) when a use case is executed: this is to “prove” that the functionality described by the use case is realized.

### 2.1. Overview of attributes and methods

The UML itself does not provide a naming convention for attribute and method names. However, we often use capitalized names in analysis-level diagrams. In design-level diagrams the names of properties approach those that we will need in the implementation so here we switch to the use of the Java naming conventions. The most of the changes we made during this step is very straightforward. They are the direct translations of analysis-level elements onto design-level elements. At design level you should assign a data type to each of your attributes! Therefore we have selected some appropriate types for those that were omitted earlier.

We have decided to use String instead of PersonalName and TelephoneNumber and ISBNumber, the respective primitive types that are no longer used has been deleted. It is a design-level decision to decide on data types that
are not motivated by the domain, that is why we used them earlier (hence the designer can make assumptions and find pros and cons of the change including the introduction of some kind of validation if needed). Some other types motivated by the domain like Address and Money stayed in the model: on one hand, they occur several places and changing Address to String would disallow the possibility of finding customers from a particular city since if it only stored as a String we can never be sure how to parse it properly in order to retrieve the name of the city. On the other hand, the use of a number type like float or double instead of type Money would mean that a single currency is supported only (that is not necessarily a problem but will limit future possibilities).

Enumerations PaymentMethod and OrderState has been included into class Order as internal enumerations so we can meet the notation of nesting a classifier inside another one.

Few method specifications have also been decided. It is common to have much more methods in the design level diagram than we had during analysis because analysis classes might have only used some high-level business processes as methods. Now we are looking for the solution (how to solve ) therefore appropriate behavior (responsibility) needs to be assigned to each class.

Figure 5.2. First version of our design model

At design level, these decisions must be documented. When following a plan-driven approach these pieces of information usually go to the software design descriptions document (SDD) which (like SRS) has IEEE-recommended structure and contents. The best practice says that each attribute and operation of our classes should be documented. Of course this is about the final design not about such an intermediary step we execute right now. However, due to space limits we will not provide full details on each property at the end of design therefore we show an example documentation for classes Order and Manager.

Table 5.1. Class Order

<table>
<thead>
<tr>
<th>Name of property</th>
<th>Type of property</th>
<th>Default value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>datePlaced</td>
<td>Date</td>
<td>The actual date</td>
<td>The date when the order has been placed.</td>
</tr>
<tr>
<td>billingAddress</td>
<td>Address</td>
<td></td>
<td>The address to which the invoice</td>
</tr>
</tbody>
</table>
The Order class represents an order. An instance of this class will be created when the contents of a non-empty shopping cart are checked out.

<table>
<thead>
<tr>
<th>Name of property</th>
<th>Type of property</th>
<th>Default value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>shippingAddress</td>
<td>Address</td>
<td></td>
<td>The address products need to be delivered to.</td>
</tr>
<tr>
<td>customerName</td>
<td>String</td>
<td></td>
<td>Name of the customer who placed the order.</td>
</tr>
<tr>
<td>totalValue</td>
<td>Money</td>
<td></td>
<td>Derived attribute containing the total price of the order.</td>
</tr>
<tr>
<td>status</td>
<td>OrderState</td>
<td>NEW</td>
<td>The status of the order.</td>
</tr>
<tr>
<td>paymentType</td>
<td>PaymentMethod</td>
<td>CREDIT_CARD</td>
<td>The selected payment method.</td>
</tr>
<tr>
<td>pay()</td>
<td>void</td>
<td></td>
<td>Initiate payment with the default payment method. Payment is implemented</td>
</tr>
<tr>
<td>pay(PaymentMethod)</td>
<td>void</td>
<td></td>
<td>Initiate payment using the given payment method.</td>
</tr>
</tbody>
</table>

From the specification it is clear that managers can set some discounts. Setting at least two kinds of discounts (per-Customer and per-Product) should be possible.

**Note**

In fact, this was underspecified. The best we can do in such cases collect enough information on the expected behavior during requirements analysis. Since our case study serves exemplary goals it was not our goal to provide a full set of well-specified requirements.

Let us consider the expected behavior of discount settings: managers can set discounts to individual products, individual customers and product categories. If a discount is set to a category then all products belonging to this category will be discounted. The amount of discount can be specified either using an absolute value (e.g., the discount is x EURs regardless of the actual price) or by providing a percentage for specifying a relative value (e.g., y % discount is given from the actual price). In cases when multiple discounts are present (e.g., when a discounted customer buys a discounted product), it should not be possible to give more than one discounts. Instead the one which is more friendly to the customer should be applied.

**Table 5.2. Class Manager**
The Manager class represents a user of the system. All new... methods return with an int that is an id for that rule which can be used later for deleting it.

<table>
<thead>
<tr>
<th>Name of property</th>
<th>Type of property</th>
<th>Default value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>newDiscount(Customer, Money)</td>
<td>int</td>
<td></td>
<td>Sets customer-based absolute discount.</td>
</tr>
<tr>
<td>newDiscount(Customer, int)</td>
<td>int</td>
<td></td>
<td>Sets customer-based relative discount (int is the percentage).</td>
</tr>
<tr>
<td>newDiscount(Product, Money)</td>
<td>int</td>
<td></td>
<td>Sets product-based absolute discount.</td>
</tr>
<tr>
<td>newDiscount(Product, int)</td>
<td>int</td>
<td></td>
<td>Sets product-based relative discount (int is the percentage).</td>
</tr>
<tr>
<td>newDiscount(ProductCategory, Money)</td>
<td>int</td>
<td></td>
<td>Sets absolute discount to all products of the given category.</td>
</tr>
<tr>
<td>newDiscount(ProductCategory, int)</td>
<td>int</td>
<td></td>
<td>Sets relative discount to all products of the given category (int is the percentage).</td>
</tr>
<tr>
<td>removeDiscount(int)</td>
<td>void</td>
<td></td>
<td>Discount rule with the provided id is removed.</td>
</tr>
<tr>
<td>removeDiscount(int, int)</td>
<td>void</td>
<td></td>
<td>Discount rules with ids of the given range are removed.</td>
</tr>
<tr>
<td>removeDiscounts()</td>
<td>void</td>
<td></td>
<td>All discounts are removed.</td>
</tr>
</tbody>
</table>

This design also implies some further design. The ids returned by the `newDiscount` methods needs to be stored if we ever want to remove them. On the other hand, this solution for setting discounts is not generic enough. For more sophisticated cases some kind of rule engines should be applied.

