www.solms.co.za

Object-Oriented Analysis and Design using UML

Dr. Fritz Solms <fritz@solms.co.za>

Object-Oriented Analysis and Design using UML

by Dr. Fritz Solms

Table of Contents

List of Figures

List of Tables

List of Examples

Preface: About Solms TCD

1. Copyright

We at Solms Training, Consulting and Development (STCD) believe in the open and free sharing of knowledge for public benefit. To this end, we make all our knowledge and educational material freely available. The material may be used subject to the *Solms Public License* (SPL).

1.1. Solms Public License (SPL)

The *Solms Public License* (SPL) is modeled closely on the GNU, BSD, and attribution assurance public licenses commonly used for open-source software. It extends the principles of open and free software to knowledge components and documentation including educational material.

1.1.1. Terms

- “Material” below, refers to any such knowledge components, documentation, including educational and training materials, or other work.
- “Work based on the Material” means either the Material or any derivative work under copyright law: that is to say, a work containing the Material or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".)
- “Licensee” means the person or organization which makes use of the material.

1.1.2. Application domain

This License applies to any Material (as defined above) which contains a notice placed by the copyright holder saying it may be distributed under the terms of this public license.

1.1.3. Conditions

- The licensee may freely copy, print and distribute this material or any part thereof provided that the licensee

  1. prominently displays (e.g. on a title page, below the header, or on the header or footer of a page or slide) the author's attribution information, which includes the author's name, affiliation and URL or e-mail address, and
  2. conspicuously and appropriately publishes on each copy the *Solms Public License*.

- Any material which makes use of material subject to the *Solms Public License* (SPL) must itself be published under the SPL.
- The knowledge is provided free of charge. There is no warranty for the knowledge unless otherwise stated in writing: the copyright holders provide the knowledge “as is” without warranty of any kind.
- In no event, unless required by applicable law or agreed to in writing, will the copyright holder or any party which modifies or redistributes the material, as permitted above, be liable for damages including any general, special, incidental or consequential damages arising from using the information.

- Neither the name nor any trademark of the Author may be used to endorse or promote products derived from this material without specific prior written permission.

2. Overview

The training pillar of Solms TCD focuses on vendor-neutral training which provides both, short- and long-term value to the attendees. We provide training for architects, designers, developers, business analysts and project managers.

2.1. Vendor-neutral, concepts-based training with short- and long-term value

None of our courses are specific to vendor solutions, and a lot of emphasis is placed on entrenching a deeper understanding of the underlying concepts. In this context, we only promote and encourage public (open) standards.

Nevertheless, candidates obtain a lot of hands-on and practical experience in these open-standards based technologies. This enables them to immediately become productive and proficient in these technologies, and entrenches the conceptual understanding of these technologies.

When a “product” is required for practicals, we tend towards selecting open-source solutions above vendor products. Typically, open source solutions adhere more closely to public standards, and the workflows are less often hidden behind convenient wizards, exposing the steps more clearly. This typically helps to gain a deeper understanding of the appropriate technologies.

2.2. Training methods

We provide instructor based training at our own training centre as well as on-site training anywhere in the world. In addition to this we provide further guidance in the form of mentoring and consulting services.

2.2.1. Instructor-Led Training

Our instructors have extensive theoretical knowledge and practical experience in various sciences and technologies.

After familiarising ourselves with the individual skill and needs of candidates, we adapt our training to be most effective in the candidate's context - deviating from the set course notes if necessary.

Students spend half their time in lectures and half their time in hands-on, instructor-assisted practicals. Class sizes are typically limited to 12 to enable instructors to monitor the progress of each candidate effectively.

3. About the Author(s)

3.1. Fritz Solms


Fritz Solms is the MD and one of the founding members of the company.

Besides the management role, he is particularly focused on architecture, design, software development processes and requirements management.

Fritz Solms has a PhD and BSc degrees in Theoretical Physics from the University of Pretoria and a Masters degree in Physics from UNISA. After completing a short post-doc, he took up a senior lectureship in Applied Mathematics at the Rand Afrikaans University. There he founded, together with Prof W.-H. Steeb, the *International School for Scientific Computing* - developing a large number of courses focused on the immediate needs of industry. These include *C++*, *Java*, *Object-Oriented Analysis and Design*, *CORBA*, *Neural Networks*, *Fuzzy Logic* and *Information Theory and Maximum Entropy Inference*. The ISSC was the first institution in South Africa offering courses in *Java* and the *Unified Modeling Language*. During this period he was also responsible for presenting the OO Training for the Education Division of IBM South Africa.

In 1998 he joined the Quantitative Applications Division of the *Standard Corporate and Merchant Bank* (SCMB). Here he was the key person developing the architecture and infrastructure for the QAD library and applications. These were based on Java and CORBA technologies, with a robust object-oriented analysis and design backbone.

In 2000 he and Ellen Solms founded Solms TCD.

E-Mail:	`fritz@solms.co.za`
Tel:	011 646 6459
Mobile:	072 128 2314

3.2. Dawid Loubser


Dawid Loubser has a history of developing Java-based web applications in the financial industry, and specialises in Application Architecture, Graphical Inter-

faces (web and otherwise), and usability. In addition to object-oriented and web technologies, he also takes great interest in various graphical formats and frameworks - especially the application of XML for this purpose.

After a career at the JSE Securities Exchange (formerly *Johannesburg Stock Exchange*) as Systems Analyst and Web Architect, he joined Solms TCD in order to explore a larger field for applying interesting technologies to solve interesting problems, and to share this passion with others through mentoring and consulting.

E-Mail:	dawidl@solms.co.za
Tel:	011 646 6459

4. Solms TCD Guarantee

We hope that the course will comply with your expectations, and hopefully exceed it! Should you for some reason you feel that you are not satisfied with

- the course content,
- the teaching methods,
- the course presenter or
- any auxillary services supplied

Please feel free to discuss any complaints you may have with us. We will do our best to address your complaints. Should you feel that your complaints are not satisfactorarily addressed within our organization, then you can raise your complaints with:

- Professor W.-H. Steeb from the *University of Johannesburg*, Tel: `+27 (11) 486-4270`, E-Mail: `whs@rau.ac.za`
- Dr A. Gerber from the Computer Science Department of the *University of South Africa*, E-Mail: `gerberaj@unisa.ac.za`

At the time of writing we are in the process of obtaining *ISETT SETA* accreditation. Complaints can be raised directly to that institution.

Chapter 1. Introduction and Overview

1. Introduction to Object-Oriented Modeling

Object-oriented modeling is used in more and more disparate fields. Let us first look at what object-oriented modeling is and then at some benefits provided by object-oriented modeling.

1.1. What is Object-Oriented Modeling?

Object-Oriented Modeling is a way of modeling systems in a language natural to the system. The system, which itself is an object, is iteratively decomposed into objects (components) each with its own attributes and services. These components collaborate to provide the required functionality of the system.

Object-oriented modeling thus aims to structurally and functionally decompose a system into smaller units with less complexity and less responsibilities. These units should be as independent as possible and should preferable be testable outside the system. The system is then iteratively assembled from these tested components.

There is an overhead associated with developing and maintaining the models. In fact, in many projects the model has not been kept at the same maintenance level as the code, often because developers find it a burden to do that. But even in these projects the benefits from having based the system on an initial analysis and design usually justify the extra effort incurred.

1.2. What is Object-Oriented Analysis and Design?

Object-Oriented Analysis and Design (OOAD) refers to the application of object-oriented modeling to the analysis of systems or system requirements and to designing object-oriented systems fulfilling these requirements.

The system may be virtually anything including an organization (e.g. a company), a business unit, a software system which exists or one for which the requirements specification are written, a data warehouse or even a hardware system.

1.3. Why use Object-Oriented Modeling?

Object-Orientation (OO) has been the buzz word for more than a decade now. The initial expectations where perhaps inflated and many of the initial OO projects where either much more expensive than anticipated or failed completely. The OO technologies have, however, matured a lot and the delivery of good quality, cost-efficient products are today largely in the OO paradigm.

1.3.1. Clean conceptual modeling of domains

Object oriented models provide a farmework within which one can formalize clean, intuitive conceptualizations of domains.

1.3.2. Reducing Realization Costs

One of the main attractions of object-oriented methodologies is the promise of reduced realization costs. This is facilitated by

- improved communication between role players,
- complexity reduction for complex domains, and
- contract and component based approach to service providers.

1.3.3. Improved Communication between Role Players

Because OOAD languages (e.g. UML)

- aim to remain close to the problem domain
- allow for direct mapping onto various programming languages, particularly onto OO programming languages

they provide an ideal communication medium between the various role players in the system analysis and development process, from domain expert to analysts, designer, implementor and maintainer.

Graphical languages like the Unified Modeling Language (UML) allow for diagrammatic designs which provide selective views, focusing only on those elements which are relevant to those points one currently wants to illustrate.

1.3.3.1. Communication with Domain Experts

Of particular importance is the communication with domain experts who are in the best position to understand the system requirements. A domain expert is one who is an expert of the problem domain. Typically domain experts are found within the client company. They should be included formally or informally in the development team, at least during the requirements analysis stage.

Many studies have shown that typically around half of the system bugs are due to errors in understanding the system requirements and that these errors contribute more than 80% of the cost of system errors. It is thus critical that one obtains a complete and precise (unambiguous) understanding of the client's requirements and there is no better source for information regarding these than the domain experts.

Good object-oriented models are usually easily accessible to domain experts and they facilitate communication between domain experts, system analysts and developers.

1.3.4. Reducing Complexity

Systems are generally complex and a large part of the effort should be directed at reducing the intrinsic complexity. This is not only true for software systems but for systems in general. A modern motor car, for example, is in its entirety a very complex system. However, by viewing the car not as a single monolithic apparatus, but as a collection of components which interact with one another in standardized ways the perception of the system is a lot less complex. One can understand the functioning of the car at a higher, more abstract level by talking abstractly about the components without zooming into the design and implementation details of these components. These components can then be designed and developed by separate development teams residing very often in separate companies.

In a similar way complexity reduction in software systems can be achieved via a component based approach. Furthermore, object-oriented analysis and design techniques allow for selective viewing of system features. These selective views are incomplete views which only present those features of the system relevant to the current view.

1.3.5. Component Based Approach

In the previous section we touched upon how system complexity reduction has been achieved in the manufacturing world using a component based approach. A car manufacturer does not have to understand the implementation complexities of the electronic ignition or gearbox. He has to understand the specifications and the interface. Understanding is sufficient for judicious component selection and interfacing/assembling of components to produce the product.

In a similar way we are moving towards a component based approach in software development. This is facilitated by object orientation and various system integration methodologies (particularly CORBA) as well as business and presentation logic containers like servlet and EJB containers or the containers provided by .Net.

Each component deployed on a machine or in a component container complies to some specifications and has a well defined interface. The components themselves can be produced by component vendors. Such components may be bought from these component vendors. A component-based approach aims for increases in

- productivity,
- quality and
- re-use.

1.3.6. Increasing Productivity and System Quality

The industrial revolution of the 19'th century has shifted the production process from single producers developing entire systems from scratch to a component-based approach where systems are largely assembled from tested components. This has resulted in an increase in quality as well as in productivity. These are driven by specialization of manufacturers and by component reuse.

The component-based approach in software production is starting to result in a similar increase in quality and productivity. The components interface via predefined interfaces and comply to manufacturer specifications. They are tested thoroughly and software systems are, to a large extend, assembled from these tested components.

1.3.7. Reducing Risk of Failures

Traditionally a large proportion of system development projects simply failed. They did not deliver what the client really needed, were often error prone and hugely over budget.

The focus on communication with domain experts in order to understand the system requirements together with the component-based approach of assembling systems from tested components should significantly reduce the risk of failure.

The biggest risk is perhaps that of building the wrong system. This can be due to not understanding what the clients requested. It is, however, not uncommon that the client themselves do not understand what they actually need. OOAD can help to prevent both problems. It facilitates communication with clients and domain experts and it can help clients to understand their own needs.

Other factors which should help to reduce the risk of failure are

- **Incremental delivery.** The required functionality (all the use cases) are not delivered all at once but incrementally. This gives clients the chance to evaluate the completed modules and give early feedback.
- **Solving difficult problems first.** The difficult elements of a system should be attacked early. This makes certain that problems are identified early and allows more time to think about these issues. Furthermore, should a problem be unsurmountable, this will be apparent early in the process where the project cost is still relatively low. Also, once these have been completed the scheduling of delivery should be a lot more certain.
- **Testing.** Testing should be an integral part of the process starting with
  - testing of the business model and continuing with
  - testing of the requirements,
  - testing of the architecture and high-level design,
  - testing of the logical units at various levels of granularity and finally

- high-level system testing in a deployment environment.

1.3.8. Reduced Maintenance Costs

The reduced system complexity and a component-based approach should result in significant lower maintenance costs. The systems should be much more accessible to persons who were not part of the original development team and even for those who were, the model should help to understand the system once again.

Furthermore, often maintenance can be localized to within single components and sometimes these components can be simply exchanged for more modern components which may even be supplied by different suppliers.

1.3.9. Facilitating Dynamic Systems

Successful software systems generally live longer than anticipated as is seen by the large number of legacy systems in use. Over such long periods the environment can change substantially. The result is that the requirements for systems may change frequently. Often companies which can change their systems more rapidly and can hence deliver new products faster have the edge over those companies not in the position to do that.

Object-oriented modeling and in particular a component based approach where newer components can be plugged in to replace the older components can increase the flexibility of systems considerably. Also, the ability to understand a system from the design and to test the system after modification can increase the speed with which systems can be modified.

Furthermore, aspects of legacy systems can be re-used in a clean object-oriented way and integrated cleanly into modern systems via integration technologies like CORBA and SOAP.

1.3.10. Availability of Analysis and Design Tools

Finally, object orientation lends itself to developing powerful analysis and design tools. Many of these support the full software development cycle, including analysis, design, implementation, documentation generation and verification. They often support automatic code generation from designs and automatic design generation from code. Some of these support round-trip engineering which allows the user to make changes either on the code or on the design with the rest of the system being automatically updated. This ensures that design and code are always on the same maintenance level.

2. Overview of the Unified Modeling Language (UML)

2.1. What is the UML?

UML is the *Unified Modeling Language* which is a diagrammatic object-oriented modeling language. It hence uses diagrams to document an object-based decomposition of systems and to show the interaction between these objects and the dynamics of these objects.

2.1.1. Where does UML come from?

The first object-oriented modeling languages were developed in the mid 1970's. By the mid 1994 there were more than 50 object-oriented modeling languages most of which were only used by a small group of system designers. The most widely used methods were the *Object-Oriented Modeling Technique* (OMT) of Jim Rumbaugh, and co-authors [Rumbaugh-Blaha-Premerlani-1991], the object-oriented analysis and design method of Grady Booch [Booch-1994] and the *OOSE* (Object-Oriented Software Engineering) method of Ivar Jacobson [Jacobson-Christerson-Johnson-Overgaard-1992]. Other methods include *Fusion* developed at Hew-

lett Packard [Coleman-Arnold-Bodoff-1994] which is a fusion of the *OMT*, *Booch*, *CRC cards* and *Objectory* methods, and the Object Oriented Analysis and Object Oriented Design method (*OOA/OOD*) of Coad and Yourdon (see [Coad-Yourdon-1991a] and [Coad-Yourdon-1991b]).

In October 1994 Jim Rumbaugh and Grady Booch joined forces in order to unify the OMT and Booch methods resulting in a draft for the *Unified Method* as it was then called. In the second half of 1995 Ivar Jacobson joined the group, resulting in the unification of OOSE with the unified method. The result, which was called *Unified Modeling Language* or *UML*. UML is not simply a unification of the most widely used object-oriented modeling methods. It is a new modeling language built on a sound and thorough semantic basis upon which a consistent notation was developed. This foundation is the UML Meta-Language. It includes many concepts which have not been included in either of its predecessors, OMT, Booch or OOSE.

In 1996 several businesses joined as UML partners and actively participated in the formalization of UML 1.0. These include Digital Equipment Corp., HP, i-Logix, Intellicorp, IBM, ICON Computing, MCI Systemhouse, Microsoft, Oracle, Rational Software, TI and Unisys.

In January 1997 UML 1.0 was submitted to the Object Management Group (OMG) to be considered for adoption as a standard. The OMG in a non-profit organization formed for the promotion of object-oriented technology. It has now about 1000 member companies and its most notable publications are the specifications are CORBA, the Common Object Request Broker Architecture which forms a language and location transparent brokering service for object-oriented service requests (it is an OO middleware) and UML. UML is now public property and is maintained by the OMG. The current language specifications can be downloaded from http://www.omg.org.

2.2. What is object-oriented modeling?

Object-oriented modeling involves modeling based on object-oriented concepts. Object-oriented modeling is thus an approach where a system is modeled in terms of objects which have attributes and perform operations. If one object wants to make use of the services of another object it sends a service request message to that object.

The objects provide a natural way to distribute responsibilities among systems or components.

2.3. Why UML?

here are many reasons for using UML including

- A graphical view simplifies the understanding of system.
- UML diagrams provide a convenient communication mechanism between all role players in the software development field from the client to the business analyst to the architects, designers and developers. The improved communication usually
  - reduces the risk of failure,
  - improves productivity,
  - improves quality,
  - facilitates maintainability and hence dynamic systems and
  - usually results in reduced complexity.
- UML is an object-oriented modeling language supporting all the standard object-oriented concepts.

2.4. Applicability of the UML

The use of UML has spread quite rapidly from being used mainly for software design to being used in a wide range of areas including

- requirements analysis and specification,
- business analysis and modeling,
- data modeling and
- modeling of hardware (e.g. engineering) systems.

2.5. The UML diagrams

UML supports a range of diagrams including:

1. use-case diagrams,
2. class and object diagrams,
3. interaction diagrams including sequence and communication diagrams,
4. behavior diagrams including activity and state diagrams,
5. deployment diagrams.

These diagrams are used to generate a range of complimentary views,the sum-total of which defines the system one is modeling.

2.5.1. Use case diagrams

Use case diagrams show look at the system from an outsider (e.g. user) perspective. The system is treated as a black box and one solely identifies what the system is used for.

2.5.2. Class diagrams

Class diagrams are used to specify

- the structure of objects,
- the services offered by objects, and
- the relationship between instances of different classes.

2.5.3. Interaction diagrams

Interaction diagrams are used to document the way in which objects interact in the context of some work flow. UML provides two types of interaction diagrams, *sequence* and *communication* diagrams. Both are used to specify the *sequence of messages exchanged in the context of a specific scenario (example) of a work flow*.

2.5.3.1. Sequence diagrams

Sequence diagrams show the messages in time order. They are very intuitive and ideal for discussing requirements with clients or to acquire an initial understanding of the system one is modeling.

2.5.3.2. Communication diagrams

Communication diagrams, on the other hand, show the messages sent along message paths, i.e. in the context of the static structure in which they are exchanged. The time ordering is specified via a numbering system. Collaboration diagrams can help to make the transition between the dynamics of the model and the structure supporting the dynamics.

Note

Communication diagrams where known in UML 1.x as collaboration diagrams.

2.5.4. Behavior Diagrams

Recall that the state of an object was not only defined by the attributes of an object, but also by its current associations and its current activities. Hence any change in either, the attributes, the associations or the activities resembles a state transition.

2.5.4.1. State charts

State charts can be used to document the complete state dynamics of an object. They show the

- the activities and attributes of each state,
- the possible state dynamics,
- the events which cause the state transitions and
- the exceptions which may be raised as well as the way in which the exceptions are handled.

2.5.4.2. Activity diagrams

Activity diagrams represent really a subset view of state charts. They show only the activity aspects of states and the events which cause the transitions from performing one activity to performing another.

They can, however,show activities across objects while currently state charts are restricted in UML to show the state transitions within a single object.

2.5.5. Components and component diagrams

Component diagrams are no longer regarded as a separate diagram in UML 2. Instead class diagrams with the component stereotype are used. Nevertheless, one may still draw component diagrams which show the components of a system and the relationships between them (even though a separate notation is no longer used in UML 2).

2.5.5.1. What is a component?

A component is

- a high-level object which
- implements an interface laying down the contract for the services to be supplied by the component
- and is deployed in some environment which may be a business, a machine or a component container like
    - *business logic containers* like EJB containers hosting enterprise Java beans which resemble Java-based deployable business logic components, and

- *web containers* like servlet and JSP containers for Java-based deployable presentation layer components.

2.5.5.2. What is shown in a component diagram?

Component diagrams are really special class diagrams showing

- the high-level components of systems,
- the communication paths between them and
- the interfaces they implement.

2.5.6. Deployment diagrams

The final UML diagram is a deployment diagram. Deployment diagrams are really an extension of component diagrams adding deployment information to the components, i.e. *showing the nodes onto which the components are deployed* and the physical realization of the communication paths between the components. These nodes themselves could resemble

- *business sites* onto which business units are deployed,
- *computational resources* (i.e. computers) onto which software components are being deployed,
- *component containers* into which business-logic or presentation-logic components are being deployed, or even
- *machines hosting hardware components* like a kitchen appliance onto which you can deploy milk-shake makers, slicers, coffee grinders etc..

2.6. Views onto a context

UML provides a range of diagrams. Each diagram will itself provides a view on to the context one is modeling, adding some information to the complete definition of the context.

There is, however, considerable overlap between the information shown in the various diagrams, and it many people involved in object-oriented modeling advocate a set of quasi-orthogonal views, each of which may use multiple UML diagrams. We like to use the following four views:

- Use-Case view
- Responsibilities view
- Static View
- Dynamic view
- Deployment view

2.6.1. The Use-Case View

In the *use case view* we look at the context from the actors' perspective. An *actor* is an external object interfacing with the context. In the case where the context is a business, the actors may be the clients, the shareholders, the auditors and so on. In the case where the context is a software system,

the actors may be users or other systems our system is interfacing with.

The context being modeled is thus viewed as a black box, exposing only

- what the context used for and
- how the external objects, the actors, interact with the context in the context of these use cases.

Obviously we are going to use UML use-case diagrams for the use-case view. However, in addition to use-case diagrams we will also use

- **Sequence diagrams.** to document the sequence of messages exchanged between the context and its actors in the context of a scenario (an example flow) of a use case and
- **Activity diagrams.** to show the multiple paths for a use case.

In either case the context is still viewed as a black box, exposing only how it interacts with the external objects.

2.6.2. Responsibilities view

We use the responsibilities view extensively in the cotext of Use-case/Responsibities Driven Analysis and Design (URDAD). The responsibilities view

1. documents the responsibilities which need to be addressed when realizing a use case, and
2. assigns these responsibilities to core context components (at a particular level of granularity).

For the former we use use case diagrams showing the use case, the abstract collaboration and the responsibilities within responsibilities comments.

To show the responsibility assignments we use a class diagram showing the abstract collaboration, the components which partake in the collaboration and the responsibilities assigned to these components.

2.6.3. The Static View

The static view shows a structural decomposition of the context. It depicts the components of the context as classes and objects and the relationship between them. This includes specifying the attributes of each class and the services the class offers. The relationships between classes include generalization and specialization relationships (i.e. inheritance), composition and associations as well as more general dependencies between classes.

Static relationships in UML are graphically represented by UML *component diagrams*, *class diagrams* and *object diagrams*.

2.6.4. The Dynamic View

The dynamic view of the context shows the workflows of the context including

- how the context components interact with one another,
- the objects exchanged in these workflows,
- the activities performed, the various object states and the events which cause the state transitions.

One uses

- **Sequence diagrams.** showing the messages exchanged between components of the context and between these components and the actors,
- **Activity diagrams.** showing the activities done by the various components of the context in the context of a workflow as well as the events which cause the transition from one object performing a particular activity to another (or the same) performing another activity, and
- **State charts.** providing a state machine view onto the components of the context.

2.6.5. The Deployment View

The deployment view shows how the context is implemented in its environment. In particular, it shows

- the physical nodes onto which the components are deployed and
- the physical communication channels which realize the required communication paths between the components.