**Note**

Of course we identified only a subset of attributes and methods. Readers should continue this work by introducing attributes and methods they found. But be warned! At design level we are not interested in methods like getters/setters. They simply do not add any value to the model. They are important from the implementation point of view since they provide access to attribute values that have been declared as private for achieving encapsulation. Modeling on the design level is done on a higher level of abstraction.

We should also store the discounts set for customers and products therefore we need additional attributes (and `discounts : double of Product` has to completely replaced). We need two attributes for each discount since otherwise we could not find out which is better for the buyer.
Now we can define the contracts of the methods listed above. We only give a single yet informal example for Manager’s newDiscount(Product, int), the rest is left to the reader. The precondition for this newDiscount method is that the first parameter is a valid product while the second parameter is between 0 and 100 (as it depicts a percentage value). If this is fulfilled, then the contract’s postcondition part will be provided to guarantee that the product is looked up and its discountRelative attribute is set to the value given in the second parameter if it is bigger than the actual value of that attribute (so this is a bigger discount than the earlier one if such existed—remember our business rule to provide the best case for customers).

2.2. Shopping cart issues

A relatively big change to the analysis model has also been introduced: we changed the way how shopping cart is handled. For analysis purposes it was OK but from design considerations we had to change. Our analysis model did not use the concept of an item (let it be shopping cart item or order item). This is not a problem: in the real world we still rarely use that concept. We talk about orders, products, shopping carts but we do not talk about order items. It proves that our initial analysis model was good since it used the same concepts as people do so no one was forced to use unfamiliar concepts. It is not a real part of the problem domain.

However, it is an integral part of the solution itself. When an invoice is needed these items should be printed on them, for example. Therefore, instead of the solution used in the analysis model (an association class if you remember) needs to be replaced with a functionally equivalent solution that allows printing invoices if we need. An item is for sure connected to a product but then the question arises: what has an item? A shopping cart? An order? Any of those but not both! If you consider the activities of ordering, the shopping cart should be populated first. An order object will not even be instantiated until the contents of the cart are checked out. But checkout is a final stage of the cart’s lifecycle: the order is instantiated, the items will become order items and the shopping cart will never be needed again. That is why we decided not to differentiate between cart items or order items: they are only items that can hold a given amount of quantity of a product. First these item instances will be attached to the cart and upon checkout they will be provided to the order. UML provides a notation to denote that participation in those two associations (i.e., compositions in this particular case) is mutually exclusive. A proper notation is to use a note that is attached to both associations and the note should contain the constraint {XOR} to tell that these associations are exclusively or-ed together.

The following sequence diagram illustrates this concept: when the customer visits a product’s page, he/she might want to add some items of this product to the cart. Later, when c

Figure 5.4. Sequence diagram of order creation
2.3. What about Catalog?

The class `Catalog` has been introduced in the analysis model to somehow provide access to all products if it was needed. This is acceptable at analysis level (strictly because we do not execute any code then) but really having such an object might become a bit more than a nightmare. The maintenance costs of such an object will increase together with the number of products (it might be even worse) since we should guarantee that no product objects are left out the catalog, none of them are duplicated and the problems of having a really huge number of objects instantiated in the memory seems pointless, especially if we know that only few of them is really needed.

Hopefully it is clear why we need to get rid of that class. But what should we do then? How can perform such operations like search for a particular product if we do not have such a class? We need a persistent storage where huge amounts of unnecessary data can be stored. Use of persistent stores provide further advantages: our domain object can survive the stoppage of the application (when the contents of the memory are lost) let it be wanted or unwanted. We also need persistent stores because the size of memory is limited and it is possible that some objects would not fit if everything stored in the memory. Disks (that we predominantly use as a persistent store are bigger with some orders of magnitude so it is less to be feared that we run out of space there (however, it might happen, of course).

**Important**

Persistent storage of data can be considered as a necessary evil. Ideally we would have enough memory (probably the same amount of memory that Turing-machines have would be enough), we would not fear our machine freeze and so on. Then we would not need persistent storage. Or we still would since today's database management systems provide transactional capabilities and locking mechanisms for concurrent data access.

We just want to underline that dealing with persistent data is not preordained. Even if we often use persistent stores, not all of the applications will make use of them.

So we need a persistent data storage solution. The high majority of domain objects should be persisted. The logic of the persistent data access can change over time. We start persisting our small set of data in XML files but later when the amount of data will dramatically increase, we can look for a database-backed solutions. We can regularly change our databases, as well: upgrading to a new version, finding one of a better performance, switching platforms might all result in that our code which depends on the persistence layer implemmentation needs to be modified.
Therefore high-level data services our applications require (such like give me a list of all products belonging to a given category or create a report on the orders that have been placed in calendar year of 2013, etc.) should be decoupled from the data access logic. Even SQL has several dialects but if you consider that the schema can also change it is easy to understand that we should only concentrate on high-level data requirements first that are not subject to such frequent changes.