One uses UML deployment diagrams for the deployment view.

2.6.6. Views at different levels of abstraction and granularity

in each of the above views we will still look at the context in both, different levels of abstraction and different levels of granularity.

3. Introduction into the core object-oriented concepts

Object-orientation is used across business modeling, requirements analysis, architecture, system architecture, system design and system implementation, irrespective of whether the system is a software system, hardware system or even an organization.

3.1. OO as our natural language

When we construct sentences we use nouns, verbs, adjectives and adverbs. Let us first focus on the first three. An example sentence would be

The green car drives.

Here the noun, *car*, is an object which has the attribute *green* and performs the function (provides the service) *drive*.

The sentence is thus a statement about an object, a service it supplies and an attribute it has. We talk OO.

3.1.1. Mapping between natural language and OO

The nouns in our sentences (in our thoughts) typically refer to objects. An object has identity (it is unique and identifiable), has attributes and supplies services (i.e.\ can perform operations).

We can map

- *nouns* onto *objects*,
- *adjectives* onto *attributes objects*,

- *verbs* onto *operations performed by objects* or *services offered by objects*, and
- *adverbs* onto *attributes of operations*,

3.2. What is an object?

An object is a *identifiable physical or conceptual unit* which

- may have attributes,
- may offer services and
- may have message paths to other objects (e.g. to delegate some of the responsibilities which need to be addressed in the context of a service request to lower-level service providers).

3.3. Classes as abstractions of objects

The sentence

> *The green car drives.*

refers to a particular car -- a particular object. On the other hand, the statement

> *Cars drives.*

is more general and hence more abstract. It claims that the entire class of cars drives, i.e. that all objects which are instances of the class of cars drive. Hence classes are abstractions of objects.

Typically the concept of a class encapsulates the commonalities (common attributes and services) of all instances of that class. For example, a commonality of all cars is that they can all drive. Driving is thus a service offered by all cars. But all cars are not green. The particular car we referred to in the first sentence is green. Every instance of the class of cars has a color and a particular instance has a particular color.

3.4. Service request messages

We request services from an object by sending a message to it. For example, stepping on the accelerator sends an *accelerate* message to the car.

Within the context of individual service requests, the object requesting the service is called the *client* and the object providing the service is called the *service provider* or *server*. Clients can request a service from a service provider via different types of messages. One typically defines the following types of messages:

- synchronous
- asynchronous
- timeout

3.4.1. Synchronous messages

A synchronous message is one where the client waits for a response before continuing with its workflow. For example, when you request a telephone connection by dialing a telephone number, you wait until you received a response before starting to talk.

3.4.2. Asynchronous messages

An asynchronous message are messages which are sent by clients without them waiting for a response. The client sends the message and continues with his/her/its workflow directly. The response would be a separate message in the context of a *call-back*.

3.4.3. Timeout messages

Timeout messages are really synchronous messages where the client has finite patience, i.e. the client does not wait indefinetily for a response and *if a response is not obtained within the timeout period, the message is abandoned from the clients perspective*.

Note that the message is abandoned from the client's perspective. The service provider may still continue to try and provide the requested service.

3.5. Encapsulation

We need not know the way in which the service is supplied by the object. We only need to know what message we have to send to request the service. For example, we do not need to know the implementation details of the engine and gearbox to be able to request the accelerate service. Those implementation details are hidden from the user. The hiding of implementation details is called *encapsulation*.

3.6. Composition

An object is usually composed of a number of components which are themselves objects. The question which is not always trivial to answer is

> *What makes one object a component of another?*

To this end we need to understand the implications of a composition relationship. Composition implies

- **Ownership.** Linguistically one would say that the aggregate object (the container object) *has* a component. For example

  > *The car has an engine*.

  The container object must be able to say with a good conscience that the component is his/hers.

- **Adopting Responsibility for Component.** If a car is yours then you are responsible for it. For example, if the car crashes into a building because the brakes failed, then you are responsible for the damage. Similarly, if the car is yours, nobody should be able to make use of it or to change its state without your permission. We shall see later that a lot of object-oriented modeling revolves around judicious distribution of responsibilities.

- **Coincident life spans.** In object-oriented software systems composition may imply that the life span of the component is limited to that of its owner/container. For example, when an account is closed and deleted, its balance and account number will no longer exist.

3.7. Links and associations

In order to send a message to an object you have to have a link to it. Thus a *link provides a message path* between two objects. For example, my clutch pedal is linked to my clutch, enabling me to send service request messages to the clutch. Message paths are typically uni-directional. I can send messages from my clutch pedal to the clutch but the clutch may not be able to send messages back to me.

The message path itself is an object (it is a noun). For example, the clutch cable has attributes and

can provide services. It's abstraction is a class. On a class level one says that clutch pedals are associated with clutches. Associations between classes implies links between instances of the corresponding classes. For example, a particular clutch pedal is linked to a particular clutch.

The difference between composition and association is that the former implies ownership and coincident life spans. A good example illustrating the difference between the two is a financial contract like a bond. If you buy a bond from an institution (government or corporate) then that institution usually agrees to pay a specific interest rate on your investment over a certain period. For example, the R150 is a South African government bond which pays 12.5% interest biannually. The notional (i.e. amount invested) and the interest you earn are part of the contract. You cannot change these without changing the actual contract and hence these attributes are components of the contract. However, once you bought the contract, its value depends on environmental factors which are not part of the contract. In the case of a bond these would include the prevailing interest rates. Assume the remaining period of the bond is for 5 years. Then, if the interest you can earn by investing for 5 years increases (i.e. if rates go up), the value of your bond diminishes, because you could earn higher interest elsewhere. On the other hand, if rates fall, then the value of your bond increases because you could sell it at a higher price to somebody who wants to invest his/her money over the 5 year period at a higher rate than the prevailing interest rate.

Thus the interest the bond pays (i.e. the coupon) is part of the contract while the prevailing interest rate is part of the environment in which the contract lives. The relationship between the bond and its coupon is a composition relationship while you need an link between the bond and the environmental 5-year rate in order to value the bond. Once your contract is annulled, its coupon is annulled, but the 5-year interest rate prevailing in the market still exists. Also, the prevailing interest rate changes due to a series of factors, while the coupon can only be changed by changing the bon contract -- the bond takes responsibility/ownership of its coupon.

3.8. Abstraction via superclasses

We generally think and talk at various levels of abstraction. We may say that

> *Volvo's are reliable.*

referring quite abstractly to all Volvos. On the other hand we may be more specific, saying

> *The Volvo T80 is a large car.*

Or even more specific via

> *My Volvo T80 has given me excellent service.*

In the first example we talk about the *abstract class* of Volvos. Why is it abstract. Because you can't simply go into a car dealer and say you want to buy a Volvo. He/she will ask you to be more specific, i.e. to specify which model you would like to buy. Thus the class of Volvos is more abstract than the class of Volvo T80s. The class *Volvo* is a super-class to the class *Volvo T80*. Conversely, the class *Volvo T80* is a subclass of the class *Volvo*.

The class *Volvo T80* may still be abstract since the T80 may not specify the model completely, but a further subclass, say *Volvo T80E* is not abstract because you can instantiate it, i.e. they are actually manufactured. This class is a *concrete class*. Thus the difference between an abstract and a concrete class is that the latter can be instantiated, while the former cannot.

Talking about *my Volvo T80E* we are even more specific. Now we are not referring to a class anymore, we are referring to a specific instance of a concrete class. Thus, going from my Volvo T80 to the concrete class of Volvo T80s to the abstract class of Volvo's and perhaps further to the class of vehicles, we are becoming more and more abstract. Going in the reverse direction, each level is a specialization of the higher, more abstract level.

The superclass encapsulates attributes and services that are common among instances of itself (a superclass need not be abstract) and all its subclasses. For example, all Volvo T80s may be station wagons and may supply the *open back door* service via a common interface (the same switch, say).

Hence, these attributes and services can be specified at the more abstract level of *Volvo T80* and will be inherited among all Volvo T80s, e.g. Volvo T80Es and T80Ss.

3.9. Interfaces

An interface represents a client's view of a service provider. It encapsulates the services required by the client as well as the messages which the client will send when requesting the services.

Different service providers may implement the same interface and a client can switch between these service providers without changing the way in which the service is requested.

3.10. Polymorphism

Polymorphism in the OO sense is a direct consequence of the fact that you request services by sending messages, NOT by calling functions. The object which receives the message decides how it is going to realize the service, i.e. what function is going to be called. You as client simply send the service request message.

Simply think of Frank Sinatra's “I do it my way”. You can request a service from several objects, but of these service providers may realize the service in a different way.

3.11. The state of an object, events and state transitions

The state of an object is defined by

- the value of its attributes,
- services which are available while the object is in that state,
- the links it currently has to other objects, and
- the operations it currently is busy performing.

If any of these change the object undergoes a state transition. For example, my telephone has certain attributes (e.g. color), has a link to an exchange and might be in the idle state. If I paint it, or if it starts ringing or if I change the connection of the phone to another exchange, the telephone has undergone a state transition. If on, the other hand the state of the exchange changes (e.g. the number of connections it currently serves), the state of my telephone does not.

3.12. What is a component?

A component is a reusable and deployable object realising a *contract* defined by

1. an interface publishing the services offered by the component and the messages through which these services are requested,
2. the preconditions which must be satisfied before the service provider is willing to provide the service, and
3. the post-conditions (i.e. the deliverables) which are going to be provided by the service provider upon successful completion of the service.

4. Contract Based Approach

Whereever possible one would like to assume a contract-based approach where service providers

who offer services to clients do this in the context of a contract.

The contract is made up of

1. the interface which specifies

   - the message through which the service is requested including all objects/information clients must provide to service providers when requesting the service as input parameters,
   - the pre-conditions (i.e. the conditions under which the service provider will not provide the service) in the form of exceptions which notify the client that the service provider will not realize the service the client requested,
   - and the output parameters and return value which represent the deliverables the service provider will provide to the client upon successful completion of the service.

2. any other post-conditions which apply to the state of the service provider,

4.1. Example

Consider, for example, an internet banking portal. One of the services it offers may be that of making a payment to another party.

The interface specifies that

- the client must send a `makePayment` message providing the source account details, the destination account details and the transfer amount to the service provider,
- that the service provider may raise either an `InsufficientFunds` exception or a `DestinationAccountDoesNotExist` exception (the preconditions which must be met before a service provider ralizes the service are thus that the source account must have sufficient funds and that the destination account details are correct), and
- that the service provider will provide a `TransactionConfirmation` to the client.

Other post-conditions which apply to the final state of the service provider include that the balance of the account must have been adjusted accordingly, and that the transaction must have been entered into the transaction history of the sourcec account.

4.2. Benefits of a contract-based approach

A contract-based approach provides a range of benefits including

- the ability to smoothly replace one servicec provider with another fulfilling the same contract,
- the ability to test whether the way a service providers renders a service in such a way that the client needs are addressed.

  Note

  One will test that if a service is requested in a way as specified in the interface, with all preconditions having been met, that the service provider provides all deliverables/ post-conditions as per contract.

- the encapsulation of the client requirements in a contract which contains the information of the

exact dependencies of the client on the service provider, and

- an open competitative market for service providers which have an exact requirements specification for any services they wish to tender for.

5. Overview of CORBA

CORBA is a universal, non-proprietry standard for object-oriented middle ware. Similar to Java RMI, it supports remote service requests, but the service requests may be to CORBA objects which may be implemented in any language of choice. For example, CORBA can be used to wrap legacy COBOL-based mainframe systems and clients (for example Java and C++ clients) can access the services of the wrapped legacy system in an object-oriented, simple way without knowing the implementation language, platform or network protocols used by the server. CORBA has also standardized a wide range of general and industry specific services and is a universal standard for integrating systems.

5.1. A broker for service requests to objects

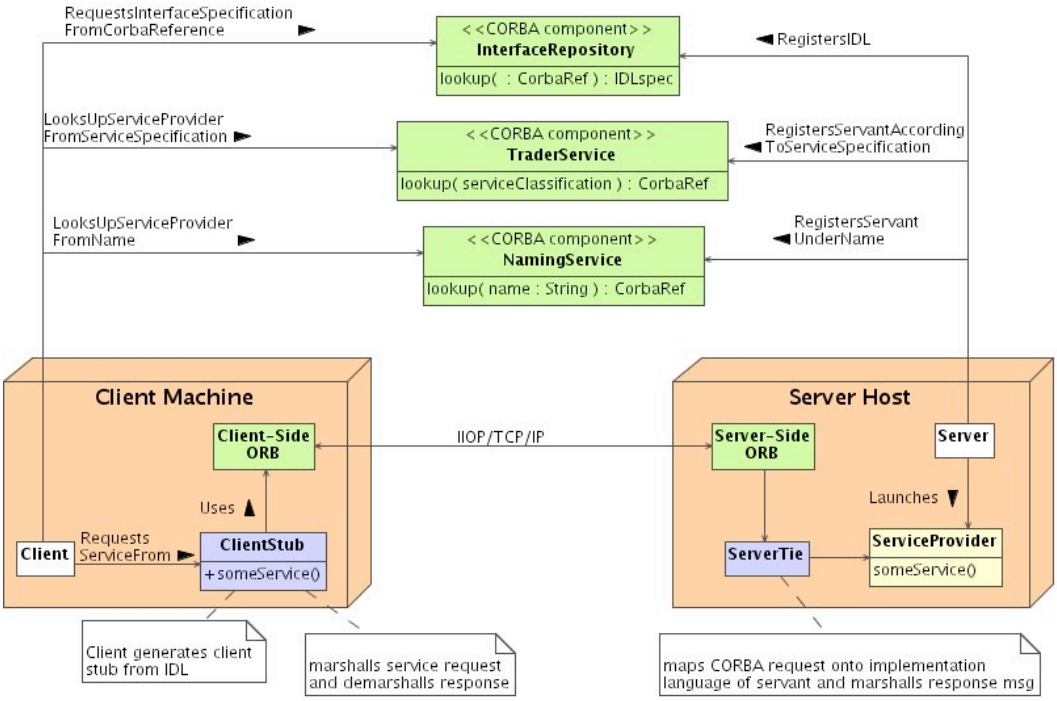
A CORBA client does not request a service directly from a service provider. That would expose the client to the technical implementation details of the service provider. Instead the client makes use of an Object Request Broker (ORB) which brokers the service request for the client. CORBA thus enables clients to request services from service providers without knowing

- the implementation language in which the service provider is developed,
- the location of the service provider, and
- the network protocols required to communicate with the service provider.

5.2. Overview of the Core CORBA Architecture

In Figure 1.1, “Overview of the core CORBA Architecture” we show the core CORBA architecture.

Figure 1.1. Overview of the core CORBA Architecture



Some of the core elements of CORBA are

- **GIOP and IIOP.** IIOP, the *Internet Inter-ORB Protocols* is CORBA's vendor-neutral service request protocol across the internet. It is based on the *General Inter-ORB Protocol* which may form a base protocol for other inter-ORB protocols using other transport layer protocols.
- **The client-side ORB.**
  - uses the CORBA object reference to establish a connection to the service provider via a server-side ORB,
  - marshalls service requests onto a IIOP/GIOP message and sends the message to the service provider and
  - receives and demarshalls the service provider responses and forwards them back to the client code.
- **The server side ORB.**
  - de-marshalls the service request and maps it onto a call in the server side implementation language.
  - marshalls the response with the return values back onto a IIOP message returned to the client code.
- **Interface repository.** provides a resource where service providers can publish their interfaces in a vendor and implementation technology neutral language, the IDL (CORBA's Interface Definition Language).
- **Naming Services.** provide a mechanism for obtaining object handles from names. A naming service is often compared to a normal telephone book.
- **Trader Services.** enable clients to find service providers from a hierarchical service classifica-

tion for the service they require. A trader service is often compared to a yellow pages telephone book which enables you to get handles to service providers (telephone numbers in that case) by looking up a service provider from the type of service your require.

5.3. CORBA is an object-oriented middleware technology

CORBA provides an object-oriented middleware solution. CORBA service providers publish there services in an object-oriented way, providing clients an object-oriented view supporting object-oriented features like persistent object references, encapsulation, and even polymorphism irrespective of whether the underlying implementation is object-oriented or not.

5.4. The CORBA object model

Assume the current realization of a particular CORBA server object is in the form of a COBOL mainframe implementation running on MVS. Assume the associated maintenance costs prompt the company to re-implement these services in, say C++ running on a Linux platform.

One dark night the COBOL/MVS implementation is brought down and the C++/Linux implementation is brought up. The CORBA implementation may be such that users continue using the same CORBA objects with the same object references (object identities) as though nothing has changed, even though we have a completely different realization of a service provider running on a different platform, at a different location and potentially using different network protocols.

5.4.1. The CORBA object is thus an object from the user's perspective

CORBA implements complete implementation hiding publishing only

- an abstract/virtual object handle and
- the interface containing the service request messages supported. For each service offered the interface contains
  - the service name,
  - the information which the client needs to provide to the service provider upon service request (the *input parameters*),
  - the *preconditions* under which the service provider offers the service,

    ### Note

    These are the *exceptions* which are potentially raised. The service provider will check the preconditions prior to providing the service and throw an exception if the service cannot be provided. Should the service provider not be able to check the preconditions up-front, the service provider will have to start a *transaction* which can be rolled back.

  - and the *deliverables* of the service, i.e. the *output parameters*.

    ### Note

    These deliverables are an essential part of the *post-conditions*. There may be further post-conditions around the state of the server side system after the service request. These should be added in order to have a complete requirements specification for a service.

But CORBA goes beyond implementation hiding, providing a *virtual object representation* whose realization may change at any stage. Nevertheless, from the client's perspective it is still the same object, e.g. the same account and the client continues to use the same object reference (message path).

5.5. CORBA wrapping

CORBA can be used to wrap legacy systems like, for example, legacy mainframe systems implemented in, say, COBOL. In this way the functionality in the legacy system can be published in a clean object-oriented way, hiding potentially dirty system interfaces under a clean object-oriented wrapper.

Note

It is important that the CORBA wrapper should be designed from a clean use-case/business perspective and should be a direct mapping of the current system interface onto CORBA.

5.6. CORBA object request brokers

There are well over 200 CORBA object request brokers out there. Most major software organizations have their own CORBA object request broker including IBM, Sun Microsystems, BEA, Oracle, Borland, Iona,

Part I. UML with Code Mappings

Table of Contents

Chapter 2. The Use-Case View

1. Introduction to Use Cases

The use case model has been developed in the Object-Oriented Systems Engineering (OOSE) method of Jacobsen and has been incorporated into the unified modeling language. Use cases describes uses of a context from the perspective of the user. The context itself is viewed as a black box, i.e. we do not look at how the context realizes the use case or at the structure of the context itself.

1.1. What is the context?

The context itself may be

- an organization or a business unit of an organization,
- a system which is to be acquired or built for which the requirements specification need to be understood,
- a software component or class which is being developed or even
- a hardware system for which is modeled.

1.2. What is a use case?

A use case represents a task performed by the system for the user. It

- represents one example usage of the context,
- must provide value to the user, and
- typically realizes one discrete goal for the user, i.e. a user should achieve his or her goal by using a use case, not by combining use cases.

Use cases are particularly important when developing a requirements model for the system (or class). By nature they specify what the system should be able to do, not how the system should do it. They are generally used to identify and document the requirements specifications of the system.

Use cases can be stated as text and/or can be graphically represented via Use Case Diagrams.

2. Objects and StereoTypes in use case diagrams

Use case diagrams show, at a very abstract level, which objects get what use out of which other objects. The objects are the system we are modeling and the external objects (organizations, persons, systems, ...) which interact with our system.

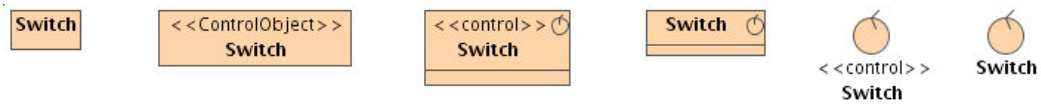
These objects (or their abstractions as classes) will be drawn using UML object and class diagrams. For this reason we will have to introduce in particular class diagrams in their simplest form already at this stage.

Recall that an *object* is an identifiable physical or conceptual unit which may have attributes and may offer services and have relationships with other objects. A *class* is an abstraction of an object which specifies all the commonalities of all the instances of that class.

For example, a particular switch is an object which has attributes (whether it is on or off), services (that the switch can be toggled) and relationships to other objects (for example a message path to the object being controlled by the switch).

For now we are not interested in the attributes and services. We simply want to show a class. A class is shown in UML as a rectangle with the name of the class in the center. (see the left diagram in Figure 2.1, “Simple class diagrams with stereotypes”).

Figure 2.1. Simple class diagrams with stereotypes



2.1. UML stereotypes

UML supports an abstract form of classification called a *stereotype*. For example, we may classify our switch as a control object, i.e. an object which is used to control other objects. A stereotype is specified in UML using French quotation marks, “guillements”. The second diagram in Figure 2.1, “Simple class diagrams with stereotypes” shows the switch being classified as a *ControlObject*.

Additionally, UML supports the definition of a stereotype icon. There are in fact some standard stereotypes, and that of a *control* object is one of them. The remaining diagrams show the stereotype icon with and without the definition of the stereotype within guillements.

Note that in the last two diagrams in Figure 2.1, “Simple class diagrams with stereotypes”, we collapse the class diagram, showing only the stereotype icon with the name of the class underneath.

Now, you may ask:

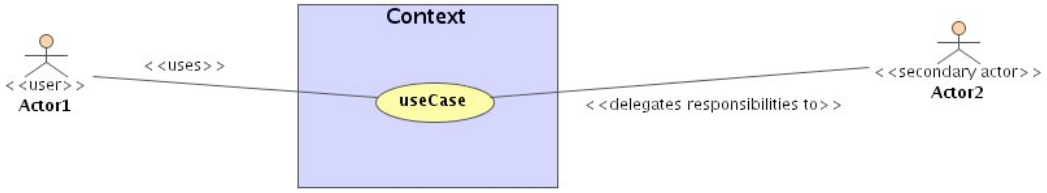
> What has this to do with use-case diagrams?

We shall see that the first notation is typically used for the system in a use case diagram while the last is typically used for the actors.

3. Simple use case diagrams

We use use case diagrams to show what the context we are modeling is used for. The context is treated as a black box, exposing only its uses.

Figure 2.2. Elements of a use case diagram

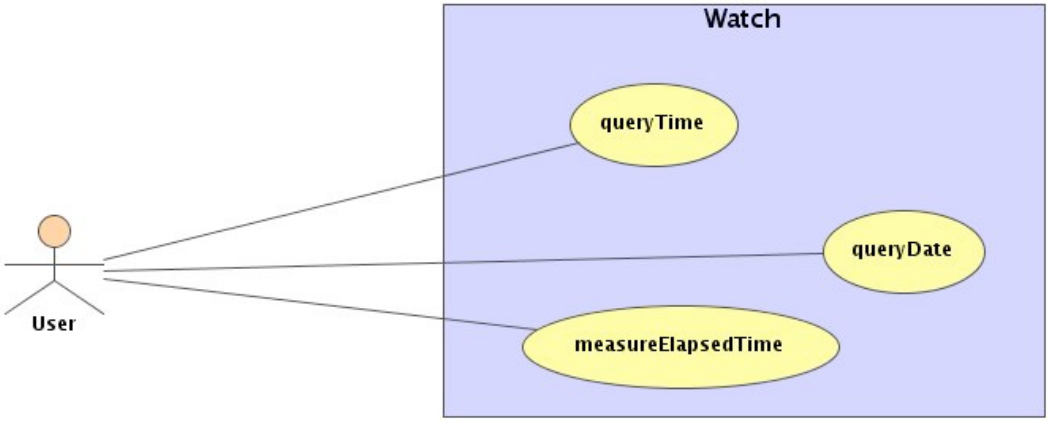


The elements of a simple use case diagram are

- the actors, i.e. the external objects interfacing with the system,
- the context responsible for realizing the use cases,

- the actual use cases and
- communication links between actors and use cases.