A well-known example is the use of the Data Access Object (DAO) pattern.

**Data Access Object**

The DAO implements the access mechanism required to work with data sources. A data source could be a persistent store like an RDBMS, an external service like a B2B exchange, a repository like an LDAP database, or a business service accessed via CORBA Internet Inter-ORB Protocol (IIOP) or low-level sockets. The business component that relies on the DAO uses the simpler interface exposed by the DAO for its clients. The DAO completely hides the data source implementation details from its clients. Because the interface exposed by the DAO to clients does not change when the underlying data source implementation changes, this pattern allows the DAO to adapt to different storage schemes without affecting its clients or business components. Essentially, the DAO acts as an adapter between the component and the data source.

The structure of DAO is the following (all images are taken from the link above):

**Figure 5.5. Data Access Object**

This pattern has the following participants:

- **BusinessObject**: represents the data client. It is the object that requires access to the data source to obtain and store data. A BusinessObject may be implemented as a session bean, entity bean, or some other Java object, in addition to a servlet or helper bean that accesses the data source.

- **DataAccessObject**: it is the primary object of this pattern. The DataAccessObject abstracts the underlying data access implementation for the BusinessObject to enable transparent access to the data source. The BusinessObject also delegates data load and store operations to the DataAccessObject.

- **DataSource**: represents a data source implementation. A data source could be a database such as an RDBMS, OODBMS, XML repository, flat file system, and so forth. A data source can also be another system (legacy/mainframe), service (B2B service or credit card bureau), or some kind of repository (LDAP).

- **TransferObject**: an optional object (which means that it is not necessarily used, DAO can work well even without it) that represents a Transfer Object (TO or DTO for Data TO) used as a data carrier. The DataAccessObject may use a Transfer Object to return data to the client. The DataAccessObject may also receive the data from the client in a Transfer Object to update the data in the data source.

A sequence diagram that shows the interaction among participants follows:
DAO is often combined with Factory Method or Abstract Factor patterns. When the underlying storage is subject to change from one implementation to another, this strategy may be implemented using the Abstract Factory pattern. The Abstract Factory can in turn build on and use the Factory Method implementation, as suggested in Design Patterns: Elements of Reusable Object-Oriented Software [GoF]. In this case, this strategy provides an abstract DAO factory object (Abstract Factory) that can construct various types of concrete DAO factories, each factory supporting a different type of persistent storage implementation. Once you obtain the concrete DAO factory for a specific implementation, you use it to produce DAOs supported and implemented in that implementation.

The class diagram for this strategy is shown below. This class diagram shows a base DAO factory, which is an abstract class that is inherited and implemented by different concrete DAO factories to support storage implementation-specific access. The client can obtain a concrete DAO factory implementation such as RdbDAOFactory and use it to obtain concrete DAOs that work with that specific storage implementation. For example, the data client can obtain an RdbDAOFactory and use it to get specific DAOs such as RdbCustomerDAO, RdbAccountDAO, and so forth. The DAOs can extend and implement a generic base class (shown as DAO1 and DAO2) that specifically describe the DAO requirements for the business object it supports. Each concrete DAO is responsible for connecting to the data source and obtaining and manipulating data for the business object it supports.

**Figure 5.6. Data Access Object sequence diagram**

![Sequence Diagram](image)

**Figure 5.7. DAO with abstract factory**
The sequence diagram describing the interactions for this strategy is shown on the following figure.

Figure 5.8. DAO with abstract factory sequence diagram
Such a DAO with a well-defined interface will be a much better alternative to our Catalog object.

**Figure 5.9. Interface ProductDAO**

You can add as many operations as are needed in your domain. Notice that where multiple values are returned we used the array notation. This is not because it is not acceptable to retrieve a java.util.List (as an example) in Java but the design models are intended to be language agnostic. UML does not support collection directly, however, one of its supporting standards called the Object Constraint Language (OCL) defines Set(T) and Bag(T) (among others) so it would be enough to write Set(Product) but Product[] is equally OK as all programming languages support arrays. Later, when you further refine your design onto implementation level models you can map this array to any collection type of the given language.

In our design model, we will replace class Catalog with interface ProductDAO and during implementation programmers should implement this interface with a technology-specific class. You can also decide to introduce
the factory, too, if the anticipated benefits outweigh increased complexity. You should not introduce a solution into your design just because you can. Each decisions have pros and cons and if you deal with unnecessary things then probably you will be at a pinch due to the consequences of them.

Note

You should create DAOs for encapsulating all of your data access including an OrderDAO that can be used by managers to create various reports based on the order history or a CustomerDAO that offers a method as a part of its API to retrieve the list of customers whose user account is locked.

2.4. Decouple OrderManagement from external dependencies

For similar reasons why we introduced DAOs (for decoupling our business-related system parts from technology-specific or implementation-dependent parts, we can also introduce two additional modules even if they might seem surprising for the first sight. These modules should be:

- Inventory, and
- Shipping

As the requirements clearly stated, these are independen (probably legacy) systems. However, we can decide to capture them as subsystems that will delegate invocations towards the legacy systems. The intent of these decisions was that this way we can decouple our modules (especially Order management) from these external systems since both Inventory and Shipping subsystems will encapsulate the complexity of those external systems. This is the reason why Order management does not need to implement 3 different methods to access the traceability information of orders if there are 3 different delivery service providers. Instead of the need for multiple implementations, the Shipping subsystem will provide a simple interface to Order management and will implement the appropriate mappings to the legacy systems inside. If a new delivery company will start to ship then it is will be enough to add the mapping to its interface but subsystem Order management will not be bothered with the change,This is, however, a special combination of the Façade and the Adapter design patterns as there is an interface transformation which is hidden behind a façade.