Figure 2.3. A simple use case diagram of a watch



3.1. Actors

The actors are *roles played by objects which interface with the context* in the context of a use case.

An actor is usually shown in a UML diagram using a stick man icon -- this is the default stereotype icon for actors. The stick man icon does not imply that the actor role will be fulfilled by a person. Even if the role is realized by a machine or a software component, one still may use the stick man stereotype icon.

Nothing prevents one, however, to define other icons for specialized types of actors. Stereotypes provide an extension mechanism in UML and one can thus define one's own stereotypes and assign one's own icons to them.

3.1.1. Users (primary actors)

In the context of a particular use case, one actor is usually the user or primary actor. The user gets the primary use out of a use case -- his/her goals are fulfilled with that use case and they receive a meaningful deliverable from the use case. Usually it is also the user who initiates and drives the use case.

3.1.2. Secondary actors

Secondary actors are used by the context. In order to realize a use case for its users, the context (system or organization) will have to address certain responsibilities. If some of these responsibilities are delegated (outsourced) to an external object which is not part of the scope of the context, then a secondary actor is used.

3.1.2.1. Example of user versus secondary actor

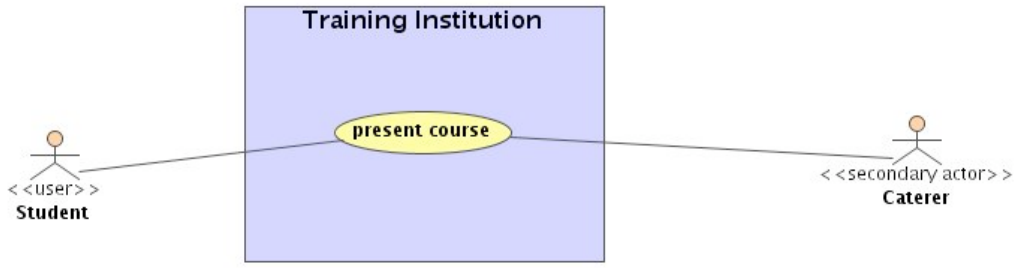
For example, in the context of a training institution offering a *present course* it may have to address the responsibility of providing lunches to the course delegates.

The training institution may decide to realize the lunches itself. In this case it would be addressed by the black box.

Alternatively, the training institution may decide to out-source the preparation of the lunches to an

external service provider, i.e. the responsibility of preparing the lunches is no longer within the scope of the training institution (the context). In this case it delegates this responsibility to a secondary actor who plays the role of a caterer.

Figure 2.4. A student (the user) uses a training institution (the context) for to present a course (a use case). The training institution delegates the responsibility of preparing lunches to a caterer (a secondary actor)

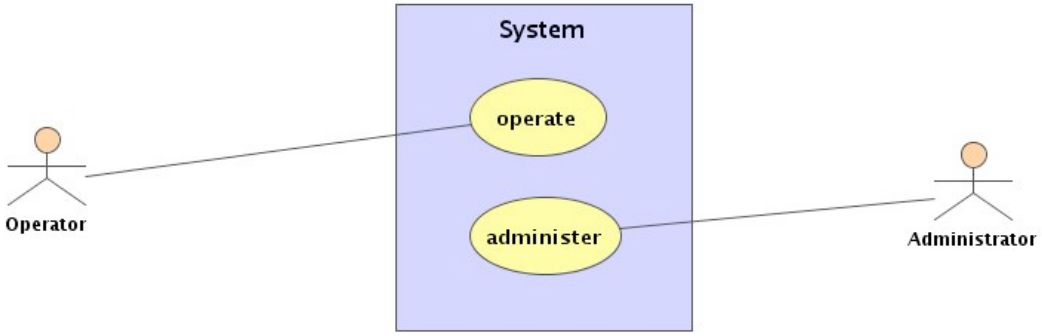


3.1.3. Actors as roles

Note that the same entity (e.g.the same person) may interface with a system under different roles.

For example, one may define for a system two actors, an *administrator* and a *operator*. The same person could, however, play at times the role of the administrator and at other times the role of a user.

Figure 2.5. The same person could play at times the role of an operator and at other times the role of a administrator



3.2. The context

The context is the subject being modeled. It offers the use cases to its actors and will ultimately realize the use cases. The context may be an organization, a software system, an mixed software/hardware system or anything else whose uses are modeled.

One uses a standard UML class diagram for the context. The name of the class follows the Camel naming convention, i.e. class names always start with capital letters while everything else starts with a lower case letter.

3.3. Use case

A use case resembles a coherent unit of functionality which realizes a potential use of the context

with observable deliverables for the actors. A use case must enable an actor to realize a complete user goal. The goal must be meaningful from the perspective of the user (primary actor).

A use case is shown in UML as an ellipse with the name of the use case inserted into it. The name starts with a capital letter which tells us that a use case should itself be seen as a class. This is because a usage represents a conceptual object. The ellipse is thus a stereotype icon for a use case.

3.4. Communication channels

The element aspect of a simple use case diagram is the association between an actor and a use case. This association implies that an actor is involved in that particular use case. The association resembles a communication channel implying that some information is exchanged between the actor and the context when the context is used for a particular use case.

Example 2.1. A use case diagram for an ATM

Let us, as a simple example, consider an automatic teller machine (ATM). The actors are

- a card holder,

  Note

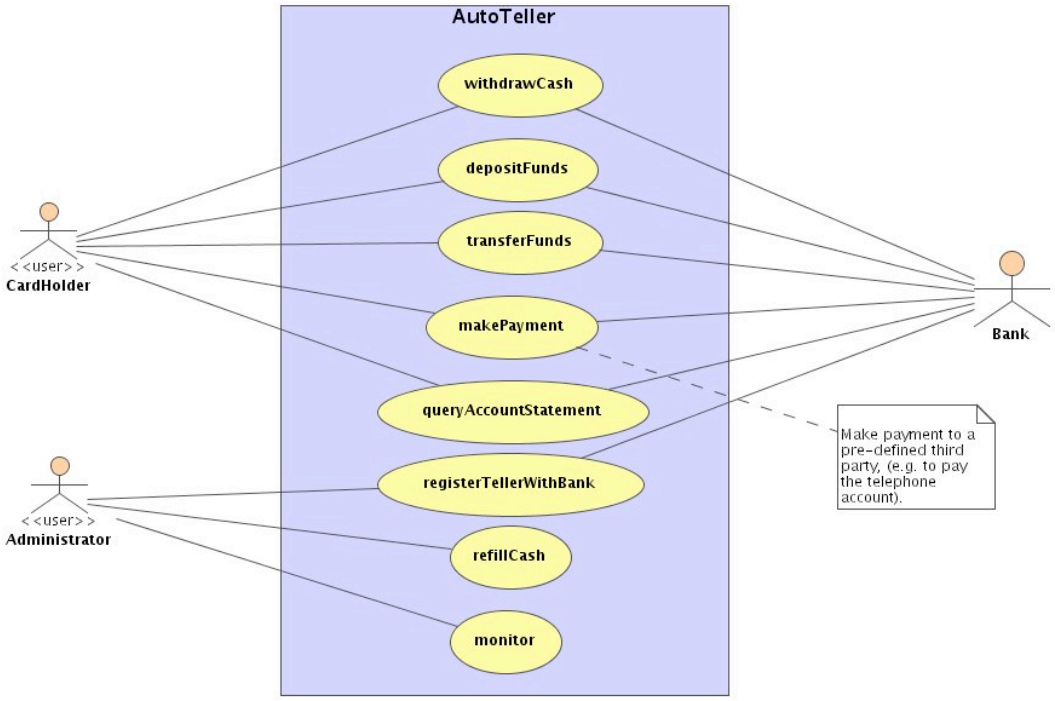
  This is not necessarily a customer: he or she may have picked up the card and would like to make use of it fraudulently. Having identified the actor correctly makes one more aware of the generality of the actor and may help to avoid neglecting aspects (like security) which are important for non-customers.

- the bank

  Note

  It would most probably be more realistic to factor out the role of an auto teller owner which may be the bank itself, another bank or even an independent company.

Figure 2.6. Use case diagram for an ATM



There are two actors involved in most of the use cases. The card holder is, for those use cases in which he/she is involved, the primary actor. He drives these use cases and gets the primary use out of them. The bank plays in these cases the role of a secondary actor who also may exchange messages and hence participates in the use case.

4. Notes in UML

UML supports the assignment notes. A note is an annotational element used to either

- attach a comment to some UML element or collection of elements or to
- specify a constraint for those elements.

A note is shown ion UML as a piece of paper with the top right hand corner turned over. The note is attached to a UML model element via a dashed line.

For example, in Figure 2.6, “Use case diagram for an ATM” we have attached a comment note to the `MakePayment` use case.

5. Relationships between use cases

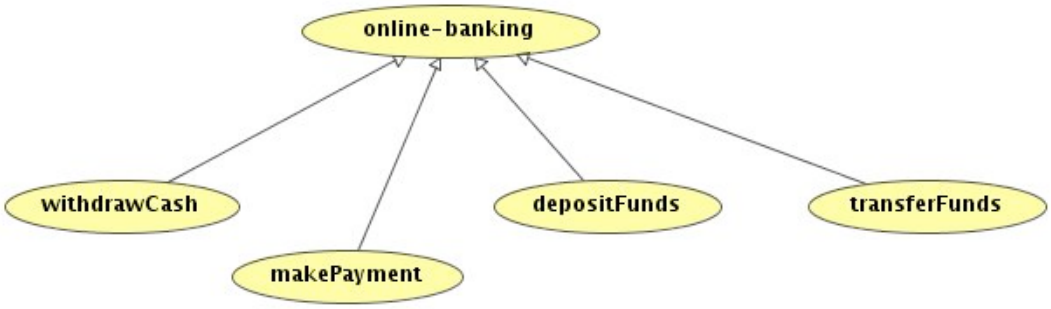
UML supports 3 types of relationships between use cases:

- **Use-case specialization.**
- **Use-case Extension.**
- **Includes relationship.**

5.1. Use-case specialization

A more specialized realization can extend a more general use case. Use case specialization enables us to look at the uses of a system at different levels of abstraction. We shall see the benefit of having a hierarchical structure of use cases.

Figure 2.7. A more general use case can have a number of specializations.



Specialization is shown in UML as a solid line with a triangular arrow pointing from the specialized entity to the more general entity.

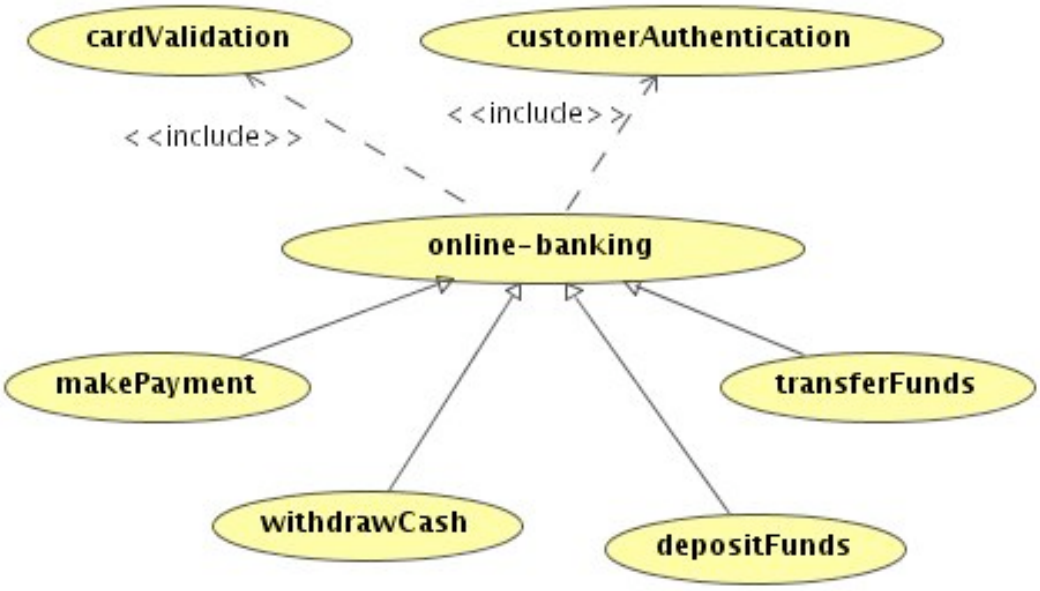
5.1.1. Abstract (high-level) uses cases and scoping

The high-level, abstract use cases ultimately specify the system scope. Any requirmenent for a new case must be validated against this high-level scope.

5.2. Includes relationships

One use case may include the behavior of another use case. This resembles an unconditional inclusion where the deliverables of the included use case are always supplied if the deliverables of the use case which includes it can be supplied (i.e. if the including use case is performed to successful completion).

Figure 2.8. One use case can include the behavior of other use cases.



An includes relationship is a special form of a *dependency*. Dependencies are shown in UML as a dashed line pointing from the entity which has the dependency to the entity it is dependent on. We show that a particular dependency resembles an *include* relationship by adding an *include* stereotype.

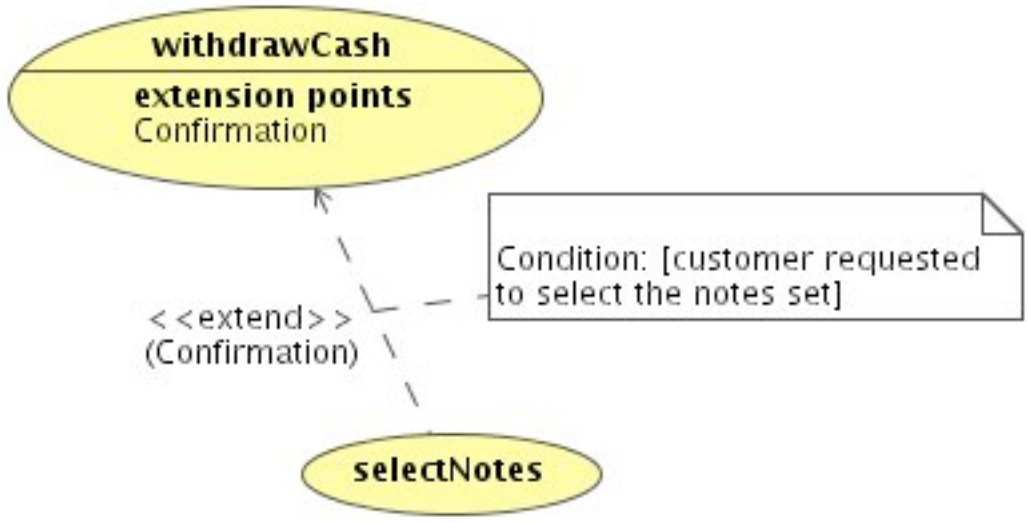
In the example shown in Figure 2.8, " One use case can include the behavior of other use cases. " the `PerformATMTransaction` use case includes the `CardValidation`, `CustomerAthentication` and `PrintTransactionSlip` use cases.

Consequently all its specializations like `WithDrawCash`, `MakePayment`, `DepositFunds` and `TransferFunds` will too.

5.3. Specifying use-case extensions

The behavior of one use case can extend that of another use case, typically fulfilling more actor goals and providing more deliverables. The extensions are thus accessed from within the context of the use case for which it provides the extension.

Figure 2.9. One use case can extend the behaviour of another providing more deliverables to the actors.



Extensions are, like include relationships, a special form of a dependency. This is because the extension is accessed from the context of the use case it extends. A *extends* stereotype is added to show that the dependency resembles an extension.

5.3.1. The use case extension

The extending use case provides optional additional behavior for a use case. The extending use case must, however be truly a use case in the sense that it does fulfill additional user goals and that it does provide meaningful additions to the deliverables of the use case it extends.

The use case extension is thus itself a use case and is shown with a standard use case diagram.

5.3.2. Extension points

An extension point specifies a point in a process which will realize the use case where the use case can be extended by the behavior of another. In our example we define the `Confirmation` extension point, i.e. that at the point where the card holder confirms the withdrawal transaction he/she can request to select the notes to be used for the pay out.

UML supports thus the definition of such extension points within a use case which is to be extended (see Figure 2.9, “ One use case can extend the behaviour of another providing more deliverables to the actors. ”). However, the existence of such extension points do not require that the use case needs to be extended before it can be used. The extended use case is a complete use case with or without extensions, even if it does define extension points.

5.3.3. Conditionality of an extension

An extension is conditional. If it were not, it would always be part of the use case it extends and in that case one could model it via an *include* relationship.

The conditionality of an extension is specified in a *condition note*. In our example

6. Actor specialization

UML also supports the concept of specializing actors. For this we use the standard specialization relationship,the solid line with the triangular arrow, between the actors.

Recall that actors resembled roles fulfilled by external objects in the context of using the system. Actor specialization resembles thus role specialization.

Figure 2.10. One actor can be a specialization of another.

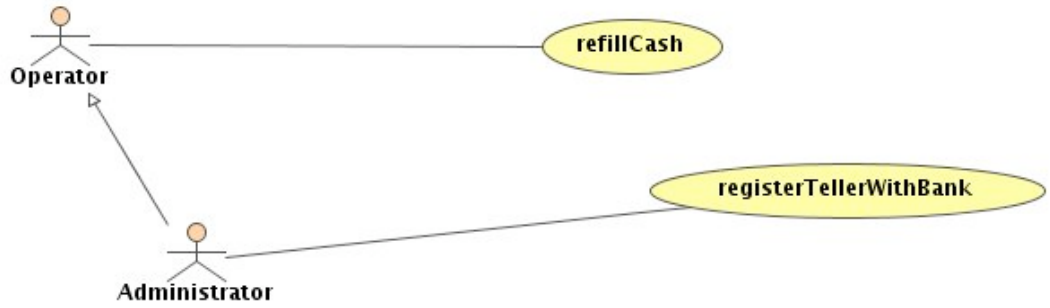


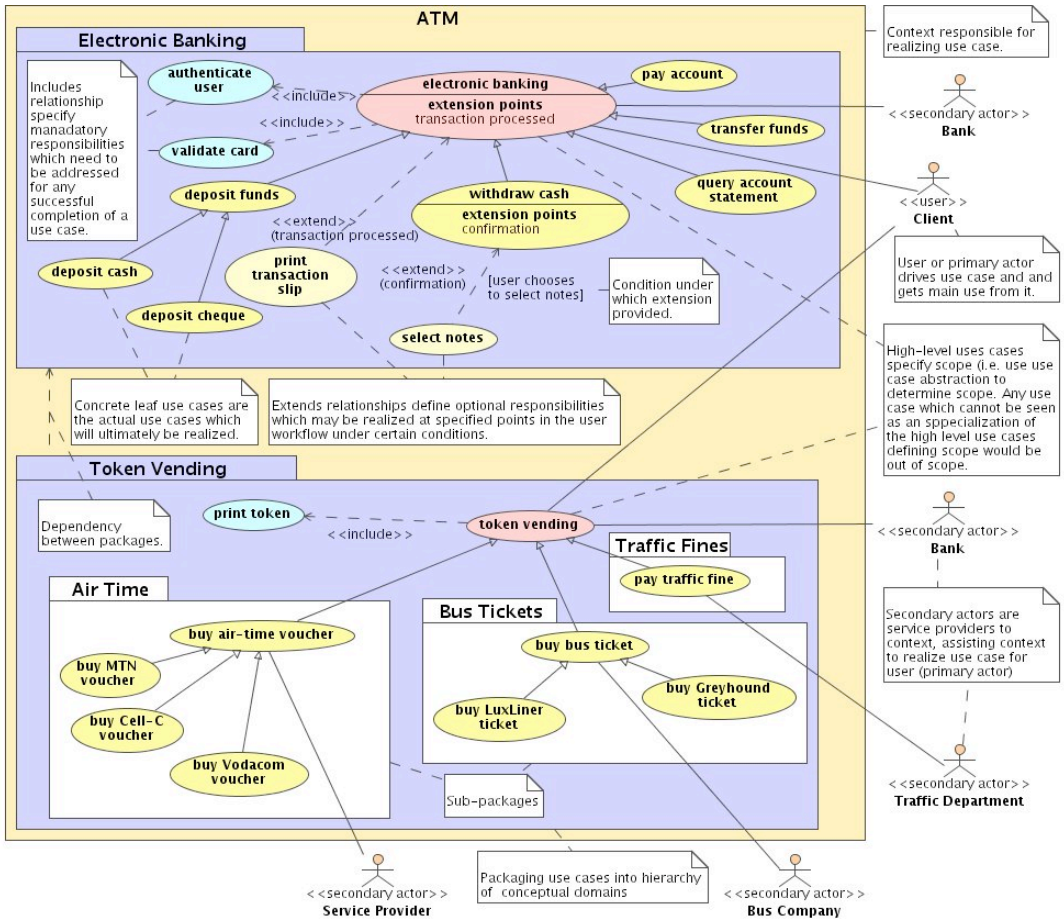
Figure 2.10, “ One actor can be a specialization of another. ” shows the `Administrator` role as a specialization of the `Operator` role, i.e. every `Administrator` is also an `Operator` and hence administrators will also make use of the use case `RefillCash`.

7. Packaging use cases

UML supports the concept of packaging. We shall look at packaging in more detail in Section 16, “Packaging”. For now we shall look at packaging use cases.

Packaging is a form of grouping. One typically groups related use cases together, i.e. use cases which cover a similar domain of responsibility.

Figure 2.11. Summary of use case diagram with packaging.



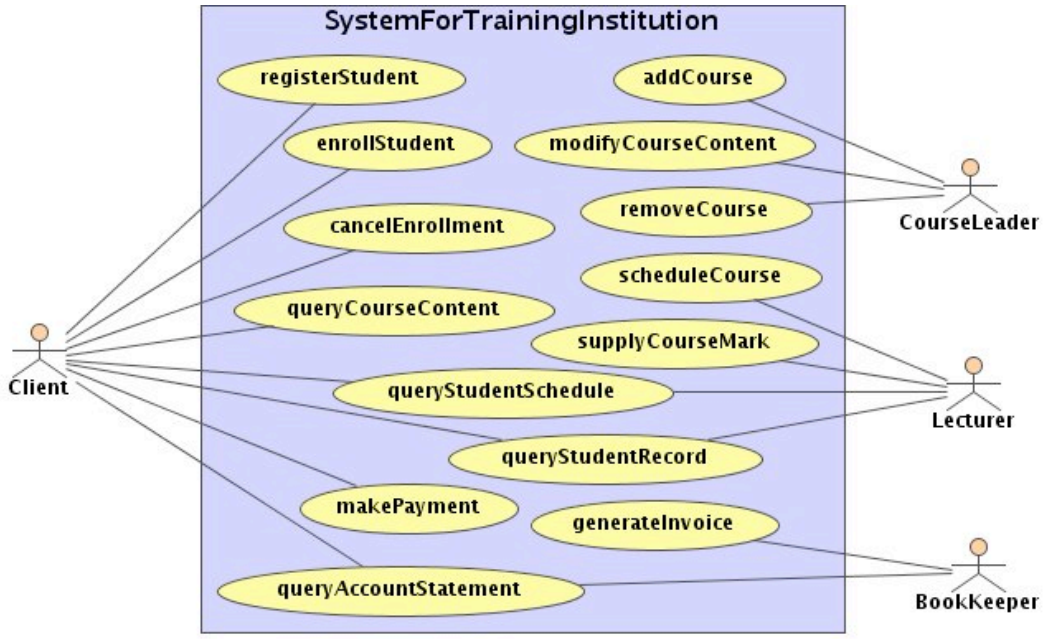
In Figure 2.11, “Summary of use case diagram with packaging.” we packaged the auto teller use cases across three packages:

1. **Customer services package.** This package contains the use cases resembling complete customer services. These are the use cases customers use the auto teller for.
2. **Low-level use cases package.** These are lower-level use cases which are either
   - included by the high-level use cases or
   - optional extensions to the high-level use cases.
3. **Maintenance package.** This package includes the use cases for maintaining the system.