2.5. The show must go on

Of course, a design can never be final. It is not just about that the environment and the requirements are always change but we should take into consideration that the design of a software product is unique. It carries and captures the designer's internals like a fingerprint. There are lots of issues that could be designed differently. There are hundreds of design decisions that must be made in order to “finalize” a design, which is a step towards completing the system. However, our goal was not to provide a fully detailed design but to show the major steps, emphasize the most important activities and provide an approach how you should perform your modeling tasks.
Chapter 6. Using UML profiles

In this chapter we discuss the extensibility of UML, based on the information available in [UML-DIAGRAMS.ORG]. It is permitted to extend the language in controlled ways. These mechanisms are used to tailor the UML to the specific needs by creating standardized collections of UML elements that may be reused.

Profile diagram is a structure diagram which describes lightweight extension mechanism to the UML by defining custom stereotypes, tagged values, and constraints. Profiles allow adaptation of the UML metamodel for different:

- platforms (such as Java EE or .NET), or
- domains (such as real-time or business process modeling).

For example, semantics of standard UML metamodel elements could be specialized in a profile. In a model with the profile "Java model," generalization of classes should be able to be restricted to single inheritance without having to explicitly assign a stereotype «Java class» to each and every class instance.

The profiles mechanism is not a first-class extension mechanism. It does not allow to modify existing metamodels or to create a new metamodel as MOF does. Profile only allows adaptation or customization of an existing metamodel with constructs that are specific to a particular domain, platform, or method. It is not possible to take away any of the constraints that apply to a metamodel, but it is possible to add new constraints that are specific to the profile.

Metamodel customizations are defined in a profile, which is then applied to a package. Stereotypes are specific metaclasses, tagged values are standard metaattributes, and profiles are specific kinds of packages.

OMG now (as of April 2014) provides a number of UML Profile Specifications that are useful to tailor the language to specific areas—some for business modeling; others for particular technologies. These specifications include profiles for BPMN processes, CORBA, QoS and Fault Tolerance and Testing. Besides those formally specified profiles, UML Superstructure informally describes additional example profiles for J2EE/Enterprise Java Beans (EJB), COM and .NET components.

Profiles can be dynamically applied to or retracted from a model. They can also be dynamically combined so that several profiles will be applied at the same time on the same model.

Profiles were present in UML 1.x. Profile diagrams have been introduced in UML 2.0 but first appeared on "official" taxonomy of UML diagrams in UML 2.2.

Graphical nodes and edges drawn on profile diagrams are: profile, metaclass, stereotype, extension, reference, profile application.

1. Profile

Profile is a profile package that extends a reference metamodel (such as UML) by allowing to adapt or customize the metamodel with constructs that are specific to a particular domain, platform, or a software development method. In other words, profile is a lightweight extension mechanism to the UML standard.

A profile introduces several constraints, or restrictions, on ordinary metamodeling through the use of the metaclasses defined in this package. The primary extension construct is stereotype, which is defined as part of profile and extends some metaclass.

A profile is a restricted form of a reference metamodel that must always be related to some reference metamodel that is created from MOF such as UML or CWM. It is not possible to define a standalone profile, without its reference metamodel.

Profile uses the same notation as a package, with the addition that the keyword «profile» is shown before or above the name of the package.

Figure 6.1. Profile EJB [UML-DIAGRAMS.ORG]
A profile can define classes, stereotypes, data types, primitive types, enumerations.

**Figure 6.2. Profile Servers [UML-DIAGRAMS.ORG]**

One profile might reuse some or all parts of another profile, to extend already existing profiles. Multiple profiles could be applied to the same model.

The constraints that are part of the profile are evaluated when the profile has been applied to a package. These constraints need to be satisfied in order for the model to be well-formed.

Package since UML 2.4 has optional URI attribute which serves as unique identifier of the package. Profile is a package, and the URI attribute was introduced mostly to support exchange of profiles using XMI.

The URI attribute of a profile may be rendered in the form `{uri=uri}` after the profile name. OMG normative profiles follow OMG normative naming scheme for URIs. For non-standard, custom profiles convention recommended by OMG looks like:

```
uri ::= http://qualified-profile-parent/profile-version/profile-name.xmi
```

- `qualified-profile-parent` is the qualified name of the package containing the profile (if any), with `::` replaced by `"/"` (forward slash), and all other illegal XML QName characters removed. For example: `www.uml-diagrams.org/profiles`.

- `profile-version` is a profile version id. OMG normative profiles use for this purpose date in the format `YYYYMMDD`, for example: `20110331`.

- `profile-name` is the name of the profile, should contain only valid XML QName characters. For example: `ejb-30`.

**Figure 6.3. EJB Profile shown as a package with URI attribute [UML-DIAGRAMS.ORG]**

2. **Metaclass**

*Metaclass* is a profile class and a packageable element which may be extended through one or more stereotypes.

A metaclass may be shown with the optional stereotype «Metaclass» shown above or before its name (all lowercase «metaclass» was used in UML versions prior to 2.4).
3. Stereotype

A stereotype is a profile class which defines how an existing metaclass may be extended as part of a profile. It enables the use of a platform or domain specific terminology or notation in place of, or in addition to, the ones used for the extended metaclass.

A stereotype extends the vocabulary of the UML, allowing to create new kinds of building blocks that are derived from existing ones but that are specific to the problem. For example, during modeling a network, designers deal with concepts like routers and hubs. By creating stereotypes for them, UML is extended by new building blocks that have special properties, semantics and notation.

A stereotype cannot be used by itself, but must always be used with one of the metaclasses it extends. Stereotype cannot be extended by another stereotype.

A stereotype uses the same notation as a class, with the keyword «stereotype» shown before or above the name of the stereotype. Stereotype names should not clash with keyword names for the extended model element.
Because stereotype is a class, it may have properties. Properties of a stereotype are referred to as *tag definitions*. When a stereotype is applied to a model element, the values of the properties are referred to as *tagged values*.

**Figure 6.9. Device extended by Server stereotype with tag definitions and custom icon [UML-DIAGRAMS.ORG]**

3.1. **Stereotype application**

Profile diagram is used to show *definition* of stereotype. Stereotype is applied when it is used on use case diagrams, class diagrams, deployment diagrams, etc.

When a stereotype is *applied* to a model element, an instance of the stereotype is linked to an instance of the metaclass. The name of the applied stereotype is shown within a pair of guillemets («») above or before the name of the model element.

UML versions before 2.4 required the first letter of the name of the applied stereotype to be in lower case (e.g. «servlet»). Starting from UML 2.4, the first letter should normally be in upper case. Naming stereotype applications with lower-case letters where the stereotypes themselves are defined using upper-case first letter is still valid but is considered obsolete.

**Figure 6.10. Stereotype «Servlet» applied to the model element SearchServlet [UML-DIAGRAMS.ORG]**

If multiple stereotypes are applied to the same element, the names of the applied stereotypes are shown as a comma-separated list within a pair of guillemets.

When the extended model element has a keyword, then the stereotype name could be displayed close to the keyword, within separate guillemets (example: «device» «server»).
When a stereotype includes the definition of an icon, this icon can be graphically attached to the model elements extended by the stereotype. Every model element that has a graphical presentation can have an attached icon. When a model element is extended by one single stereotype the icon can be presented in a reduced shape, inside and on top of the box representing the model element.

**Figure 6.11. Servlet stereotype applied to the class SearchServlet** [UML-DIAGRAMS.ORG]

![Servlet stereotype applied to the class SearchServlet](image1.png)

When stereotype is applied, the whole classifier box can be replaced by enlarged icon of the stereotype.

**Figure 6.12. Servlet stereotype applied to the class SearchServlet** [UML-DIAGRAMS.ORG]

![Servlet stereotype applied to the class SearchServlet](image2.png)

Some model elements are already using an icon for their default presentation. A typical example of this is the actor model element, which uses the "stickman" icon. In that case, when a model element is extended by a stereotype with an icon, the stereotype’s icon replaces the default presentation icon within diagrams.

**Figure 6.13. «Web Client» stereotype applied to the Geek actor** [UML-DIAGRAMS.ORG]

![«Web Client» stereotype applied to the Geek actor](image3.png)

**Figure 6.14. Computer stereotype with tags applied to class** [UML-DIAGRAMS.ORG]

![Computer stereotype with tags applied to class](image4.png)

### 3.2. Stereotype relationships

A stereotype must always be used in conjunction with one of the metaclasses it extends. A metaclass may be extended by one or more stereotypes. Each stereotype may extend one or more metaclasses.
Stereotypes can participate in binary association. The opposite class can be another stereotype, a non-stereotype class owned by a profile or a metaclass. The stereotype must own property at the association end to be able to navigate to the opposite class. If the opposite end is not a stereotype, the opposite property must be owned by the association itself.

A stereotype may generalize or specialize only another stereotype.

Figure 6.15. Abstract stereotype Session EJB is specialized by stereotypes Stateless EJB and Stateful EJB [UML-DIAGRAMS.ORG]

4. Tag definition

Properties of a stereotype are referred to as tag definitions (or metaproperties).

Figure 6.16. Stereotype Computer with tag definitions for vendor, CPU, and memory [UML-DIAGRAMS.ORG]

4.1. Tagged value

Stereotype is applied when it is used on use case diagrams, class diagrams, deployment diagrams, etc.

When a stereotype is applied to a model element, the values of its properties may be referred to as tagged values.

UML 1.x defined tagged value as one of UML extensibility mechanisms permitting arbitrary information (which could not be expressed by UML) to be attached to models. Tagged value is a keyword-value pair that may be attached to any kind of model element.

The keyword is called a tag. Each tag represents a particular kind of property applicable to one or many kinds of model elements. Both the tag and the value are usually encoded as strings though UML tool allow to use other data types for values.

Tagged value specification in UML 1.x has the form

\[ \text{name} = \text{value} \]

where \text{name} is the name of a tag or metamodel attribute and value is an arbitrary string that denotes its value. For example,

\[ \{\text{author=”Joe Smith”, deadline=31-March-1997, status=analysis}\} \]
Boolean tags frequently have the form isQuality, where quality is some condition that may be true or false. In these cases, the form "quality" may usually appear by itself, without a value and defaulting to true. For example, {abstract} is the same as {isAbstract=true}. To specify a value of false, omit the name completely. Tags of other types require explicit values.

Tagged value (as well as metamodel attribute) is displayed as a comma delimited sequence of properties inside a pair of curly braces "{" and "}" (see figure).

**Figure 6.17. Stereotype Computer applied using "traditional" tag values notation [UML-DIAGRAMS.ORG]**

```
«Computer»
{Vendor = "Acer",
 CPU = "AMD Phenom X4",
 Memory = "4 GB DDR2"}
Aspire X1300
```

In UML 1.3 tagged values could extend a model element without requiring the presence of a stereotype. In UML 1.4, this capability, although still supported, was deprecated, to be used only for backward compatibility reasons.

Since UML 2.0, a tagged value can only be represented as an attribute defined on a stereotype. Therefore, a model element must be extended by a stereotype in order to be extended by tagged values. To support compatibility with the UML 1.3 some UML tools can automatically define a stereotype to which "unattached" attributes (tagged values) will be attached.