8. Scoping

Even for simple systems one may very quickly end up with a very large number of use cases. For example, in Figure 2.12, “A typical example of a use case explosion.” we have a lot of at best semi-related use cases and it is not easy to understand the system responsibilities from this diagram. And, you can be certain, if we spend a little more time with the use case elicitation we will end up with many multiple times the use cases shown in Figure 2.12, “A typical example of a use case explosion.”.

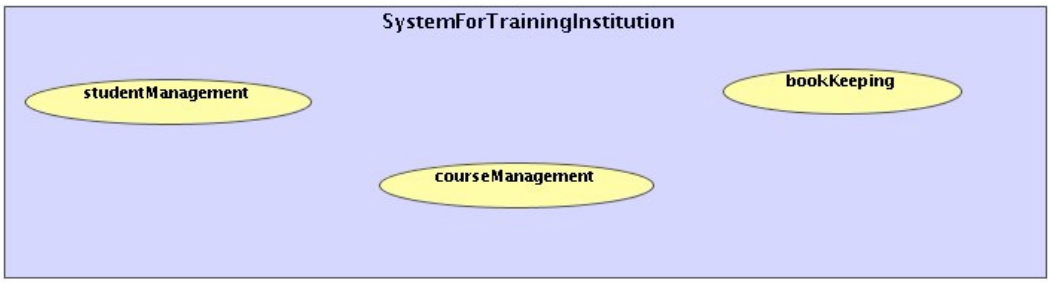
Figure 2.12. A typical example of a use case explosion.



For systems which one is not intimately familiar with, this development is virtually unavoidable. One simply needs to collect the bag full of use cases before one is able to develop a more abstract, higher-level understanding of the system one is modeling.

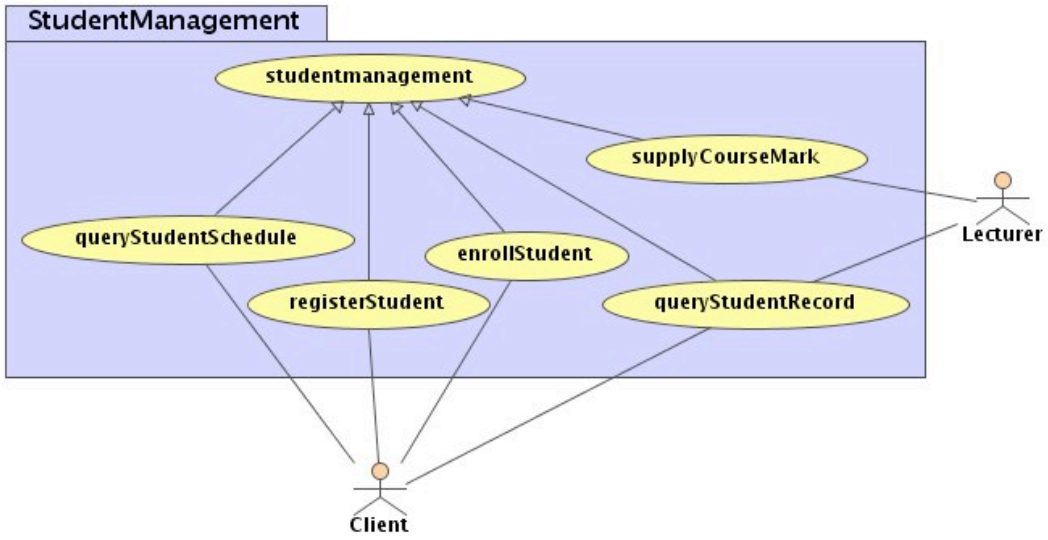
However, once one has collected the bag full of use cases, one should sit back in order to try and understand the system at a more abstract, higher level. After spending some time with the use case diagram shown in Figure 2.13, " Developing higher-level, more abstract understanding of a context identifies the scope of the context. ", one may come up with the following conceptualization of the system:

Figure 2.13. Developing higher-level, more abstract understanding of a context identifies the scope of the context.



We then show the concrete use cases as specializations of these more abstract use cases. We can also add packaging to modularize and through this simplify our understanding of the system.

Figure 2.14. Showing the concrete student management use cases.



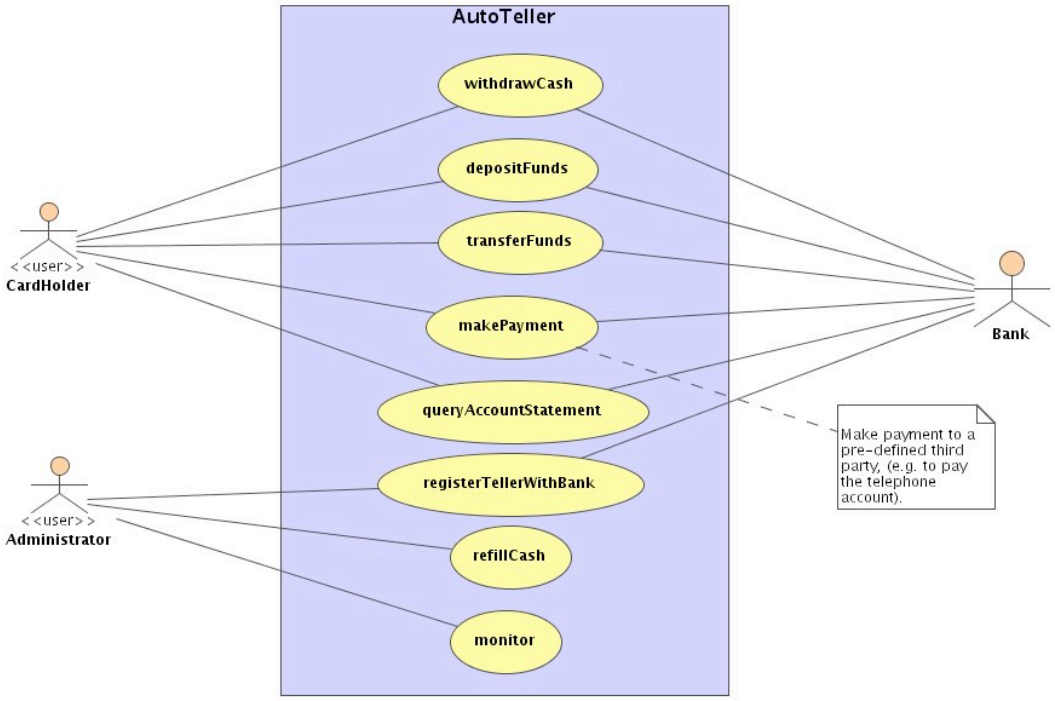
9. Using sequence diagrams to elaborate on use cases

Identifying the use cases is only one part of the requirements specification. Another aspect of the requirements specification is to specify how the context responsible for realizing the use case is to interact with the external objects, i.e. which messages are exchanged between the external objects and the system.

Sequence diagrams show how objects interact with one another in the context of a particular scenario (example usage) of a use case. They show sequences of messages exchanged between objects in time order. One places the objects which collaborate in the process one documents along the one axis (often the horizontal axis) and time along the other axis (usually time increases downwards along the vertical axis).

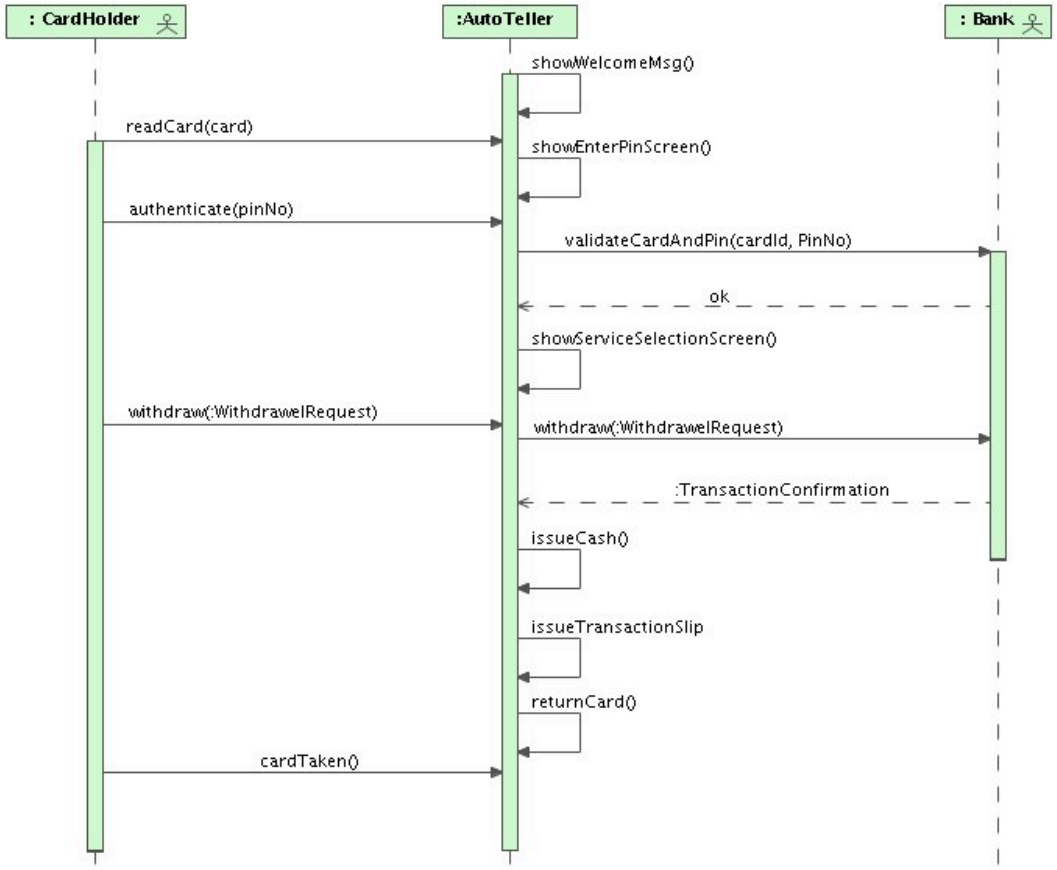
When using sequence diagrams to refine the use case view, the context is still treated as a black box. Hence it is not decomposed into components and the only other objects shown on the sequence diagram, bar the system itself, are those actors which participate in the use case one is modeling.

Figure 2.15. A use-case diagram for an auto teller.



Consider, for example, an auto teller. Figure 2.15, " A use-case diagram for an auto teller. " shows the use cases for our auto teller. For each use case one develops one or more sequence diagrams. In Figure 2.16, " A sequence diagram for the withdraw-cash use-case. " we show a sequence diagram for the *WithDrawCash* use case.

Figure 2.16. A sequence diagram for the withdraw-cash use-case.



9.1. Components of a typical sequence diagram

The objects are arranged on the horizontal axis. Time increases downwards. Below each object there is a vertical dashed line. That is the *life-line* of the object spanning the period (the vertical axis represents time) over which the object lives.

On top of the life lines we have bars. These bars are the *activity bars* representing the period over which the objects are active. At this stage this is irrelevant for us because the system is still treated as a black box and the behavior of the actors are outside the scope of our model (that is why they are actors).

9.1.1. Messages

Between the life lines (or activity bars) we show arrows representing the *messages* which are exchanged between objects. The message itself is shown as a label on the message arrow. *Service request messages* are shown as a solid line with an arrow pointing from the client to the service provider. The label contains

- the *name of the service,*
- the information provided with the service requests as *parameters* of the service request.

The sequence diagram in Figure 2.15, " A use-case diagram for an auto teller. " shows messages being sent from the card holder to the auto teller and the auto teller services from itself as well as from the bank. Note that the auto teller does not really have a message path to the card holder. For example, the *showWelcomeMessage* is shown, irrespective of whether there is a card holder in front of the auto teller or not. It is simply a request to the system itself to perform a actor-visible function.

Note

Even though we show the operations the context is to perform, we are still modeling the system as black box. The operations shown are the operations as they are to be perceived by the actors and does not constrain the realization of the use case any more than required by the client, (these operations are requirements specifications).

For example, we show that the card holder request a withdrawal of a certain amount from a specified account. How the amount is provided or how the account is to be selected is left to the design of the system.

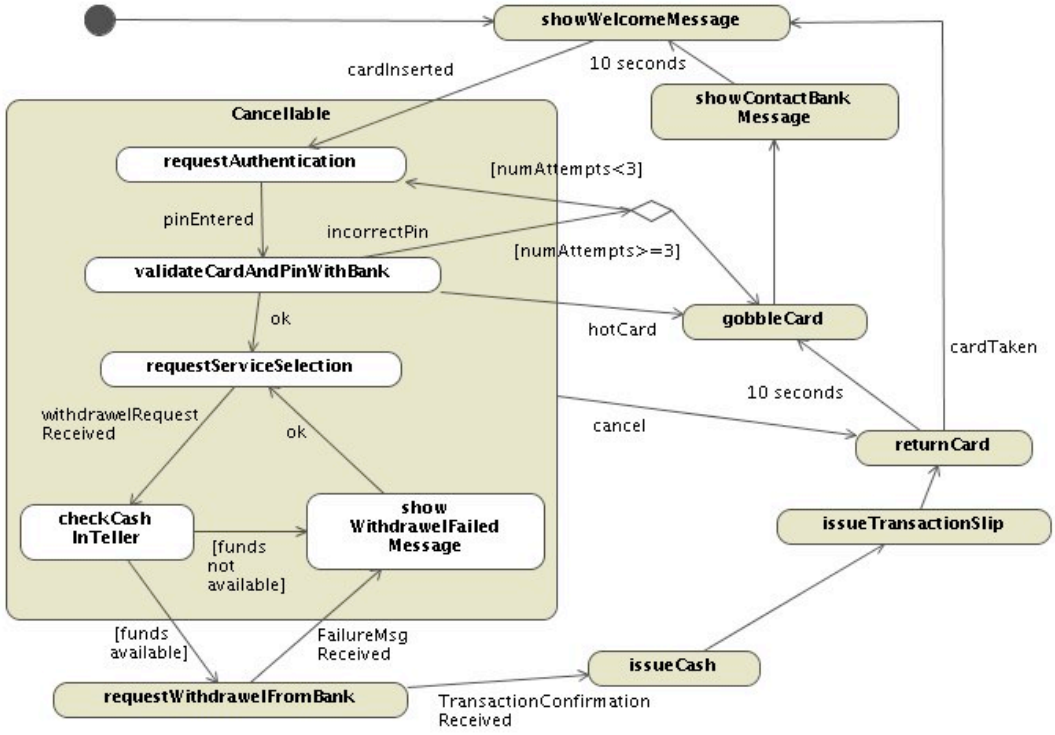
Similarly, we specify that a transaction slip is to be issued containing information from the transaction confirmation received from the bank. We don't specify how the transaction slip is to be generated.

10. High-level activity diagram for a use case

Recall that each sequence diagram documents a particular scenario,i.e. a particular example path through the system. There may potentially be very many scenarios (e.g. product not available, invalid tender, no change, ... for the vending machine's buy-product use case). Instead of drawing a large number of sequence diagrams, or alternatively leaving the system requirements as under-specified and open to different interpretations by architects, designer and developers, one can use an activity diagram to show the multiple paths.

While sequence diagrams focus on the interactions, i.e. the messages which are exchanged between objects,activity diagrams focus on the activities performed by objects, the transitions between performing one activity to performing another and the events which cause them

Figure 2.17. An activity diagram for an auto teller.



10.1. Example: Showing multiple scenarios for the

withdraw-cash use case of an auto teller

Figure 2.17, “ An activity diagram for an auto teller. ” depicts an activity diagrams which shows different scenarios in a single diagram activity diagram.

10.2. Components of a simple activity diagram

Activities are drawn in rounded rectangles. The arrows between the activities depict state transitions, which, in the context of an activity diagram which only shows the activity aspects of a state, is the transition between performing one activity to performing another.

A transition may have a label. The label may specify

- the event (if any) which causes the state transition,
- an activity (if any) which is done during the state transition and
- any messages sent to other objects during the state transition.

The signature for a general state transition label looks as follows:

```
eventName /activity ^messageSentToOtherObject
```

10.2.1. Automatic transitions

If there is no event on a transition label, then the transition is said to be an *automatic* transition. An automatic transition is one which is not induced by an event, but which is automatically executed after the activity of the preceding state is completed.

10.2.2. The context of the activities

Activities are performed by a context. For example, in our example the context is the system we are modeling, i.e.the vending machine. The context can be left unspecified as we have done or one can draw the activities into a class diagram for the context.

10.2.3. Entry states

A system is ill-defined if one does not specify an entry state. This is the default state the system will be in when switching the system on. It is denoted by a filled circle with an automatic transition to the entry state.

11. Mapping between sequence and activity diagrams

To map between from a sequence diagram onto an activit diagram we

1. Identify from which objects context we draw the activity diagram.
2. Map any incoming messages and returns (for that object) onto events.
3. Map any outgoing messages (including the messages the object sends to itself) onto activities.

12. Documenting Use Cases

Naturally use cases are documented using use case diagrams. This is done first at a very abstract level where the use cases are simply stated within use case icons.

For a particular instance usage (scenario) there is usually a sequence of messages which is exchanged between the product and the actors. These are naturally documented using sequence diagrams. Activity diagrams can be used to document multiple scenarios within a single diagram.

12.1. Allister Cockburn's use case template

Additionally a use case is usually accompanied by a textual description which can but need not necessarily include all the following aspects the aspects shown in Allistor Cockburn's use case template shown below:

1. **User (primary actor).** The person/system initiating and driving the use case. The user uses the context for the use case and gets the main use out of the use case (i.e. it is the user's goal the use case realizes).

2. **Secondary Actor.** Other external objects involved in the use case.

3. **Goal.** What goal is achieved by the use case.

4. **Context of Use.** In which context is the primary stake holder going to use the use case.

5. **Context.** The object (entity, system, subsystem, package, class, company, department) offering the use case. The context is responsible for realizing the use case for its users, but may delegate some of the responsibilities which need to be addressed when realizing a use case to secondary actors.

6. **Stakeholders and Interest.** A listing of all objects which have a stake (interest) in the use case and a description of that interest.

7. **Pre-conditions.** The conditions that must be met before the use case can be successfully completed and the user gets the use case deliverables.

8. **Post-conditions.** The deliverables provided by the system upon successful completion of the use case.

9. **Trigger.** The event which initiates the use case.

10. **Scenarios.** The sequence of messages exchanged between the actors and the system during a use case. Note that there are scenarios for successful and unsuccessful completion of the use case. The complete set of scenarios may be shown in an activity diagram.

11. **Use Case completion.** When is the use case completed.

12. **Priority.** The priority the use case from the perspective of the client (driver, constraint, important, optional).

13. **Constraints.** List each constraint for that use case. This should include the non-functional requirements like performance constraints, stability, scalability, usability, modifiability/maintainability, integrability constraints and so on.

14. **Channels to Actors.** For each actor list the communication channel(s) used to transmit the messages between the system and that actor.

15. **Open Issues.** Any issues about the use case which have not yet been resolved.

12.2. Using UML

A large proportion of the information in the template can be documented in UML. Below we show

how these elements are specified in UML.

- **Users and secondary actors.** The actors are of course shown in the use case diagram. A `<<User>>` or `<<Primary Actor` or stereotype may be added to define the concept of a primary actor in UML.
- **Context.** The context is directly shown in the use case diagram as the system or subsystem which contains the use cases.
- **Preconditions.** The preconditions can be specified in the use case diagram by attaching a *pre-condition constraint* on the the use-case.
- **Post-conditions.** Similarly, post-conditions for a use case can be specified by attaching a *post-conditions constraint* to the use case.
- **Trigger.** The trigger will be documented in the activity diagram as the event launching the process realizing the use case and in the sequence diagram as the first message coming into the system.
- **Scenarios.** These will be naturally documented in sequence diagrams and an activity diagram showing the multiple scenarios.
- **Use case completion.** Once again, this is shown in the sequence and activity diagrams.
- **Channels to actors.** The requirements specifications around communication channels to the actors are part of the deployment requirements and are best documented using an UML deployment diagram.
- **Priority.** This may be documented by attaching priority constraints to the use cases. Often one would document the priority also in the textual description for the use case.

So, nearly all information is naturally contained in the UML diagrams, and often in a clearer and more complete way then what the textual descriptions would (e.g. the guarantees for the individual scenarios). The remaining aspects which need to be documented in text include

- the primary actor's goal which is realized by the use case,
- the stakeholders and their interest in the use case,
- the priority of the use case and
- the non-functional requirements like the performance, legal, and maintainability requirements.

12.3. Example: ATM

Let us take the example of a client wanting to withdraw funds from an ATM. The use case could be described as follows:

1. **User.** The ATM card holder – typically an account holder at a bank.
2. **Secondary Actors.** The ATM company managing the ATM.
3. **Goal.** The card holder withdraws money from an account linked to the card and receives it.
4. **Context of Use.** The client of a bank needs cash.
5. **Context.** The ATM.

6. **Stakeholders and Interest.**

   - *Card holder:* Wants to withdraw funds from his/her account and receive it in cash.
   - *ATM company:* Wants to offer the use case and receive service fees from the client.
   - *Bank:* Wants to make certain that the client only withdraws funds which are available to the client.

7. **Preconditions.**

   - The card must be linked to an account at a bank which is registered with the ATM company.
   - The bank is prepared to provide the service..
   - The ATM must have sufficient funds.
   - The ATM must be able to establish communication to the ATM company.
   - The ATM company must be able to establish communication to the card holders bank.

8. **Post-conditions.**

   - The card holder is given the withdrawel amount in cash.
   - The card holder has been provided with a transaction slip (either on the user interface or a printed copy).
   - The card is returned to the card holder.

9. **Trigger.** A user inserts a card into the ATM.

10. **Scenarios.** Here show sequence diagrams for the messages exchanged during a successful withdrawal as well as for unsuccessful attempts due to

    - Invalid pin number.
    - Insufficient funds in ATM.
    - Insufficient funds in account.
    - Communication to bank cannot be established.
    - User aborted transaction.

11. **Use Case completion.** The card is returned to the user or the card is withheld by the ATM and the user is asked to contact her/his bank.

12. **Priority.** Driver.

13. **Constraints.**

- **Performance constraints.** All responses from the bank should be received within at most 10 seconds.
- **Maintainability constraints.** The auto teller must be able to print five thousand transaction slips before requiring paper or ink refilling.
- **Usability constraints.** 90% of the users must be able to effectively use the system without any prior training.

14. **Channels to Actors.**

    - Screen and keypad to user.
    - Communication channel to bank.

15. **Open Issues.** What happens if the ATM encounters a power failure during a transaction?

13. Exercises

1. Develop a use case model for a system for an e-commerce retailer like *Amazon*. In this context you should:

    - Identify and name the context (system, module, organization, business unit, ...) for which you are developing the requirements and provide a one or two sentence description of the context.
    - Identify the users (primary actors) which will make use of the context (e.g. the users which use a system for the system use cases or the clients which use your organization for certain client use cases).
    - From the perspective of the user(s), identify the concrete leaf use cases (the actual use cases used by the users).
    - Identify any secondary actors, i.e. other systems or users the system is required to interface with in the context of realizing a use case or any other organizations your organization will interface with (e.g. in the context of outsourcing some responsibilities) in the context of realizing the use cases for its clients.

    ## Note

    One should only identify those actors which the system is required to interface with. During design (system design or business process design) the designers may introduce further secondary actors in order to delegate some responsibilities to them. This will, however, be a design decision and it is not a user/client requirement that these actors are used.

    - Identify use case abstractions using specialization relationships. The highest level (most abstract) use cases will define the scope of the system or organization.
    - Identify the core quality attributes for the system or the organization. These may include performance, reliability, scaleability, modifiability, security, usability, integrability, portability, ...

Note

These quality attributes will be used to design the architecture of the system or the organization.

- The requirements will typically be realized iteratively by introducing the system or business processes which realize use case for use case. The detailed requirements of each use case are, in the context of iterative use case realization or iterative development, themselves elicited, documented and verified iteratively. To this end select one concrete leaf use case and perform the following steps for that concrete leaf use case:

    a. Identify the responsibilities/deliverables the system/organization has to address/realize for its users/clients.

Note

The required responsibilities are documented in UML using include relationships between use cases.

    b. Identify the optional responsibilities/deliverables.

Note

Use extends relationships.

    c. Document the typical user/client work flow for that concrete leaf use case.

Note

Use a sequence diagram for this.

    d. Document the use/client work flow for that use case in general.

Note

Use an activity diagram for this showing the activities of the system/organization as perceived by the user/client.

    e. For that concrete leaf use case show the structure of the entity/information objects exchanged between the actors and the context (system or organization).

Note

Use UML class diagrams with only attributes and composition/aggregation relationships between classes. *(This step you will only be able to do once we have covered UML class diagrams.)*

    f. For each secondary actor, introduce a contract for the service requests/messages that service provider needs to be able to process. *You will use an interface diagram with pre- and post-condition constraints on each service. (This step you will only be able to do once we have covered UML class diagrams.)*

    g. Fill in Allister Cockburn's use case template for that concrete leaf use case.

Chapter 3. The Responsibilities View

1. Introduction

The next step would be to look at a decomposition of the system into its core components and look how these core components interact to realize the use case. Those components can be business units or system components. In either case,we are no longer treating the system as a black box and we are going to expose design information.

UML provides a notation for introducing the abstract concept of a collaboration in a diagram. A collaboration resembles a collaboration of objects in the context of realizing a use case. A collaboration is drawn as an ellipse with dashed lines instead of a solid line. An abstract collaboration can be used in use case and class diagrams.

2. Identifying the core responsibilities for a use case

Before we can allocate responsibilities across core components of the system, we will have to identify these responsibilities. In UML this is done by showing, within a use case diagram, an *abstract collaboration* which represents the abstract concept of some collaboration of some objects which will realize the use case. The responsibilities are then documented via responsibility comments which are attached to the collaboration.

In Figure 3.1, " Identifying the responsibilities which need to be addressed for the withdraw-cash use-case. " we identify the core responsibilities which need to be addressed for the *withdraw-cash* use case.

Figure 3.1. Identifying the responsibilities which need to be addressed for the withdraw-cash use-case.

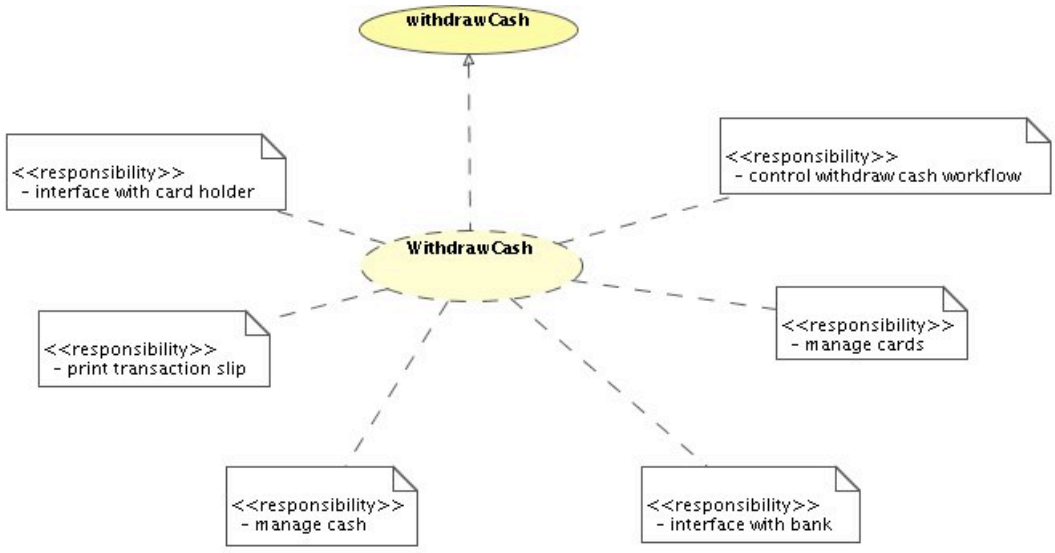
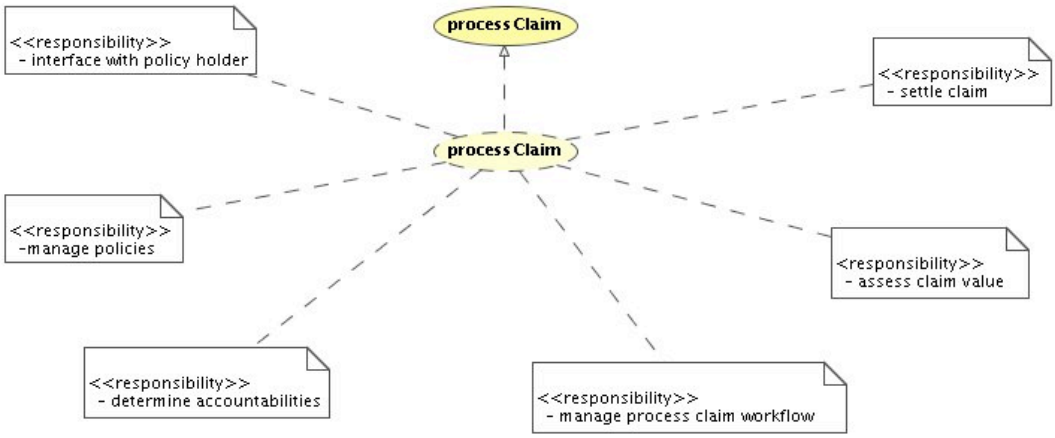


Figure 3.2. Identifying the responsibilities which need to be addressed for in the context of processing an insurance claim.



Example 3.1. Responsibilities of a process claim use case of an insurance company

As a second example, let us look at the use case of processing an insurance claim for a vehicle accident as realized by an insurance company. This is done in Figure 3.2, " Identifying the responsibilities which need to be addressed for in the context of processing an insurance claim. ".

3. Allocating responsibilities to core components

Having identified the responsibilities, we can now go ahead and assign them to core system components or business units.

Figure 3.3. Allocating the responsibilities which need to be addressed for the withdraw-cash use-case to core system components.

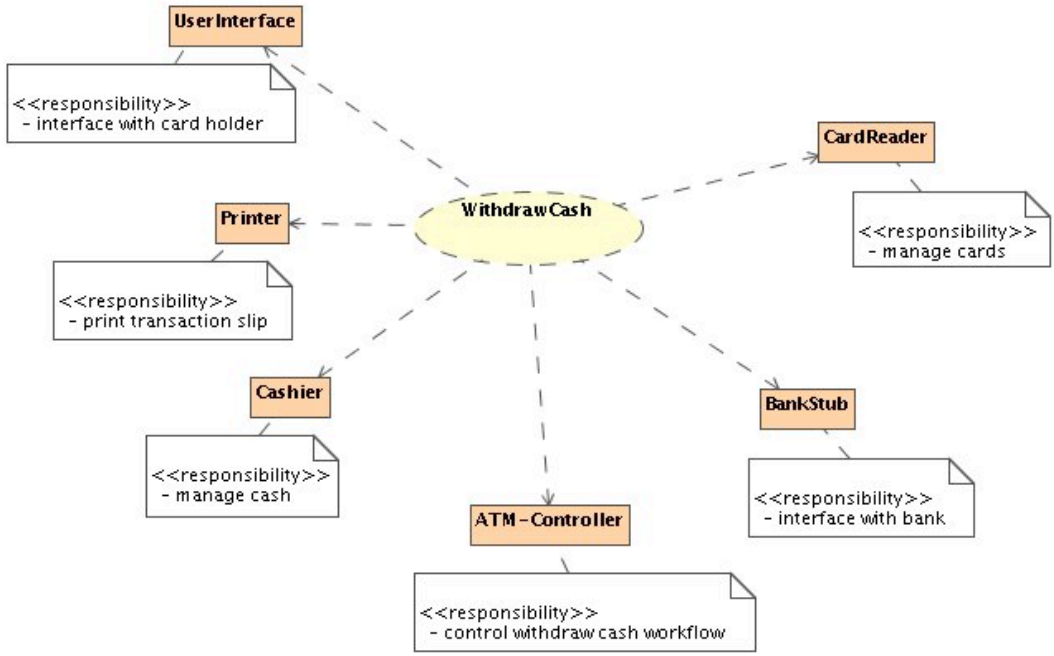
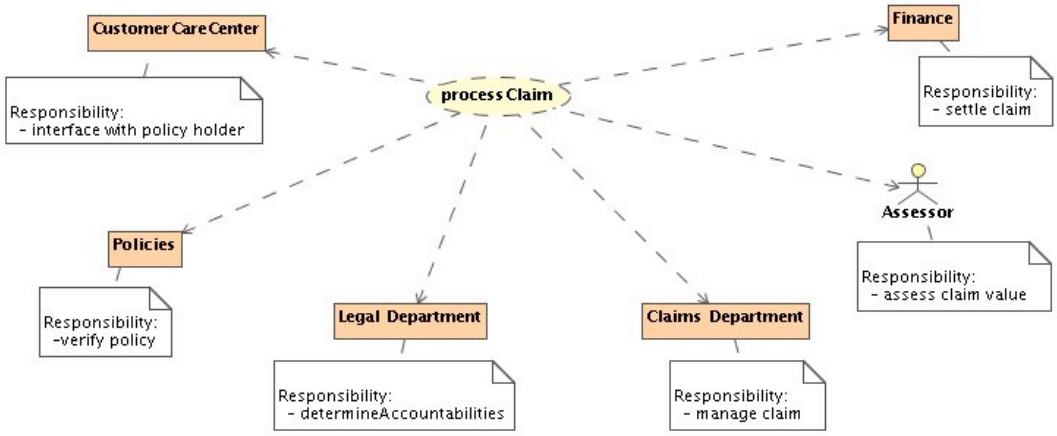


Figure 3.3, " Allocating the responsibilities which need to be addressed for the withdraw-cash use-case to core system components. " shows the core components we have identified for the auto teller, each of which acquires one of the core responsibilities which need to be addressed to realize the *withdraw-cash* use case.

Figure 3.4. Allocating the responsibilities around processing an insurance claim to core business units.



Similarly, Figure 3.4, " Allocating the responsibilities around processing an insurance claim to core business units. " shows the business units of an insurance company which acquire the core responsibilities which need to be addressed when processing an insurance claim.

4. Exercises

1. Reconsider the use case you focused on in the previous chapter. Identify the core responsibilities which need to be addressed for that use case and host each responsibility in a separate object/component.

Chapter 4. The Static View

1. Objects and classes

Objects and classes are naturally the cornerstone of object-oriented modeling. They provide the structure for the system we are modeling and the processes (e.g. business or software processes).

1.1. What is an object?

An object is a conceptual or physical entity with identity. An object usually has attributes and can perform certain operations, i.e. supply certain services. The operations (services) are requested by sending an appropriate message to the object.

For example, you have an identity (for the state your identity is captured by the identity number), certain attributes (a name, eye color, ...) and can perform certain operations (e.g. “make a cup of tea”). You are thus an object. To request a service from you one sends a message (verbal, sign language, e-mail, ...) to you.

1.2. What is a class?

Furthermore, you belong to a certain class of objects known as `homo sapiens` which itself belongs to the class of mammals and so on.

So, what is a class? A class provides a classification. You together with the other creatures which walk on two feet, wear clothes and communicate via a complex set of acoustic sound clips are classified as instances of the same class. *Objects are thus instances of classes*. The class

- provides a classification or grouping and
- encapsulates commonalities across instances of a class including
  - common attributes (we have one head, two eyes, ...)
  - and common operations (we generally walk on two feet and communicate via a mode called talking).

1.3. Identifying objects

The first step when developing a static model is to identify the objects within the system. One way to do this is to take a linguistic description of the system and to extract all the nouns. The nouns will map onto objects which are either external to the system (most probably interfacing with the system -- actors) or part of the system, i.e. system components.

Consider the following description:

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

Most of these nouns can be directly mapped onto the objects or classes of the system. If a noun maps onto an object which is not part of the system you are modeling, but interfaces with it, then the object is an actor. Identifying the nouns yields the following list of objects:

Table 4.1. The objects identified from the nouns:

Sam	Jill	client	account	ausDollar	zaRand
transaction	transactionDate	statement	period	startDate	endDate

1.4. Generalization of objects to classes

An object is very concrete. It has identity and persistent state and, at least in principle, one can identify the same object at a later stage and send further messages to it.

Usually one would like to work at a slightly more general level where the statements one makes is not only applicable to a particular object, but to any instance of that class of objects.

Abstracting the objects identified in Section 1.3, “Identifying objects” to classes we may come up with the following list of classes:

Table 4.2. The objects generalized to classes

Client	Account	Currency	Transaction	Statement
Period	Date			

1.5. Simple object and class diagrams

We have already discussed class diagrams in Section 2, “Objects and StereoTypes in use case diagrams”. To recap, a class is shown in UML as a rectangle with the name of the class written into the top centre of the rectangle. In Figure 4.1, “UML class and object diagrams” we have two class diagram, one for the class `Client` and one for the class `Account`.

Figure 4.1. UML class and object diagrams

The UML diagram in Figure 4.1, “UML class and object diagrams” also shows 3 notations for an object diagram.

- The first object diagram for the client, `sam`, has the object name followed by a colon and the class name. It specifies that the object `sam` is an instance of the class `Client`.
- The second object diagram specifies that there is an object, `jill`, without specifying the class for that object. The colon specifies that `jill` is an object of some or other class.
- In the third object diagram we specify an instance of the class `Client` without specifying a name for the object.

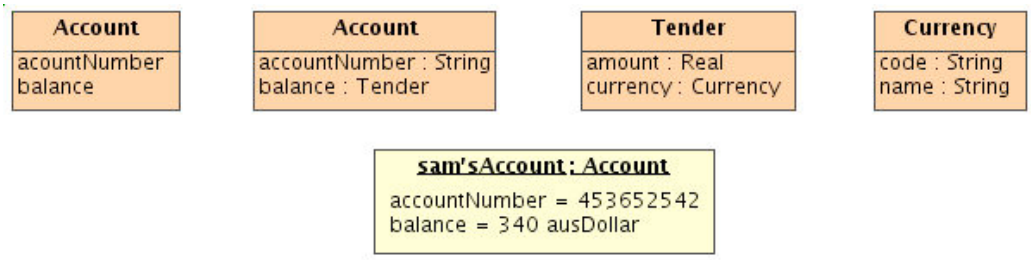
1.6. Attributes

Most objects have further features or further components. These can be specified as attributes in UML.

1.6.1. Specifying attributes for a class

An object can have attributes. For example, our account would most probably have a balance and an account number. Attributes are specified in UML in a second compartment below the compartment containing the class name and potentially the stereotype for the class:

Figure 4.2. Attributes are shown in a second compartment of the class diagram.



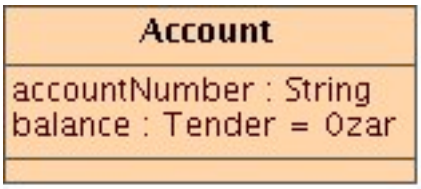
An attribute is conceptually an object which is a component of another object. For example, the balance itself is an object which is a component of the account. This object is an instance of the `Tender` class which encapsulates an amount in a currency. The tender concept is itself explained in another class diagram.

The first class diagram shows the attributes without specifying their class (their type). The other class diagrams show the attribute types. We also show an object diagram for which we specify the values of its attributes.

1.6.2. Default values

UML also supports the specification of default values. This is done via an assignment as show in Figure 4.3, “Default values for attributes are specified via an assignment.”

Figure 4.3. Default values for attributes are specified via an assignment.



1.6.3. Constraints

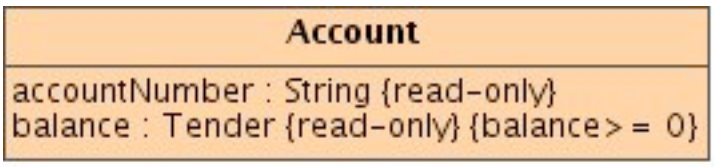
A constraint is specified in UML via curly brackets. In the context of class attributes, one can use constraints to

- restrict the domain of the attribute values, i.e. to restrict the values an attribute can acquire or to

- constrain access to the attributes.

For example, in Figure 4.4, " Attribute values and access can be constrained by placing the constrain in curly brackets behind the attribute. " we add a `{read-only}` constraint to both, the `balance` and the `accountNumber` attributes, preventing anybody to set the value directly to anything else (though one will still be able to change the balance by crediting or debiting the account as we shall see in Section 1.7, "Services").

Figure 4.4. Attribute values and access can be constrained by placing the constrain in curly brackets behind the attribute.



We have also constrained the value of balance to be non-negative via a `{balance >= 0}` constraint.

1.6.4. Multiplicities in UML

Multiplicities are can be used in UML on attributes as well as on association-based relationships (association, aggregation or composition). One can specify the multiplicity as a single number, a range, or a choice of multiplicities. The notation used in UML is summarized in Table 4.3, "Multiplicites"

Table 4.3. Multiplicites

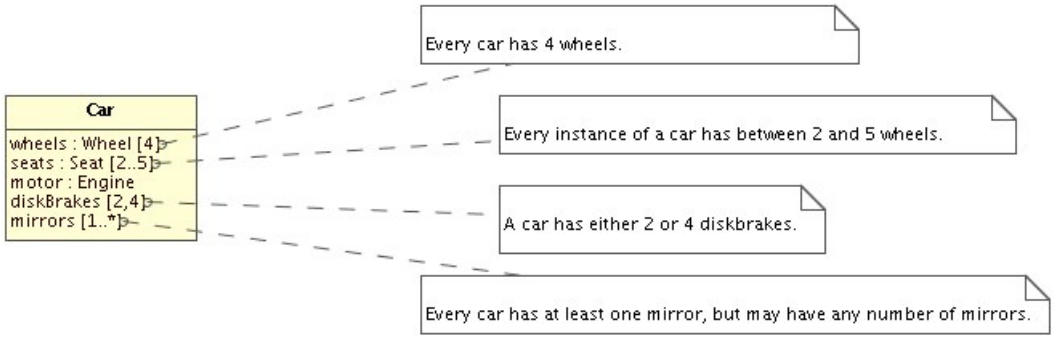
*	zero or more
1..*	one or more
n	n multiples
m,n	m or n
m..n	between m and n (inclusive)
k,l,m..n	either k,lor between m and n (inclusive)

1.6.5. Collection attributes

In UML one can specify a multiplicity constraint for attributes. This is useful when one wants to specify that instances of one class have multiple instances of another class as attributes. Multiplicities are specified in square brackets behind the attribute.

For example, Figure 4.5, "Collection attributes and derived attributes" specifiies that a car has 4 wheels, between 2 and 5 seats, one motor which is an instance of an engine, either 2 or 4 discbrakes and one or more mirrors.

Figure 4.5. Collection attributes and derived attributes

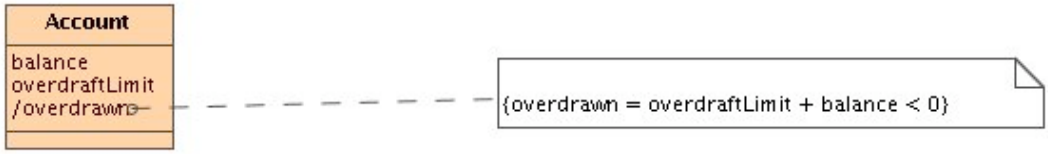


1.6.6. Derived attributes

At times attributes are not independent of one another and one attribute value can be devived from other attribute values.

For example, whether an account is overdrawn or not could be derived from the current balance and the overdraft limit of the account.

Figure 4.6.



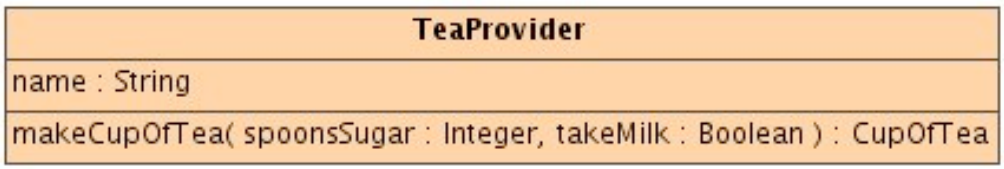
1.7. Services

So far we have looked at attributes of objects and classes. Objects may also perform operations or offer services. If we specify a service for a class it implies that all instances (objects) of the class will provide that service.

In UML one specifies the services offered by instances of a class in a third operations compartment. A service is specified by a service interface specification containing

- **A service name.** The name is part of the signature which identifies the service. The service name is often chosen as a verb which clearly describes what you want the instance of the class do for you.
- **Service arguments.** The parameters provided to a service specify any information the client sends with the service request. For example, if you request me to make you a cup of tea, you will have to provide as additional information with your service request the number of spoons of sugar you would like in your tea and whether you take milk or not. The parameters themselves are objects and hence instances of classes.
- **The return value of a service (if any).** A service might have a return value. For example, if I provide you the service of making you a cup of tea, I will return you a cup of tea (see Figure 4.7, “ A service is requested providing certain parameters and the service may provide a return value. ”

Figure 4.7. A service is requested providing certain parameters and the service may provide a return value.



1.8. OO (Camel) naming convention

We have followed the standard object-oriented naming convention, the *Camel convention*. It is standard for UML modeling as well as for many object-oriented programming languages like Java, Smalltalk, non-Microsoft C++ developers and so on. (many Microsoft technologies use the Hungarian notation which includes data type information in variable names -- it is a lot more complex than the Camel convention and, in some ways, violates the spirit of object-orientation). The Camel convention is also starting to entrench itself in XML data modeling.

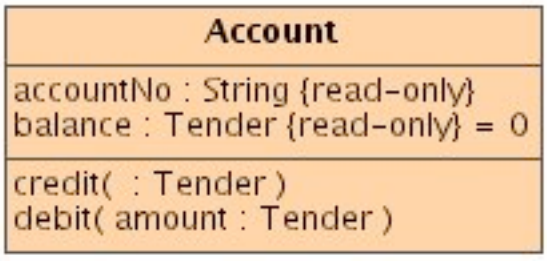
The rules for the Camel convention are simple:

1. Class names start with capital letters.
2. Everything else including object and method (service) names start with a lower case letter.
3. Word boundaries are capitalized in all cases.

Hence the class `TeaProvider` starts with a capital letter, but an instance of that class, say `fritz`, starts with a lower case letter. So does the service name, `makeCupfTea`, while the classes `CupOfTea`, `Integer` and `Boolean` all start with capital letter.

Lets, as a second example, revisit our `Account` class. One can credit an account and debit it. In either case one should provide the tender (amount in a currency) one is crediting the account by.

Figure 4.8. A service need not provide a return value.



A service need not have a return value

Note that neither the `credit` nor the `debit` service provide a return value. They both simply perform an operation.

We need not give a parameter a name.

In the specification of the `debit service` in Figure 4.8, " A service need not provide a return value. " we did not give the parameter a name. We simply state that the account is debited by an instance of a `Tender`.

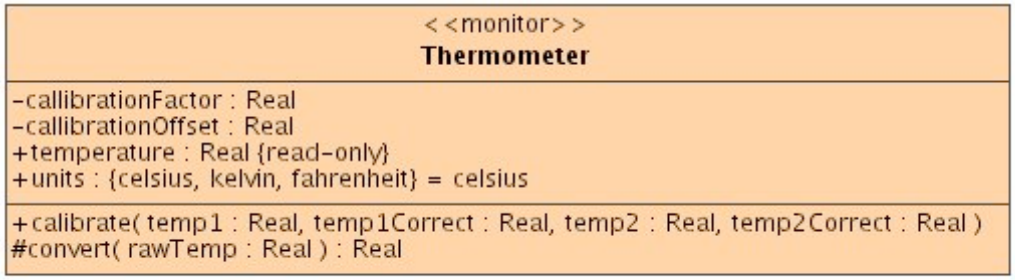
1.9. Access control

Access control is more of an implementation issue and not really relevant for business analysts. We merely mention it here for completeness sake.