Tag values could be shown in class compartment under stereotype name. An additional compartment is required for each applied stereotype whose values are to be displayed. Each such compartment is headed by the name of the applied stereotype in guillemets.

**Figure 6.18. Stereotype Computer applied with tag values in compartment [UML-DIAGRAMS.ORG]**

```
«Computer»
Aspire X1300

«Computer»
Vendor = "Acer"
CPU = "AMD Phenom X4"
Memory = "4 GB DDR2"
```

Tag values could be shown in attached comment under stereotype name.

**Figure 6.19. Stereotype Computer applied with tag values in comment note [UML-DIAGRAMS.ORG]**

```
«Computer»
Aspire X1300

«Computer»
Vendor = "Acer"
CPU = "AMD Phenom X4"
Memory = "4 GB DDR2"
```

When displayed in compartments or in a comment symbol, each name-value pair should appear on a separate line.

**5. Extension**
An extension is an association relationship used to indicate that the properties of a metaclass are extended through a stereotype, and gives the ability to flexibly add stereotypes to classes and remove later, if necessary.

One end of the extension association is an ordinary property and the other end is an extension end. The property ties the extension to a metaclass, while the extension end ties the extension to the stereotype extending the metaclass.

Extension end is a navigable end, owned by extension. This allows a stereotype instance to be attached to an instance of the extended classifier without adding a property to the classifier. Note, that until UML 2.3 extension end was "never navigable".

The notation for an extension is an arrow with the filled triangle arrowhead pointing from a stereotype to the extended metaclass.

**Figure 6.20. Metaclass Class is extended by stereotype Customer** [UML-DIAGRAMS.ORG]

![Extension Diagram]

Because extension is subclass of association, it may have usual association adornments, but navigability arrows should not be shown. Adornments of an extension are typically suppressed.

A non-required extension means that an instance of a stereotype can be linked to an instance of an extended metaclass at will, and also later deleted at will. However, there is no requirement that each instance of a metaclass be extended. An instance of a stereotype is deleted when either the instance of the extended metaclass is deleted, or when the profile defining the stereotype is removed from the applied profiles of the package.

By default, extension is non-required. When extension has no adornments, it could either mean the default value or that the {required} adornment was suppressed. Multiplicity 0..1 on the extension end could be used as an alternative to non-required extension.

A required extension means that an instance of a stereotype must always be linked to an instance of the extended metaclass. The instance of the stereotype is typically deleted only when either the instance of the extended metaclass is deleted, or when the profile defining the stereotype is removed from the applied profiles of the package.

Required extension is shown using the {required} property near the extension end.

**Figure 6.21. Required extension of metaclass Component by stereotype WebService** [UML-DIAGRAMS.ORG]

![Required Extension Diagram]

It is also allowed to use multiplicity 1 on the extension end as an alternative to the {required}.

A metaclass may be extended by one or more stereotypes. This is obvious and expected. However, it may cause unexpected behavior, e.g., when stereotype Provider extends either (or both?) Interface or Class metaclasses. In such cases it should clearly be noted and documented when is Interface or Class used. In the case of SoaML (see next chapter) Interface is used for a non-composite service contract while Class is used for a composite service contract.

**Figure 6.22. Stereotype Provider extends either (or both?) Interface or Class metaclasses** [UML-DIAGRAMS.ORG]
6. Reference

Reference is import relationship represented by "metaclassReference" element import and "metamodelReference" package import.

The "metaclassReference" element imports and "metamodelReference" package imports serve two purposes:
1. they identify the reference metamodel elements that are imported by the profile and
2. they specify the profile’s filtering rules.

The filtering rules determine which elements of the metamodel are visible when the profile is applied and which ones are hidden.

Note that applying a profile does not change the underlying model in any way; it merely defines a view of the underlying model. In general, only model elements that are instances of imported reference metaclasses will be visible when the profile is applied. All other metaclasses will be hidden. By default, model elements whose metaclasses are public and owned by the reference metamodel are visible.

Figure 6.23. Metaclass Component is referenced (imported) by profile Servlets [UML-DIAGRAMS.ORG]

7. Profile application

Profile application is a directed relationship used to show which profiles have been applied to a package.

One or more profiles may be applied to a package that is created from the same metamodel that is extended by the profile. Applying a profile means that it is allowed, but not necessarily required, to apply the stereotypes that are defined as part of the profile.

It is possible to apply multiple profiles to a package as long as they do not have conflicting constraints. If a profile that is being applied depends on other profiles, then those profiles must be applied first.

When a profile is applied, instances of the appropriate stereotypes should be created for those elements that are instances of metaclasses with required extensions. The model is not well formed without these instances.
Once a profile has been applied to a package, it is allowed to remove the applied profile at will. Removing a profile implies that all elements that are instances of elements defined in a profile are deleted. A profile that has been applied cannot be removed unless other applied profiles that depend on it are first removed.

Applied profile is shown using a dashed arrow with an open arrowhead from the package to the applied profile. The keyword «apply» is shown near the arrow.

Figure 6.24. Profiles Java and Servlets applied to package WebApplication [UML-DIAGRAMS.ORG]

If multiple applied profiles have stereotypes with the same name, it may be necessary to qualify the name of the stereotype with a profile name.

8. UML profile diagram example

Here we provide an example of simplified and unofficial UML Profile for Enterprise JavaBeans™ (EJB) 3.0 with support for session, entity, and message-driven Enterprise JavaBeans.

UML 2.4 specification [UML2.4.1SUPER] provides an example profile for unspecified version of J2EE/Enterprise Java Beans (EJB) in Annex D.1. That UML sample profile is neither normative (official) nor complete, and is provided only as an illustration.

The EJB 3.0 specification defines both stateful and stateless session beans. There are differences in the API between stateful session beans and stateless session beans. The client of a session bean may be a local client, a remote client, or a web service client, depending on the interface provided by the bean and used by the client.