UML defines three access levels:

- *public (+):* a public element (attribute or service) of a class can be accessed from anywhere.
- *private (-):* a private element can be accessed only from within the class itself, i.e. that element can be accessed only from within instances of that class.
- *package (~):* Elements denoted with package scope are accessible from within any element defined within the same package.
- *protected (#):* implies that the element can be accessed from within instances of that class an instances of any subclass, but not from within an object which is not an instance of the class or one of its sub-classes.

Figure 4.9. UML uses + for public access, - for private access and # for protected access.



In Figure 4.9, " UML uses + for public access, - for private access and # for protected access. " we have a thermometer with 4 attributes. Two of them are public while the other two are private. The values which the attributes `units` van acquire is constrained to degrees `celsius`, `kelvin` or degrees `fahrenheit`.

We have applied a `{read-only}` constrain to the temperature specifying that users can query, but not to set the temperature of a thermometer. Users can, however, `calibrate` thermometers, but access to the internal `convert` service which converts between the raw voltage reading and a temperature is restricted to within instances of that class and instances of its subclasses, i.e. the method has been declared with access level `protected`.

Note also that we assigned the `monitor` stereotype to the thermometer, conceptually classifying it as an object which can be used to monitor other objects.

As a business analyst one usually one shows public members.

1.10. Encapsulation

Encapsulation is about hiding implementation details, i.e. the way in which attributes and services

are realized. A service provider could make use of different algorithms, different internal data representations and even delegate some the responsibilities down to lower level service providers.

All this should be transparent to clients of the service provider, i.e. should be hidden. In UML one uses access control via `private` and `protected` members to enforce encapsulation.

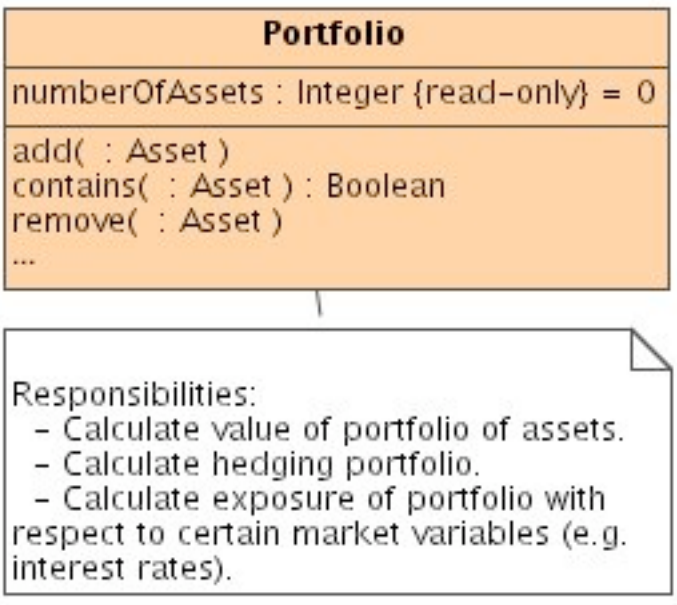
1.11. Incomplete member list

In UML it is understood that every view is at least potentially an incomplete view, i.e. that other views can show other aspects of the elements shown in one view.

For example, one class diagram can show the attributes and services of a class related to a particular responsibility of the class. Another view could show other attributes and other services of the class, perhaps related to other responsibilities of that class.

In the case where you want to emphasize that there are indeed more attributes and services which are not shown in a particular view, you can use 3 dots to explicitly point out that there are more members which are not show in this view (see the incomplete services list of the `Portfolio` shown in Figure 4.10, " Three dots can be used to show an incomplete member list. Responsibilities are specified via a responsibilities comment. ".

Figure 4.10. Three dots can be used to show an incomplete member list. Responsibilities are specified via a responsibilities comment.



1.12. Assigning responsibilities to classes

For the process we use for both, business modeling and system design, the concept of assigning responsibilities to classes will be central. In UML this is done via a responsibilities comment. These responsibilities will be ultimately realised by services offered by the class.

In Figure 4.10, " Three dots can be used to show an incomplete member list. Responsibilities are specified via a responsibilities comment. " we assign a number of responsibilities to the `Portfolio` class which have, at this stage, not yet been realized in terms of services.

2. Implementation mappings for object and class diagrams

Here we show mappings of object and class diagrams onto Java, C++ and XML.

2.1. Mapping class and object diagrams onto *Java*

Since Java is an object-oriented programming language, the mapping of UML class and object diagrams onto Java code is very simple.

2.1.1. Mapping objects and classes

Consider the account class and instance shown in Figure 4.1, “UML class and object diagrams”. The *Java* implementation of an empty class diagram is simply

```
public class Account {}
```

Note, however, that the class is not as empty as it looks. A default constructor and a `this` reference are automatically generated. Note though that a default constructor is only generated automatically if no other constructor is defined. Furthermore, *Java* implements a single-tree inheritance model where every class is ultimately derived from the mother of all *Java* classes, `Object`. Any class will thus inherit all the services from `Object` like a default `toString()` method and many others.

In order to implement an object diagram, one has to instantiate a class:

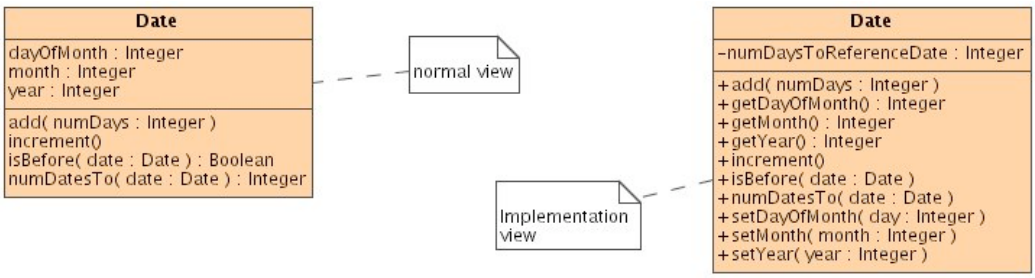
```
Account acc1 = new Account();
```

Here we define a reference, `acc1`, which can refer to any instance of the `Account` class. The `new` operator creates a `Bond` object and returns a reference to that object. The remainder of the statement is simply reference assignment.

2.1.2. Mapping UML attributes onto Java

A public attribute represents an attribute of the class as seen by users of the class. They should not map onto data fields -- implementors may or may not choose to use corresponding data fields. Instead,public attributes are mapped onto getters and setters.

Figure 4.11. Aspects of a date class



Consider, for example the `Date` class shown in the left diagram of Figure 4.11, “Aspects of a date class”. From a user's perspective a date has a day, month and year. From an implementation perspective you may decide that using the earth's rotation around its own axis, that around the sun and

the moon as reference systems results in a convoluted logic which manifests itself in, for example, a very complex leap year calculation:

> *Every 4'th year is a leap year except every century which isn't except every fourth century which is.*

In the day/month/year basis the implementation of the services offered by dates (e.g. the `increment` service) are excessively complex and a loop like

```
for (Date d=d1; d.before(d2); d.increment())
  ...
```

has considerable computational overheads. So you decide to select your favourite date (e.g. your birth date, date on which democracy was introduced, ...) as reference date and represent any other date as the number of days between it and the reference date.

Now most of the services are straight-forward. Incrementing a date is now simply incrementing an integer. To calculate the number of days between two dates involves merely integer subtraction. Only when users query the actual day, month or year,will we have to do the transformation between our internal representation for that date and the day/month/year format. The implementation view of a date is shown in the right diagram of Figure 4.11, “Aspects of a date class” and the mapping of the attributes onto Java code would be

```
public class Date
{
  public int getDay() {...}
  public int getMonth() {...}
  public int getYear() {...}

  public void setDay(int day) {...}
  public void setMonth(int month) {...}
  public void setYear(int year) {...}
  ...
}
```

2.1.3. Mapping UML operations onto Java methods

The class diagrams only show the operations (services) offered by the class, not how these will be realized. As such they map onto method headers. The implementation for these services will be obtained by mapping sequence, activity and state diagrams onto Java.

The services shown in Figure 4.11, “Aspects of a date class” thus map onto

```
public class Date
{
  public void add(int numDays) {...}
  public void increment() {...}
  public Boolean isBefore(Date date) {...}
  public int numDatesTo(Date date) {...}
  ...
}
```

2.1.4. Mapping UML access levels onto Java

The mapping is direct with `public`, `protected` and `private` access levels directly supported in Java. However, take note that while `protected` in UML gives access to the sub-classes only, `protected` in Java also gives access to any other class in the same package.

Furthermore, Java defines an additional access level, *friendly* or *package* which gives access to all other classes in the same package. This access level is requested in Java by completely omitting an access level keyword -- it is the default access level.

2.2. Mapping class and object diagrams onto *C++*

The C++ mapping of object and class diagrams is, bar some minor technical issues the same as the Java mapping.

2.2.1. Mapping objects and classes

The class diagram maps onto

```
class Account {};
```

with C++ supporting only `public` classes and the object diagram onto either

```
Account* acc1 = new Account();
```

or

```
Account acc1;
```

where the former creates the object on the heap while the second creates it on the stack.

Note, however, that C++ does not enforce a common superclass across all classes. The compiler will, however, still write a copy constructor, assignment operator, (both of which make, by default, a byte-for-byte copy of the data fields which may or may not be correct), a default constructor if no other constructor is defined and a data field, `this` representing a pointer to the object itself. The empty class diagram would thus, by default result in a not-so-empty class

```
class Account
{
  public:
    Account();
    Account(const Account& acc);

    Account& operator=(const Account& acc);

  private:
    Account* this;
};
```

If one does not want some of these feature,one has to explicitly disable them by declaing them `private`.

2.2.2. Mapping UML attributes onto C++

As with *Java*, public UML attributes should be interpreted as attributes from the user's perspective and hence the mapping is onto query and set methods. By convention, a query method has the name of the attribute and no arguments, while a set method has the same name, but takes the new attribute value as argument:

```
class Date
{
```

```
  public:
    int getDay() const;
    int getMonth() const;
    int getYear() const;

    void setDay(int day);
    void setMonth(int day);
    void setYear(int day);
};
```

The trailing `const` in the method headers specifies impotent services,i.e. services which do not change the state of that object which supplies the service.

2.2.3. Mapping UML operations onto C++ methods

This mapping is essentially the same as the *Java* mapping:

```
class Date
{
  public:
    void add(int numDays);
    void increment();
    Boolean isBefore(const Date& date) const;
    int numDatesTo(const Date& date) const;
  ...
};
```

2.2.4. Mapping UML access levels onto Java

The UML access levels have really been derived from the C++ access levels and hence `public`, `protected` and `private` access levels map directly onto its C++ equivalents.

2.3. Mapping class and object diagrams onto *XML*

With the introduction of XML schemas,XML can be seen as a full object-oriented data specification and communication protocol.

2.3.1. Mapping objects and classes

Consider the account class and instance shown in Figure 4.1, “UML class and object diagrams”. The *XML* implementation of an empty class diagram is simply

```
<xsd:complextype name="Account"/>
```

where `xsd` is the namespace prefix chosen for the XML schema vocabulary.

In order to implement an object diagram, one has to instantiate a class:

```
<xsd:element name="acc1" type="Account/>
```

2.3.2. Mapping UML attributes onto XML

The Account class has two attributes, `accountNumber` and `balance`. These are mapped onto XML child elements

```
<xsd:schema xmlns:xsd='http://www.w3.org/2001/XMLSchema'>

  <xsd:complexType name="Account">
    <xsd:sequence>
      <xsd:element name="accountNumber" type="xsd:string"/>
      <xsd:element name="balance" type="xsd:decimal"/>
    </xsd:sequence>
  </xsd:complexType>

</xsd:schema>
```

2.3.3. What about the operations?

XML is purely for information, not for functionality and hence there is no mapping for UML operations onto XML.

3. Specialization through sub-classing

One of the most important benefits of object orientation is the ability to work at various levels of abstraction. This is partially facilitated by the ability to define subclasses (specializations) for a class.

3.1. Specialization as an *is a* relationship

Instances of the specialized class (subclass) are special types of instances of the more general class (the superclass). For example, a `ChequeAccount` *is a* special type of `Account`. If anybody requires an `Account`, we can give them a `ChequeAccount` instead.

Test for substitutability

Substitutability is the central aspect of specialization and you should always check whether you can substitute an instance of the subclass very time somebody requests an instance of the superclass. This test should *never* fail.

3.1.1. Abstract references

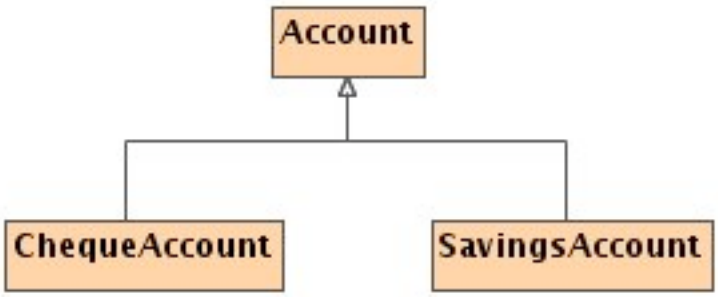
Similarly, you might have a reference to an `Account`. That reference will be able to refer to any object which *is an* `Account`.Such a reference can thus not only refer to instances of `Account` but also to any instance of any subclass of `Account`, e.g. to a `ChequeAccount`.

3.2. Documenting a Specialization relationship in UML

Specialization is shown as a triangular arrow pointing from the specialized entity to the more general entity.

In the case of subclassing the triangular arrow will point from the *subclass* (the more specialized class) to the *superclass* (the more general and more abstract class).

Figure 4.12. Subclassing is shown in UML via a triangular arrow pointing from the of the subclass to the superclass.

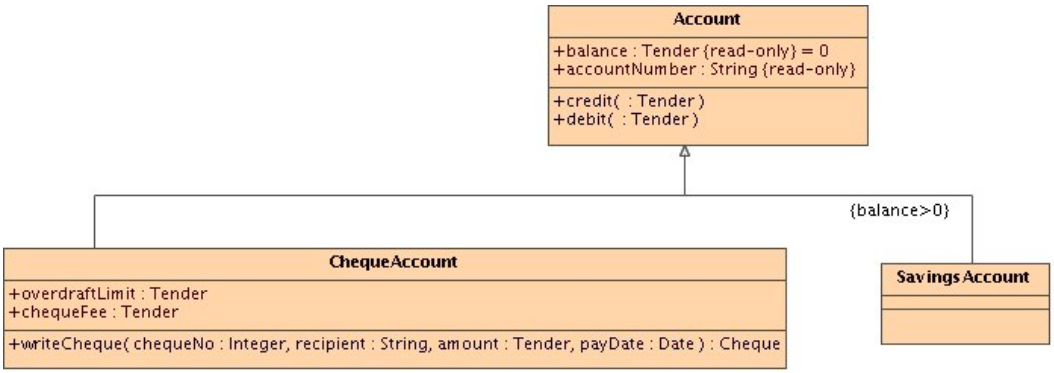


3.3. Inheritance as a by-product of sub-classing

During subclassing the instance members of the superclass are inherited by instances of the subclass. Instances of the subclass will thus have all the attributes which instances of the superclass have and will supply all the services which instances of the superclass do. They may add further attributes and provide additional services, or, as wel shall see later, provide their own implementations of the corresponding services of the superclass.

Reconsider our example of the cheque and savings accounts. They *are* both accounts and hence we can define them as subclasses of account. Cheque and savings accounts will thus inherit the `accountNumber` and `balance` attributes as well as the `credit` and `debit` services from `Account`.

Figure 4.13. A subclass inherits all instance members (attributes and operations) of the superclass.



3.3.1. Don't subclass for inheritance sake only

Subclassing is often called *inheritance*. It is, however, important to see the relationship as a *specialization* relationship with *inheritance* being simply a by-product of *specialization*, i.e. subclassing should never be done for inheritance sake only.

For example, you may feel that you would want to be able to query a student for the particulars of the university he or she is studying with and may decide (hopefully not) that instances of the student class could inherit this from university. However, a student *is not* a university! Otherwise you could a faculty to a student -- the main problem being that you don't have all your's if you design such a structure.

3.4. Implementing specialization in Java

A specialization relationship between two classes maps onto an `extends` relationship in Java:

3.4.1. Account.java

```
public class Account
{
  public Account(String accountNumber)
  {
    this.accountNumber = accountNumber;
  }

  public double getBalance() {return balance;}

  public void credit(double amount) {balance += amount;}

  public void debit(double amount) throws Exception
  {
    balance -= amount;
  }

  public String toString()
  {
    return accountNumber + ": bal=" + amountFormatter.format(balance);
  }

  private double balance = 0;
  private String accountNumber;

  public static final java.text.DecimalFormat amountFormatter
    = new java.text.DecimalFormat("########0.00");
}
```

3.4.2. ChequeAccount.java

```
public class ChequeAccount extends Account
{
  public ChequeAccount(String accountNumber, double chequeFee)
  {
    super(accountNumber);
    this.chequeFee = chequeFee;
  }

  public void setChequeFee(double newChequeFee)
  {
    this.chequeFee = chequeFee;
  }

  public void setMinimumBalance(double minimumBalance)
  {
    this.minimumBalance = minimumBalance;
  }

  public void debit(double amount) throws Exception
  {
    double totalAmount = amount + chequeFee;
    if (getBalance() - totalAmount < minimumBalance)
      throw new InsufficientFundsException();

    super.debit(totalAmount);
  }

  public double getChequeFee() {return chequeFee;}
  public double getminimumBalance() {return minimumBalance;}

  public String toString()
  {
    return super.toString()
```

```
      + " (chequeFee=" + amountFormatter.format(chequeFee) + ", "
      + "overdraft limit=" + amountFormatter.format(minimumBalance) + ")";
  }

  private double chequeFee, minimumBalance = 0;
}
```

3.4.3. InheritanceTest.java

Instances of `ChequeAccount` will inherit the `credit`, `debit`, `getBalance` and `getAccountNumber` services:

```
public class InheritanceTest
{
  public void run()
  {
    ChequeAccount acc = new ChequeAccount("876532762", 5);
    acc.credit(300);
    try
    {
      acc.debit(150);
    }
    catch (InsufficientFundsException e)
    {
      System.out.println("You're broke, bloke");
    }
    catch (Exception e) {System.out.println(e);}

    System.out.println(acc);
  }

  public static void main(String[] args)
  {
    new InheritanceTest().run();
  }
}
```

Running the application provides the following output:

```
876532762: bal=145.00 (chequeFee=5.00, overdraft limit=0.00)
```

3.5. Implementing specialization in C++

C++ supports the concepts of `public`, `private` and `protected` and each of these may be combined with `virtual` or non-virtual specialization.

Virtual specialization will be discussed in the context of multiple inheritance. For now, accept that the only for of specialization which is consistent with the substitutability criterion for specialization is `virtual` specialization.

3.5.1. Public versus protected and private specialization

With *public specialization*, the access levels of the inherited members remain unchanged, i.e. `public` members of the superclass remain `public` members of the subclass, `protected` members of the superclass remain `protected` members of the subclass and `private` members of the superclass remain `private` members of the subclass,

With *protected specialization* specialization `public` members are demoted to access level `protected` while the others are left unchanged.

Finally, *private specialization* reduces the access levels of all inherited members to `private`.

3.5.2. Why use only public specialization?

Recall that substitutability requirement requires that one should always be able to substitute an instance of a subclass for an instance of a superclass. If a public member is demoted to protected or private access level that member can no longer be used as`a substitution for subclass instances and this violating the basic requirement for specialization.

3.5.3. Specialization in C++

Specialization is requested in C++ by appending a colon to the class header followed by the access modifier and possibly the keyword `virtual` whose sigificance is discussed when we come to multiple inheritance. The only specialization relationship which is consistent with the basic premises of object-orientation is `public virtual` specialization:

3.5.3.1. Account.h

```
#ifndef ACCOUNT_H
#define ACCOUNT_H

#include <iostream>

#include "Exception.h"

using namespace std;

class Account
{
  public:
    Account(const char* accountNumber);

    virtual ~Account();

    virtual void credit(double amount);
    virtual void debit(double amount);

    double balance() const;

    const char* accountNumber() const;

    char* toString() const;

  private:
    double _balance;
    const char* _accountNumber;
};

ostream& operator<< (ostream& os, const Account& account);

#endif
```

3.5.3.2. Account.cpp

```
#include "Account.h"
#include <iostream>

using namespace std;

Account::Account(const char* accountNumber)
  : _balance(0), _accountNumber(accountNumber) {}

Account::~Account() {}
```

```
void Account::credit(double amount) {_balance += amount;}

void Account::debit(double amount) {_balance -= amount;}

double Account::balance() const {return _balance;}

const char* Account::accountNumber() const {return _accountNumber;}

ostream& operator<< (ostream& os, const Account& account)
{
  os << account.accountNumber() << ": balance=" << account.balance();
  return os;
}
```

3.5.3.3. ChequeAccount.h

```
#ifndef CHEQUEACCOUNT_H
#define CHEQUEACCOUNT_H

#include <iostream>

#include "Account.h"

using namespace std;

class ChequeAccount: public virtual Account
{
  public:
    ChequeAccount(const char* accountNumber, double chequeFee);

    virtual void debit(double amount);

    double chequeFee() const;
    void chequeFee(double chequeFee);

    double minimumBalance() const;
    void minimumBalance(double minimumBalance);

    char* toString() const;

  private:
    double _chequeFee, _minimumBalance;
};

ostream& operator<< (ostream& os, const ChequeAccount& account);

#endif
```

3.5.3.4. ChequeAccount.cpp

```
#include "ChequeAccount.h"
#include <iostream>

using namespace std;

ChequeAccount::ChequeAccount(const char* accountNumber, double chequeFee)
  : Account(accountNumber), _chequeFee(chequeFee) {}

void ChequeAccount::debit(double amount)
{
  double totalAmount = amount + _chequeFee;
  if (balance() - totalAmount < _minimumBalance)
    throw InsufficientFundsException("You're broke,bloke.");
```

```
  Account::debit(totalAmount);
}

double ChequeAccount::chequeFee() const {return _chequeFee;}
void ChequeAccount::chequeFee(double chequeFee)
{
  _chequeFee = chequeFee;
}

double ChequeAccount::minimumBalance() const {return _minimumBalance;}
void ChequeAccount::minimumBalance(double minimumBalance)
{
  _minimumBalance = minimumBalance;
}

ostream& operator<< (ostream& os, const ChequeAccount& account)
{
  os << account.accountNumber() << ": balance=" << account.balance()
     << " (chequeFee=" << account.chequeFee() << ", "
     << " minimumBalance=" << account.minimumBalance() << ")";
  return os;
}
```

3.5.3.5. InheritanceTest.cpp

```
#include <iostream>

#include "Account.h"
#include "ChequeAccount.h"
#include "Exception.h"

using namespace std;

int main()
{
  ChequeAccount* acc = new ChequeAccount("876532762", 5);
  acc->credit(300);
  try
  {
    acc->debit(150);
  }
  catch (InsufficientFundsException e)
  {
    cout << "You're broke, bloke" << endl;
  }

  cout << (*acc) << endl;

  delete acc;

  return 0;
}
```

3.6. Implementing specialization in XML

A specialization relationship between two classes maps onto extensive and/or restrictive specialization. The XML schema file specifying these classes is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd='http://www.w3.org/2001/XMLSchema'>
  <xsd:complexType name="Account">
    <xsd:sequence>
      <xsd:element name="accountNumber" type="xsd:string"/>
      <xsd:element name="balance" type="xsd:decimal"/>
```

```
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="ChequeAccount">
    <xsd:complexContent>
      <xsd:extension base="Account">
        <xsd:sequence>
          <xsd:element name="chequeFee" type="xsd:decimal"/>
          <xsd:element name="minimumBalance" type="xsd:decimal"/>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <xsd:element name="accounts">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="account" type="Account" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Here the type `ChequeAccount` inherits `accountNumber` and `balance` from the `Account` type.

3.7. Polymorphism

Polymorphism is one of the central concepts and key benefits of object-orientation. It provides the mechanism through which one can work at higher levels of abstraction.

Polymorphism is, however, not foreign to us in everyday life. We are quite comfortable with the fact that one service provider will realize a service differently from another. Frank Sinatra even wrote a song on polymorphism:

> " *I do it my way.* "

And this is really all there is conceptually to polymorphism. In object-orientation you are not calling a particular function, you are requesting a service from an object and each object will decide itself which functions are called to realize the service.

3.7.1. Polymorphism in UML

UML does not provide a separate notation for polymorphism. Like in any clean object-oriented framework it is assumed throughout. For example, if we revisit the UML diagram shown in Figure 4.13, " A subclass inherits all instance members (attributes and operations) of the superclass. ", then it is understood in UML that `ChequeAccount` and `SavingsAccount` can have their own implementations of the `credit` and `debit` services. For example, cheque accounts could raise a cheque fee on debit transactions and savings accounts could provide incentives for crediting.

3.8. Polymorphism in Java

In both, UML and Java, polymorphism is assumed throughout, i.e. all method calls are resolved polymorphically. This is illustrated in the following example program listing:

3.8.1. PolymorphismTest.java

```
public class PolymorphismTest
{
  public void run()
  {
```

```
    Account[] accounts = new Account[3];
    accounts[0] = new Account("376632865");
    accounts[1] = new ChequeAccount("876532762", 5);
    accounts[2] = new Account("654324532");

    for (int i=0; i<accounts.length; ++i)
    {
      accounts[i].credit(300);
      System.out.println(accounts[i]); /* resolved polymorphically */
    }

    for (int i=0; i<accounts.length; ++i)
    {
      try
      {
        raiseSubscriptionFees(accounts[i]);
      }
      catch (InsufficientFundsException e)
      {
        System.out.println("You're broke, bloke");
      }
      catch (Exception e) {System.out.println(e);}
    }

    System.out.println("After raising subscription fees:");
    for (int i=0; i<accounts.length; ++i)
      System.out.println(accounts[i]);
  }

  public void raiseSubscriptionFees(Account account)
                throws Exception
  {
    account.debit(100); /* resolved polymorphically */
  }

  public static void main(String[] args)
  {
    new PolymorphismTest().run();
  }
}
```

Running the application yields the following output:

```
376632865: bal=300.00
876532762: bal=300.00 (chequeFee=5.00, overdraft limit=0.00)
654324532: bal=300.00
After raising subscription fees:
376632865: bal=200.00
876532762: bal=195.00 (chequeFee=5.00, overdraft limit=0.00)
654324532: bal=200.00
```

3.9. Polymorphism in C++

A technical requirement for polymorphism is *run-time linking* or *dynamic binding*, i.e. the actual code called when sending a message to an object (the concrete realization of a service) may not be known at compile time since a concrete service provider may only be selected at run-time.

Unlike for Java, dynamic binding is neither automatic nor the default for C++. It must be explicitly requested via the method qualifier, `virtual`. Methods which have not been declared `virtual` will thus not be resolved polymophically. Hence, the only methods which have not been declared virtual in Section 3.5.3.1, “Account.h” are those which are not meant to be overridden and hence not meant to be resolved polymorphically.

In the following example program the `debit` method is resolved polymorphically:

```
#include <iostream>

#include "Account.h"
#include "ChequeAccount.h"
#include "Exception.h"

using namespace std;

void raiseSubscriptionFees(Account* account)
{
  account->debit(100); /* resolved polymorphically */
}

int main()
{
  const int numAccounts = 3;
  Account** accounts = new Account*[numAccounts];

  accounts[0] = new Account("376632865");
  accounts[1] = new ChequeAccount("876532762", 5);
  accounts[2] = new Account("654324532");

  for (int i=0; i<numAccounts; ++i)
  {
    accounts[i]->credit(300);
    cout << *accounts[i] << endl;
  }

  for (int i=0; i<numAccounts; ++i)
  {
    try
    {
      raiseSubscriptionFees(accounts[i]);
    }
    catch (InsufficientFundsException e)
    {
      cout << "You're broke, bloke" << endl;
    }
  }

  cout << "After raising subscription fees:" << endl;
  for (int i=0; i<numAccounts; ++i)
    cout << *accounts[i] << endl;
}
```

3.10. Polymorphism and substitutability in XML

Being an object-oriented data specification language, XML supports both extensive as well as restrictive specialization as well as polymorphism and hence substitutability.

Recall the XML schema discussed in Section 3.6, “Implementing specialization in XML”. Here we specified that cheque accounts are accounts with a minimum balance and a cheque fee. We also specified that XML instance documents which realize the schema contain a collection of accounts. We can now, directly, have XML instance documents containing a polymorphic collection of accounts, i.e. we can substitute a cheque account for an account:

```
<?xml version="1.0" encoding="UTF-8"?>
<accounts xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="accounts.xsd">

  <account>
    <accountNumber>8763287632</accountNumber>
    <balance>500.00</balance>
  </account>
```

```
  <account xsi:type="ChequeAccount">
    <accountNumber>7832876</accountNumber>
    <balance>2334.34</balance>
    <chequeFee>5.50</chequeFee>
    <minimumBalance>-2000</minimumBalance>
  </account>

  <account>
    <accountNumber>8763287632</accountNumber>
    <balance>500.00</balance>
  </account>

</accounts>
```

3.11. Abstract classes

Going to higher levels of abstraction we often come to a point where we want to introduce a concept for which there is no direct concrete realization. In such cases we would use an abstract class.

3.11.1. What is an abstract class?

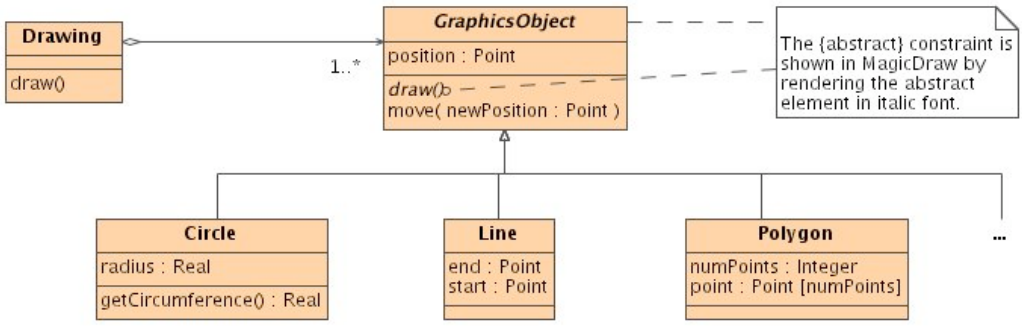
An abstract class is a class which is not instantiates, i.e. we cannot create objects of an abstract class. Declaring a class abstract is thus a constraint: we are constraining instantiation.

An abstract class will usually have concrete subclasses, i.e. subclasses which can be instantiated. The only instances of an abstract class will thus be an instances of its concrete subclasses (recall from Section 3.1, “Specialization as an is a relationship” any instance of a subclass is also an instance of its superclasses). These concrete subclasses will inherit the instance members of the abstract class.

3.11.2. Concrete versus abstract methods

We can also declare a method or service as abstract. Doing this implies that the class will not provide an implementation (a concrete realization) for the service. It is simply a statement that instances of that class must provide this service, hence the instances of its sub-classes.

Figure 4.14. GraphicsObjects is an abstract class with an abstract draw method.



Example 4.1. Graphics objects

In Figure 4.14, “ GraphicsObjects is an abstract class with an abstract draw method. ”, we define an abstract class, `GraphicsObject`. This class will never be instantiated directly, only its concrete

subclasses will.

`GraphicsObject` defines one abstract method, `draw`. The class itself does not provide an implementation (what should an abstract graphics object draw anyway?). The abstract method represents a requirements specification for the concrete subclasses, i.e. they must all implement a `draw` service. Any class which does not provide an implementation of `draw` must itself be declared `{abstract}` because it does not yet fullfill all the requirements of a graphics object.

Showing only part of a class hierarchy

It should always be assumed in UML that each view is a selective view showing only those aspects of the model which are relevant to whatever the one wants to communicate with that view. Hence, if if only some classes are shown in a class hierarchy,then one should not assume that these are all the classes in that hierarchy. If one wants to emphasise that there are more classes (in contrast to the scenario where there may or may not be more subclasses), then one can show an inheritance link from 3 dots to the superclass which definetily has more subclasses.

Requirements are enforced at instance level.

The requirement that a service must be provided (as laid down by specifying an abstract method) is enforced at instance level. A class can have sub-classes which do not provide an implementation of the abstract method. However, UML and most object-oriented programming languages, will ensure that there will be no instances (object) of type `GraphicsObject` which do not provide a `draw` service. This is enforced by requiring any subclass which does not provide an implementation of an abstract method of one of its abstract superclasses to be declared abstract itself.

3.11.3. Why abstract classes?

So let us recap. What are the benefits of introducing abstract classes if we cannot instantiate them anyway. They are

1. The abstract class may *encapsulate concrete commonalities* among its concrete subclasses, e.g. common attributes and common operations.

2. One can encapsulates *requirements specifications for its concrete subclasses* in an abstract class. In particular, one specifies which services the concrete subclasses must provide. This is done via abstract methods.

3. The abstract class can *introduce an abstract concept* which one can later work with, both at a conceptual level, but even at a concrete level in code. For example, we can build logic around our `GraphicsObject`, like specifying that a *Drawing* has a collection of graphics objects and that we can draw the drawing by sending a `draw` message to each of these graphics objects.

3.12. Implementing abstract classes in Java

Abstract classes in UML map directly on abstract classes in Java. An abstract class may or may not have abstract methods. However, if a class has an abstract method or inherits one without supplying an implementation, the class must be declared abstract.

Below we show the Java mapping of the UML diagram for the abstract `GraphicsObject` class shown in Figure 4.14, " GraphicsObjects is an abstract class with an abstract draw method. ":

```
public abstract class GraphicsObject
{
  public abstract void draw();
```

```
  public void move(Point newPosition) {...}

  private Point position;
}
```

3.13. Implementing abstract classes in C++

C++ does not have a keyword to declare a class abstract. A class is abstract either by virtue of it having abstract methods or because it inherits abstract methods without providing an implementation for them. Consequently you cannot prevent instantiation without having at least one abstract method.

A method is declared abstract in C++ by omitting an implementation and appending instead an `=0` behind the function header:

```
class GraphicsObject
{
  public:
    void draw() = 0;  /* this is the abstract method */

    void move(Point newPosition) {...}

  private:
    Point position;
};
```

3.14. Implementing abstract classes in XML

XML supports the concept of an abstract class directly. The XML mapping of the UML diagram for the abstract `GraphicsObject` class shown in Figure 4.14, " GraphicsObjects is an abstract class with an abstract draw method. " is simply:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'>

  <xs:complexType name="GraphicsObject" abstract="true">
    <xs:sequence>
      <xs:element name="position" type="Point"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="Point">
    <xs:sequence>
      <xs:element name="x" type="xs:decimal"/>
      <xs:element name="y" type="xs:decimal"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="Drawing">
    <xs:sequence>
      <xs:element name="graphicsObject" type="GraphicsObject" maxOccurs="unboun
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

3.15. Multiple inheritance

In UML a class can have multiple superclasses. In that case the subclass will inherit all the instance members of all its superclasses. Furthermore, *one will be able to substitute an instance of the sub-*

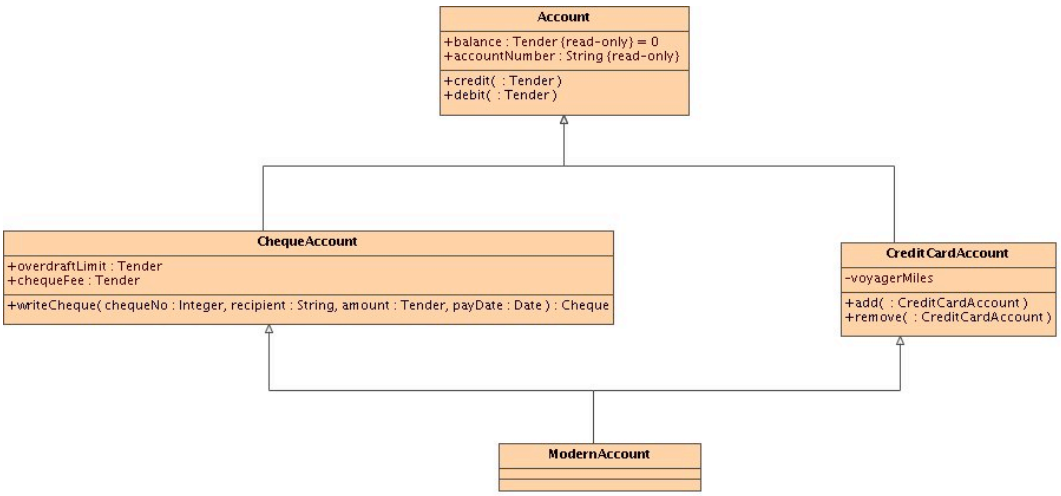
class for an instance of any of its superclasses.

Example 4.2. Credit-card cheque account

Assume a bank has cheque accounts and credit cards. Assume management decides to introduce a new account, lets call it a `ModernAccount` which acts as a credit card account and as a cheque account with a single consolidated balance and statement.

This account is to be substitutable for both, cheque accounts and credit card accounts. It would also inherit all the features and services of both, credit card and cheque accounts. The corresponding UML design is shown in Figure 4.15, " A CreditCardChequeAccount is a CreditCardAccount and a ChequeAccount."

Figure 4.15. A CreditCardChequeAccount is a CreditCardAccount and a ChequeAccount.



3.16. Implementing multiple inheritance in Java

Java does not support multiple inheritance of classes. The full cient-side behavior of multiple inheritance can be obtained by using interfaces. This will be discussed in Section 5.1.4, "Using interfaces to provide partial support for multiple inheritance".

3.17. Implementing multiple inheritance in C++

C++ directly supports multiple inheritance. For example, to specify that the class, `ModernAccount` should multiply inherit from `ChequeAccount` and `CreditCardAccount`, one uses

```
class ModernAccount: public virtual ChequeAccount, public virtual CreditCardAcc
{
  ...
};
```

Note

If the inheritance links between `CreditCardAccount` and `ChequeAccount` are not declared virtual, two accounts, each with their own account number and balance, would be inherited. This is never correct -- in that case it would be a "*has a*" relationship, perhaps

composition.

3.18. Implementing multiple inheritance in XML

XML and in particular XML schemas do not currently support multiple inheritance.

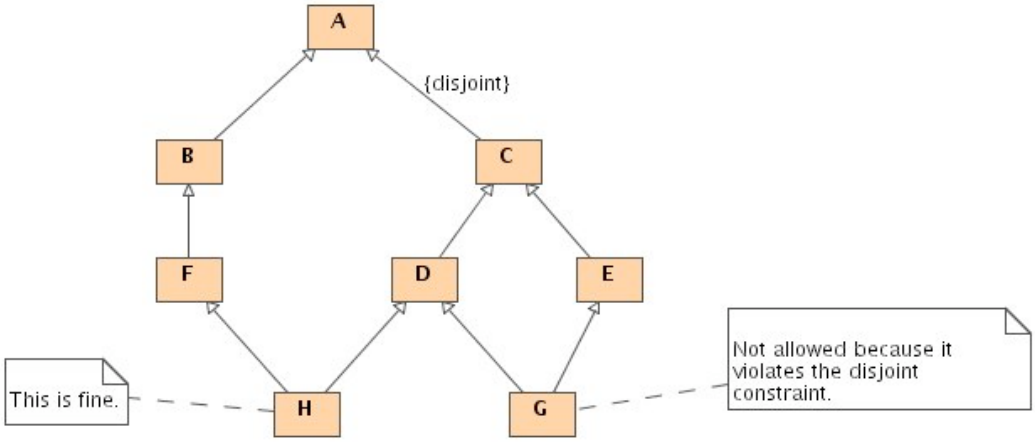
3.19. Applying constraints during sub-classing

UML supports the concept of applying constraints during subclassing. These constraints are drawn onto the specialization link.

3.19.1. The disjoint constraint

Applying a disjoint constraint specifies that the class hierarchy below the specialization link must form a clean tree structure, i.e. that there may be no multiple inheritance among the subclasses of the sub-class end of the specialization link. The implications of the `disjoint` constraint are illustrated in Figure 4.16, " The disjoint constraint prevents multiple inheritance from within a class hierarchy. ".

Figure 4.16. The disjoint constraint prevents multiple inheritance from within a class hierarchy.

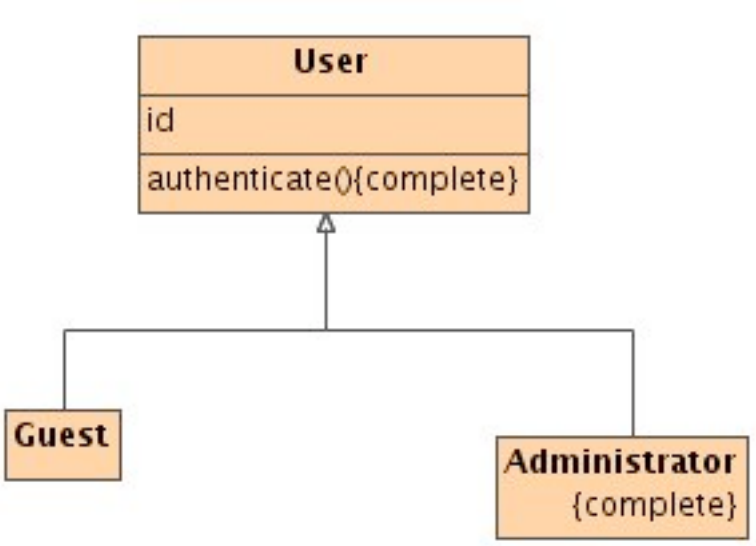


3.19.2. Preventing subclassing via a complete constraint

At times one wants to prevent subclassing. The reasons for this are often security. If one wants to be certain that an instance of a class will not be replaced by an instance of another class which potentially replaces the implementation of a service with its own realization of that service, then one can declare a class, or a class hierarchy as `{complete}`.

Alternatively, one can apply a `{complete}` constraint to a service, specifying that the class may be subclasses, but that the `{complete}` service may not be overwritten, i.e. that subclasses may not provide their own implementation of that service.

Figure 4.17. The {complete} constraint prevents subclassing when applied to a class and method overriding if applied to a method.



In Figure 4.17, “ The {complete} constraint prevents subclassing when applied to a class and method overriding if applied to a method. ” the `authenticate` service for users has been declared `{complete}` and its behavior may thus not be altered by any of its sub-classes, i.e. the method may not be overridden.

Furthermore, the `Administrator` class has been declared `{complete}` and hence every instance of an administrator will be an instance of that particular class with exactly that behavior.

3.19.3. The incomplete constraint

If one applies an `{incomplete}` constraint to a class hierarchy, one specifies that not all the classes have been identified or defined yet and that one has to revisit this class hierarchy.

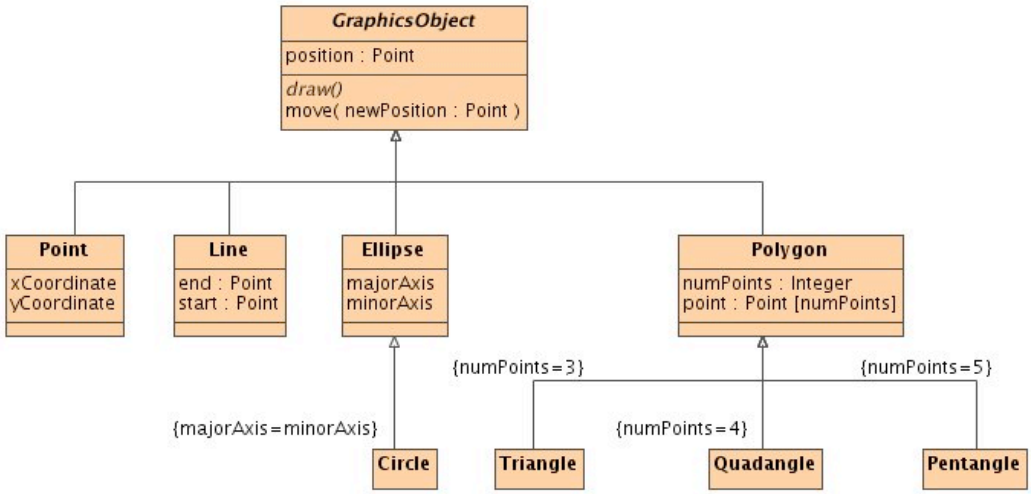
3.19.4. Extensive versus restrictive specializations

In object-orientation one can have both, extensive as well as restrictive specialization. Even XML supports both of these.

So far we have had examples which use extensive specialization. For example, a cheque account extends an account and adds the concept of cheques and a cheque fee. Similarly, an employee extends a person, adding a salary.

There are, however, cases where the subclass does not add anything to the superclass, but only applies certain constraints on the features inherited from the superclass. Consider, for example the class hierarchy of graphics objects shown in Figure 4.18, “ During restrictive specialization the subclass only applies constraints to the elements inherited from the superclass. ”. Here circle *is an* ellipse and triangles, quadangles, ... are polygons with rectangle being a special form of a quadangle.

Figure 4.18. During restrictive specialization the subclass only applies constraints to the elements inherited from the superclass.



Circle does not add any features to ellipse -- it simply applies the constaint that the major axis is equal to the minor axis. Similarly, quadangles are polygons with the number of points constrained to 4 and rectangles apply further positioning constraints on the four points of a quadangle.

We may also use a combination of restrictive and extensive specialization. In this case the subclass applies certain constraints to the elements inherited from the superclass *and* adds other elements which are not present in the subclass.

We had an example earlier, that of a savings account shown in Figure 4.13, " A subclass inherits all instance members (attributes and operations) of the superclass. ". The `SavingsAccount` added to `Account` the interest rate it earned but at the same time constrained the account balance to a positive balance.

3.20. Enforcing specialization constraints in Java

The `{disjoint}` constraint implicitly always applies in Java since multiple inheritance is not supported. The `{incomplete}` constraint is not relevant at programming language level.

Java provides explicit support for the `{complete}` constraint on both classes and services. The keyword used in Java is `final`:

```
public class User
{
  public final void authenticate() {...}

  public String getId() {return id;}

  private String id;
}

public final class Administrator extends User {...}
```

3.21. Enforcing specialization constraints in C++

In C++ there are no language constructs through which any of the UML specialization constraints can be enforced.

3.22. Enforcing specialization constraints in XML

The `{disjoint}` constraint implicitly always applies in XML since multiple inheritance is not supported.

XML provides explicit support for the `{complete}` constraint on classes (services are not relevant for XML). To declare a class (type) as `{complete}` one assigns the value of the attribute `final` to `true`:

```
<complexType name="Administrator" final="true">
  <complexContent>
    <extension base="User">
      ...
    </extension>
  </complexContent>
</complexType>
```

3.23. Lessons from Design-By-Contract

Design by Contract is a design technique developed by Bertrand Meyer, the developer of the programming language Eiffel which uses this approach as one of the core language features. The contract approach is, however, equally applicable for business modeling and requirements modeling.

3.23.1. Pre-conditions, post-conditions and invariants

Design by contract is based on the premise that the requirements for a service can be specified by three constructs:

1. The *pre-conditions* are checked by a service provider before the service is provided. If the pre-conditions are not met, the services provider raises an exception notifying the client that the service he/she requested is not going to be provided.
2. The post-conditions are the deliverables of a service, i.e. what, according to contract, must be provided to the client upon successful completion of the service.
3. The invariants are the symmetries of the system. These symmetries define certain rules (e.g. business rules) which are non-negotiable. They may be violated temporarily while a service provider is busy providing a service but they must hold before the service is requested as well as after it has been provided.