Figure 6.25. Simplified example of the unofficial Java EJB 3.0 Profile [UML-DIAGRAMS.ORG]
Stateful session bean represents a conversational session with a particular client. Such session objects automatically maintain their conversational state across multiple client-invoked methods.

An entity object represents a fine-grained persistent object. The client of an entity bean may be a local client or the client may be a remote client.

The message-driven bean class must implement the appropriate message listener interface for the messaging type that the message-driven bean supports or specify the message listener interface using the MessageDriven metadata annotation or the messaging-type deployment descriptor element.

The ejb-jar file must contain the deployment descriptor in the format defined in EJB Specification. The deployment descriptor must be stored with the name META-INF/ejb-jar.xml.
Chapter 7. Service-oriented architecture

Service-oriented architecture (SOA) is a way of architecting an enterprise application as a set of cooperating services that all the enterprise users want. The enterprise user can either be a human user or a client application. A good example is when you think of having a few associated software systems in the context of our case study like an inventory management system and a delivery system. These systems are independent systems (probably operated by different organizations) that need to interoperate in order to manage delivery of the ordered products to our customers. The entire process, from an initial order to delivery, can be managed by the interactions between several cooperating programs.

SOA is all about designing software as a set of cooperating services that can interact with each other through the Internet (mostly via the Web). The term service is essentially the business service. An enterprise application designed using SOA is typically composed of several services. Services have a loosely coupled nature that allows new services to be added and existing services to be modified (or even retired) quickly, based on the dynamic nature of the business situation.

However, the concept of SOA is not entirely new. The origin of SOA can be traced back to Remote Procedure Calls (RPCs), distributed object protocols such as CORBA and Java RMI, and component-based architecture such as Java EE/EJBs and (D)COM/COM+.NET. These technologies allowed to remotely access services that might even be written in a different programming language than the client. This allows to reach a high level of interoperability. Some of those technologies (e.g., DCOM and CORBA) did not require Internet or HTTP.

Still, HTTP played an important role in the proliferation of business-to-customer (B2C) transactions and plays an important role in business-to-business (B2B) transactions. Another standard called the Simple Object Access Protocol (SOAP) piggybacks on HTTP carrying the information in the form of a message, which can be synchronous or asynchronous in nature. Furthermore, in an enterprise scenario, existence of a combination of synchronous and asynchronous message exchange patterns is not uncommon. Another industry-standard advanced XML vocabulary called Web Services Description Language (WSDL) (pronounced “Wisdel”) enables you to describe the services an enterprise might offer. These services can be listed in a directory or registry known as the UDDI. The UDDI registry is like electronic “yellow pages” for worldwide services. These three standards (i.e., SOAP, WSDL and UDDI) are the cornerstones to the theory and practice of Web services.

According to the definition of James Bean, the author of the book entitled “SOA and Web Services Interface Design”, “A service-oriented architecture (SOA) is a combination of consumers and services that collaborate, is supported by a managed set of capabilities, is guided by principles, and is governed by supporting standards.” It is important to note that this definition does not mention any specific set of technologies. Technology capabilities are a requisite component of an SOA but SOA is not exclusively technology. Nor this definition is limited to Web services. The emphasis on Web services is from the perspective of standards-based interoperability. An effective SOA will allow participants and collaborators to interoperate. Web services simplify that interoperability by adding a standards perspective, where services and service consumers “speak” the same language.

Deeper details on SOA and Web Services are out of scope now but there are lots of books and online materials that can be consulted for more information.

1. SoaML UML profile

UML. Profile for Service Oriented Architecture is described in Service Oriented Architecture Modeling Language (SoaML) specification. "The goals of SoaML are to support the activities of service modeling and design and to fit into an overall model-driven development approach” [SoaML1.0.1]. The SoaML specification contains both a SoaML metamodel and a SoaML UML profile.

The SoaML UML profile supports modeling of service-oriented architectures, including specification of systems of services, specification of individual service interfaces, and specification of service implementations.

A service is defined there as a value delivered to another through a well-defined interface and available to a community (which may be the general public). A service results in work provided to one by another.
Service-oriented architecture

**Figure 7.1. SoaML UML Profile - Contracts**

Collaboration, Service Contract or Services Architecture represents a pattern of interaction between roles. In SoaML this interaction may be informal and loosely defined as in a requirements sketch. It may also represent formal agreements or requirements that must be fulfilled exactly.

Service description specifies how participants interact to provide or use a service. In SoaML there are three ways to specify a service interaction—UML Interface, a Service Interface and a Service Contract.

Consumer models the interface provided by the consumer of a service. The consumer of the service receives the results of the service interaction. The consumer will normally be the one that initiates the service interaction.

It looks weird that Consumer and Provider both extend Interface and Class metaclasses. The SoaML explains that Interface is used in the case of a non composite service contract while Class—in the case of a composite service contract.

**Figure 7.2. SoaML UML Profile - Services**

Participants are either specific entities or kinds of entities that provide or use services. Participants can represent people, organizations or information system components. Participants may provide any number of services and may consume any number of services. Participants provide or consume services via ports.
Agent is an autonomous entity that can adapt to and interact with its environment. Agent can be software agent, hardware agent, firmware agent, robotic agent, human agent, and so on.

Port is the part or feature of a participant that is the interaction point for a service—where it is provided or consumed. A port where a service is offered may be designated as a «Service» port and the port where a service is consumed may be designated as a «Request» port.

Request extends Port to specify a feature of a Participant that represents a service the Participant needs and consumes from other participants. The request is defined by a Service Interface. A request port is a "conjugate" port—the provided and required interfaces of the port type are inverted, creating a port that uses the port type rather than implementing the port type.

Service represents a feature of a Participant that is the offer of a service by one participant to others using well-defined terms, conditions and interfaces. A Service designates a Port that defines the connection point through which a Participant offers its capabilities and provides a service to clients.

Figure 7.3. SoaML UML Profile - Service Data

Message Type defines information exchanged between service consumers and providers. Message Type should generally only be applied to Data Type since it is intended to have no identity. However, SoaML recognizes that many existing models do not clearly distinguish identity, either mixing Class and DataType, or only using Class. Because of this, another odd thing is that SoaML allows Message Type to extend Class as well as Data Type.

Attachment is a part of a Message that is attached to rather than contained in the message.

2. Applying the SoaML profile using composite structure diagram of UML

Composite structure diagram is a type of static structure diagram that shows the internal structure of a classifier by using parts, ports, and connectors and the collaborations that this structure makes possible. The term "structure" for this type of diagrams is defined in UML as a composition of interconnected elements, representing run-time instances collaborating over communications links to achieve some common objectives.

A structured classifier defines the implementation of a classifier and can include a class, a component, or a deployment node. You can use the composite structure diagram to show the internal details of a classifier and to describe the objects and roles that work together to perform the behavior of the containing classifier.

A composite structure diagram is similar to a class diagram, but it depicts individual parts instead of whole classes. Before you can define the internal structure of a classifier, you must either show its structure compartment or open a composite structure diagram. You can then model the parts that represent the instances that the containing classifier owns. You can add connectors to link two or more parts in an association or dependency relationship.

In composite structure diagrams, ports define the interaction point between a classifier and its environment or between a classifier and its internal parts. You can use a port to specify the services that a classifier provides to and requires from its environment.

Created by XMLmind XSL-FO Converter.
You can also model collaborations and collaboration uses in composite structure diagrams. A collaboration describes the roles and attributes that define a specific behavior of the classifier. A collaboration use represents one particular use of the collaboration to explain the relationships between the properties of a classifier. To identify the roles of the parts in the collaboration use, you attach a collaboration use to a collaboration and then add the collaboration use to a composite structure diagram.

*Internal structure diagram* shows internal structure of a classifier—a decomposition of that classifier into its properties, parts and relationships. Figure Figure 7.4, “Shipping service with two customers” shows an example use of internal structure diagram. It utilizes the «Participant» stereotype defined in the SoaML profile and shows the various participants of an OrderingSubsystem as well as the interfaces they required or provided in accomplishing services. Note that the way how participants interact is not modeled in service participant diagram. This diagram focuses on the person, organization, system or anyone who take part in a services architecture.

**Figure 7.4. Shipping service with two customers**

SoaML supports both a contract and an interface-based approach to SOA. They differ in the way services are specified.

The interface-based approach involves the use of simple interfaces and service interface. Simple interface focuses mainly on one-way service delivery that requires no protocol between parties. Service interface allows for bi-directional services. Provider and consumer work together to complete a service.

Service interface diagram is a type of SoaML diagram specialized for the definition and specification of both simple interface and service interface. As an example, Figure Figure 7.5, "ShippingService service interface" applies the internal structure diagram to describe the ShippingService interface.

**Figure 7.5. ShippingService service interface**
The service contract approach defines the contract that specify how providers and consumers work together to achieve a goal, through the use of service. The service contract represents an agreement between parties for how the service is to be provided and consumed. Such agreement includes the interfaces, choreography and other terms and conditions.

Such a service contract diagram is designed for the specification of service contract by applying the Collaboration use diagram (which is also part of the composite structures diagram of the UML).

The behavior of the system is the functionality that the system under design will implement or which is already implemented by some existing system. Objects in a system typically cooperate with each other to produce the behavior of a system. A behavior of a collaboration will eventually be exhibited by a set of cooperating instances (specified by classifiers) that communicate with each other by sending signals or invoking operations. However, to understand the mechanisms used in a design, it may be important to describe only those aspects of these classifiers and their interactions that are involved in accomplishing a task or a related set of tasks, projected from these classifiers. Collaborations allow us to describe only the relevant aspects of the cooperation of a set of instances by identifying the specific roles that the instances will play. Interfaces allow the externally observable properties of an instance to be specified without determining the classifier that will eventually be used to specify this instance. Consequently, the roles in a collaboration will often be typed by interfaces and will then prescribe properties that the participating instances must exhibit, but will not determine what class will specify the participating instances. Collaboration use represents one particular use (occurrence) or application of the pattern described by a collaboration to a specific situation involving specific classes or instances playing the
roles of the collaboration. A collaboration use shows how the pattern described by a collaboration is applied in a given context, by binding specific entities from that context to the roles of the collaboration.

In this figure, Purchasing Service is an example of a collaboration name, buyer and seller are examples of roles of the corresponding role types Buyer and Seller (which are classifiers).

**Figure 7.6. Service interfaces of a compound service**

Message Diagram of SoaML is dedicated to model SoaML Message Types as well as their internals using links to classes representing the data model. This diagram covers basic cases of MessageType composition modeling. The data models should be modeled with UML class diagrams.

**Figure 7.7. MessageTypes in purchase order processing**
References

Books


Online sources


[OML2.4.1INFRA] OMG Unified Modeling Language (OMG UML), Infrastructure, Version 2.4.1. PDF.
References

[UML2.4.1SUPER] *OMG Unified Modeling Language (OMG UML), Infrastructure, Version 2.4.1.*  PDF.


[SoaML1.0.1] *OMG Service oriented architecture Modeling Language (SoaML), Version 1.0.1.*  PDF.


[SPARXSSTEMS.COM] *SPARXSSTEMS UML2 tutorial.*  Web page.