3.23.2. Example: the debit service

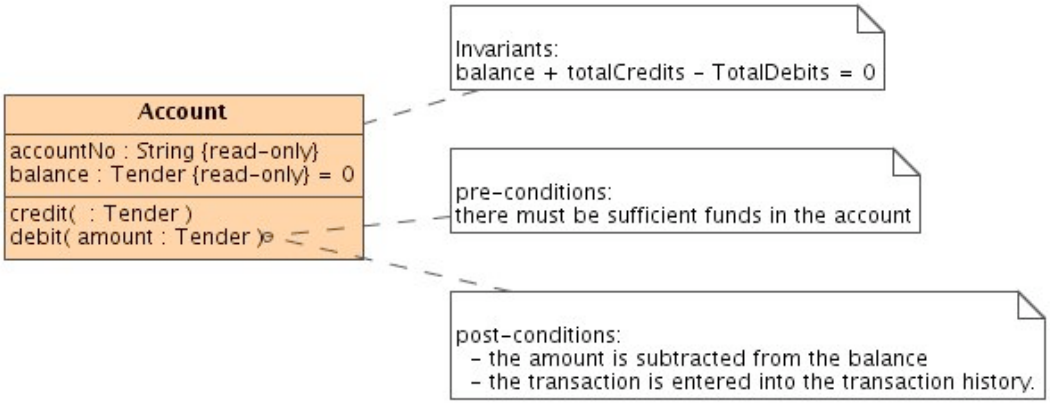
Let us look at the `debit` service of our simple `Account` from the perspective of *design by contract*:

- *Pre-condition:*
  1. There must be sufficient funds in the account.
- *Post-Conditions:*
  1. The account has been debited with the supplied amount.
  2. A debit transaction has been written into the account's transaction history.

- *Invariants:*

  1. The balance plus the total credits minus the total debits is zero.

Figure 4.19. Pre- and post-conditions and invariants can be specified in UML with corresponding pre- and post-condition comments.



UML supports the specification of pre- and post-conditions as well as invariants in corresponding pre- and post-condition and invariant comments attached to a service (see Figure 4.19, " Pre- and post-conditions and invariants can be specified in UML with corresponding pre- and post-condition comments. "). The comments may be written in the formal *Object Constraint Language* (OCL) or less formally in text. This is not a particularly elegant solution. These aspects will be addressed in a more satisfactory way with the UML 2.0 specification.

3.23.3. Design by contract and overriding methods

Design by contract provides two simple rules one should keep in mind when subclass replaces the realization of a service in a superclass with its own implementation of that service. Both rules can be seen as a direct consequence of the substitutability of subclassing:

1. **Pre-conditions may not be increased.** The realization of the subclass service must be available under no more stringent conditions than what the superclass service is. In otherwords, the set of pre-conditions for the sub-class implementation of the service must be fully contained in the set of pre-conditions which apply to the implementation of that service in the superclass. Or put yet another way, the subclass service may not raise any exceptions which are not potentially raised by the corresponding superclass service.

   ### Reason

   A client may request an instance of the superclass. You may always give him an instance of one of the sub-classes instead. He/she knows that the object provides a certain service as long as a specific set of pre-conditions is met. If the subclass instance violates this, substitutability is violated.

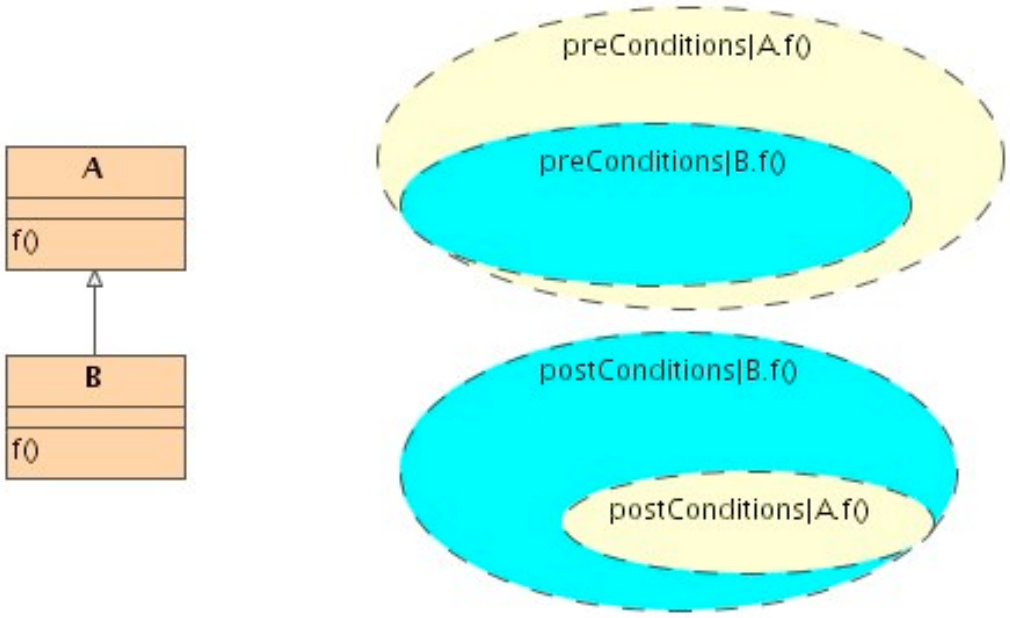
2. **Post-conditions may not be decreased.** The realization of the subclass service must provide at least all those deliverables which the corresponding service of the superclass provides. It may, though, provide additional deliverables which are not provided by the corresponding service of the superclass.

   ### Reason

A client may request an instance of the superclass. You may always give him an instance of one of the sub-classes instead. He/she knows that a service of the instance provides certain deliverables. Substitutability would be violated if the sub-class service does not provide at least the deliverables the superclass service provides.

These rules are graphically illustrated in Figure 4.20, “ When overriding a method the pre-conditions may only be decreased and the post-conditions may only be increased. ”.

Figure 4.20. When overriding a method the pre-conditions may only be decreased and the post-conditions may only be increased.



3.23.4. Example: Overriding the debit service

Let us look at the above rules from the perspective of our accounts. The first rule states that if we say that accounts can be debited as long as there are sufficient funds, then a specialized account (e.g. a cheque account) may not refuse the service for any other reasons that that of insufficient funds -- it may not, for example, raise an exception because the debit service was requested on a sunday.

Furthermore, the debit service of `Account` subtracts the debit amount from the balance of the account. Cheque accounts must do at least this. They may, however, add the transaction to a transaction history and may additionally raise a cheque fee.

3.24. Implementation guidelines from design-by-contract

From the implementation perspective one has to ensure that when a method is overridden, that the pre-conditions are never increased and that the post-conditions are never decreased. To this end the replacement implementation should

- **Not throw more exceptions than the implementation in the superclass.** The pre-conditions are checked before the service is provided and, if not met, the service provider throws an exception notifying the client who requested a service that the pre-conditions have not been met. Since

the pre-conditions may not be increased when overriding a method, the subclass implementation of a service may not throw any exceptions which are not potentially thrown by the superclass implementation of that service.

- **If possible, the subclass implementation of a service should call the superclass implementation .** Doing this enforces that all the superclass post-conditions will be met. The subclass implementation may then go ahead and add additional deliverables.

3.25. Alternatives to sub-classing

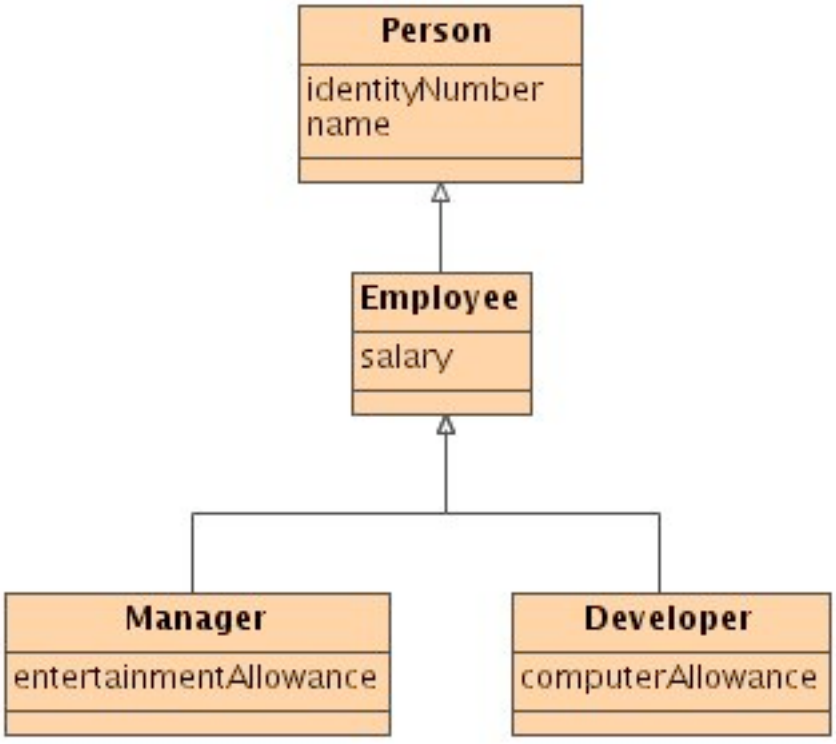
Direct sub-classing is not always the desirable analysis or design solution. At times we may want to map a specialization hierarchies onto another, more flexible structure.

One approach is to decouple completely from the concrete realizations of the various concepts via interfaces. Another approach which can provide a more flexible solution than sub-classing is that of mapping specialization onto composition.

3.25.1. Mapping specialization onto composition

Consider, for example, the scenario of employees. We may initially identify developers, managers, personal assistents and so forth as specializations (sub-classes) of the class `Employee`.

Figure 4.21. Inheritance hierarchy for employees.

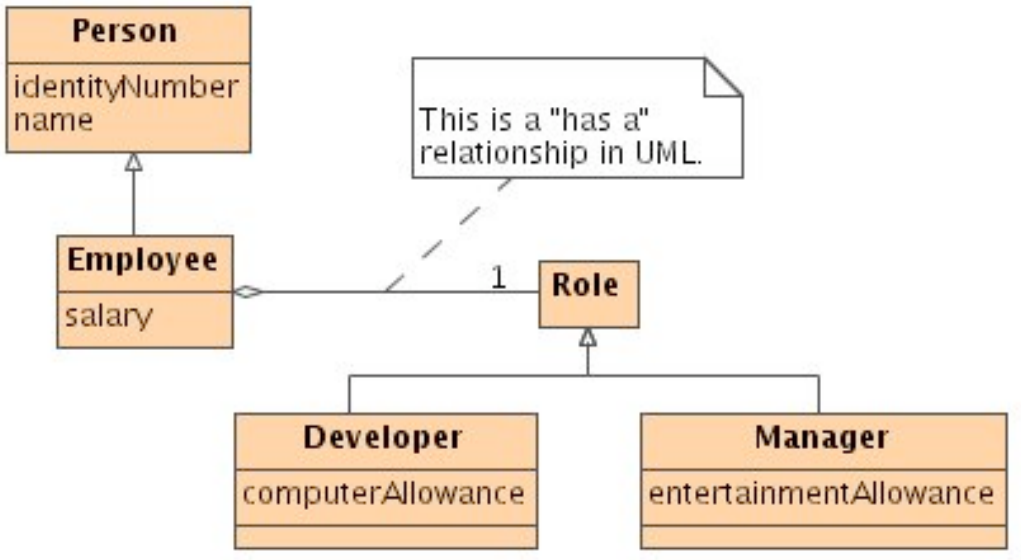


This analysis solution has, however, some quite serious drawbacks. Consider the case where a developer changes his/her job to become a manager in the same organization. It is still the same person and the same employee and deleting the developer with all his/her history to create a manager for the same employee sounds at least un-ethical if not down-right wrong.

In Figure 4.22, “Using roles for employees.” we have mapped the specialization relationship onto

roles. Now an employee simply acquires a new role in an organization when he/she changes job-description.

Figure 4.22. Using roles for employees.



4. Interfaces

So, what is an interface? An interface

- provides a mechanism for decoupling from any particular service provider and
- specify the services you require and the messages you will send when requesting these services.

Service providers can then implement these interfaces by providing the services specified in the interface. A client can use any of the service providers which implement the interface -- they are decoupled from a concrete realization of a service provider and hence avoid vendor locking.

4.1. Some example interface specifications

In both, the software development world and the manufacturing world, these interfaces are of paramount importance.

Example 4.3. Computer monitors

There is, for example, a standard interface for a computer monitor. The interface specifies the messages a monitor must be able to process. Having the standard interface avoids vendor locking. You can go ahead and buy from any manufacturer a monitor which implements the specification defined in the interface and you know that you can plug it in and it will work.

Example 4.4. Low-level component framework: CORBA

CORBA, the *Common Object Request Broker Architecture* is an object-oriented component frame-

work which enables one object to make use of the services of another object without being exposed to any of the implementation details like development language, location of the service, network protocols required to communicate with the object and so forth.

The entire CORBA specification is an interface specification -- the OMG itself has never provided an implementation. Many companies (e.g. IBM, Sun, HP, IONA, ...) and many open source development teams have provided concrete realizations (implementations) of the specification.

Example 4.5. Business logic containers: Enterprise Java Beans

Enterprise Java Beans is an interface-driven specification which is itself built on top of CORBA. It enables component developers to develop business logic components which can be deployed in an application server from any vendor and plugged into the higher-level business processes of different organizations.

Being largely an interface specification, EJB and J2EE, are vendor neutral business logic and enterprise architecture frameworks respectively, and unlike *.Net* for which there is effectively only a single vendor, there are many companies (e.g. BEA, IBM, Borland, ...) and a few open source implementations (e.g. JBoss, Jonas, ...) which realize this specification. Being an interface specification thus avoids vendor locking,i.e. that you are not left to the mercy of a single vendor and that the costs for migrating from one vendor solution to another are moderate to low.

Example 4.6. Business-2-business services via Web Services and SOAP

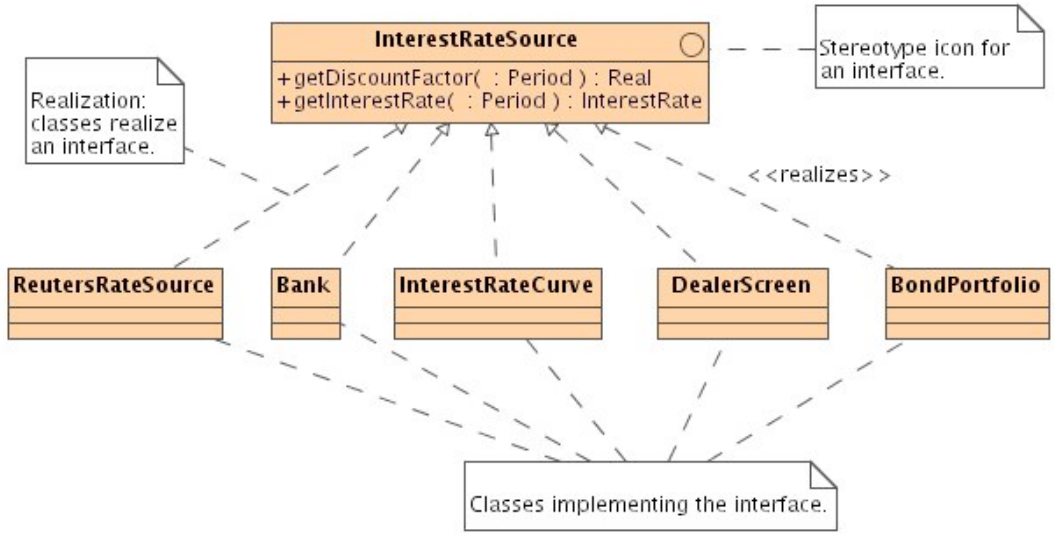
The last example of an interface specification which is making a large impact on the ways businesses operate is Web Services. Web services and its corresponding XML-based service request protocol, SOAP are expected to be the main driver in vendor and implementation technology neutral business-to-business integration.

So, one organization may decide to use *J2EE* for their enterprise architecture while another may decide to use *.Net*. Their corresponding systems can participate in business-to-business integration via web services without the one being exposed to any of the implementation specifics of the other.

4.2. Defining an interface in UML

An interface is documented in UML using a class with the interface stereotype assigned to it. The stereotype, `<<interface>>`, is inserted into the top compartment (the class-name compartment), either literally or using the stereotype icon for an interface which is a circle.

Figure 4.23. The interest rate source interface specifies the services which must be supplied by interest rate sources. Differnt service providers may realize these services in different ways.



In Figure 4.23, " The interest rate source interface specifies the services which must be supplied by interest rate sources. Differnt service providers may realize these services in different ways. " we show an `InterestRateSource` interface which states that any interest rate source must provide two services providing interest rates and discount factors over specified periods respectively.

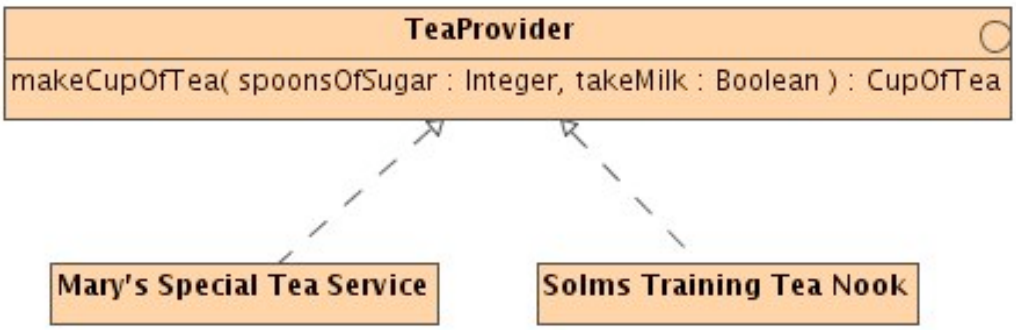
4.3. Specifying interface realizations

A service provider may decide to provide the services specified in an interface. This will enable clients who require those services to make use of that service provider.

Service providers who implement an interface must

- provide a concrete realization (implementation) of the services specified in the interface and must
- make these services available through processing messages of the signature specified in the interface.

Figure 4.24. Two tea provider realizations which provide the makeTea service.



For example, in Figure 4.24, " Two tea provider realizations which provide the makeTea service. ", we have an interface, `TeaProvider`, with two service providers who implement the interface and

hence provide the service `makeTea`.

Note that the classes which implement the interface may have very little in common bar that they, among other things, provide the services specified in the interface. For example, have another look at the interest rate sources shown in Figure 4.23, " The interest rate source interface specifies the services which must be supplied by interest rate sources. Differnt service providers may realize these services in different ways. ".

Reuters, as financial information provider, provides the market information against a fee. A bank will provide interest rates for investements in its products. Alternatively you might have calculated your own interest rate curve (getting interest rates from various banks, for example) and the resulting interest rate curve could be queried for an interest rate or a discount factor over a specified period. Alternatively you may be working in an organization which employs interest rate dealers who know the market very well. You could pop up a little window on a dealer terminal asking him/her for the market rate for a specific investment period. Finally, if you have a portfolio of priced bonds, you can also strip (calculate) interest rate information from them.

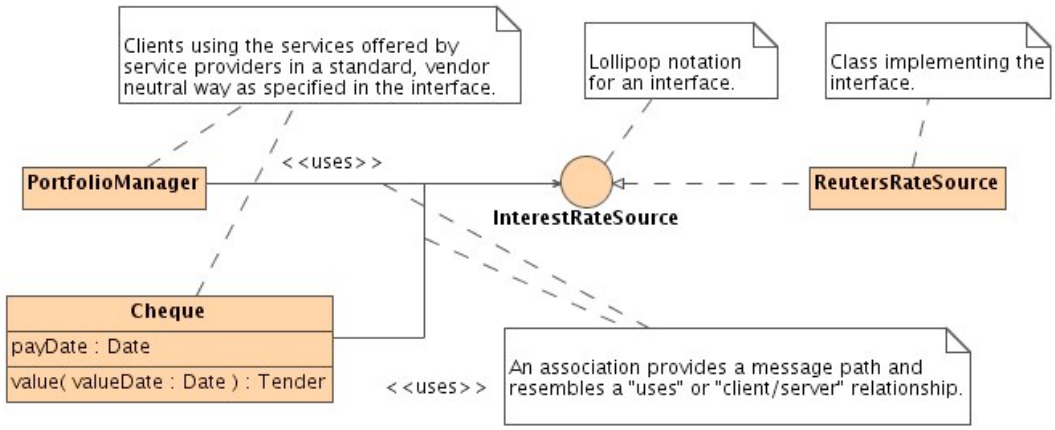
These service providers are very different ranging from a bank to Reuters to a portfolio of bonds. They have very little in common except that they all can provide interest rate information.

4.4. Provided and required interfaces

Clients would preferably not couple directly to service providers. That would result in them locking into the particular implementation specifics of that service provider and would make a transition to another service provider an expensive exercise. This is because the own business processes need to be modified to link to change from the implementation specifics of the old service provider to those of the new service provider.

Instead clients would only interface with service providers in a standard, well defined, vendor neutral way. This standard would be documented in an interface. To show in a UML diagram that clients do not interact in a direct, potentially proprietary way with a service provider, but only through standard, service-provider neutral way, wedraw the uses relationship (the association or message path) to the interface (see Figure 4.25, " Clients use an InterestRateSource only through a standard interafe, thereby avoiding vendor locking. ").

Figure 4.25. Clients use an InterestRateSource only through a standard interafe, thereby avoiding vendor locking.

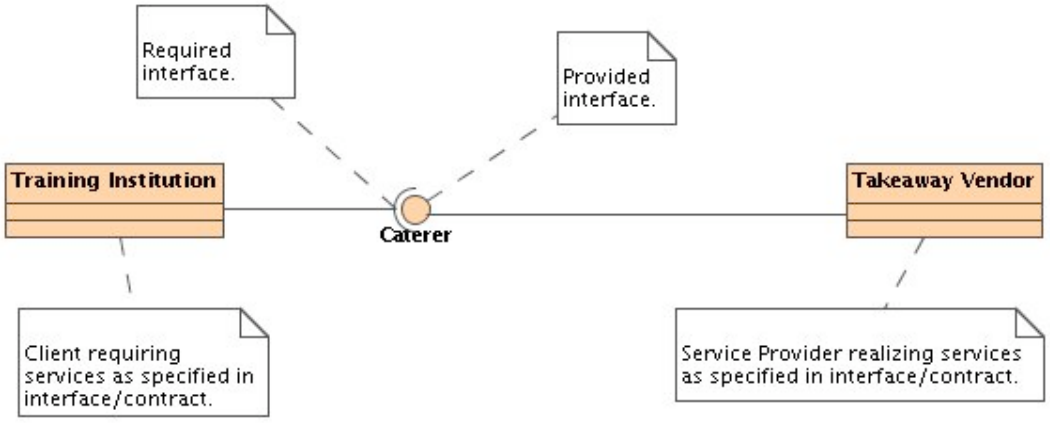


A `PortfolioManager` requires interest rates for making investment decisions and a cheque requires interest rates to be able to calculate its value on a date other than the payment date. Both make use of service providers in a way which decouples them from any concrete realization of an interest rate source.

4.4.1. Alternative notation for provided and required interfaces

UML now supports an alternative notation for highlighting that a client requires a service provider realizing a particular contract and for showing service providers which realize the contract.

Figure 4.26. Showing provided and required interfaces explicitly



4.5. Viewing an interface as the skeleton of a contract between clients and service providers

An interface can be viewed as part of a contract formalizing the relationship between a client and its service providers. The interface specifies

- the services which must be provided by service provider fulfilling the contract and
- the messages which will be sent when requesting the service as well as the return value of the service.

4.5.1. Aspects of the contract not included in the interface

There are, however, elements which need to be included in the contract between a client and its service providers which are not included in the interface. In particular

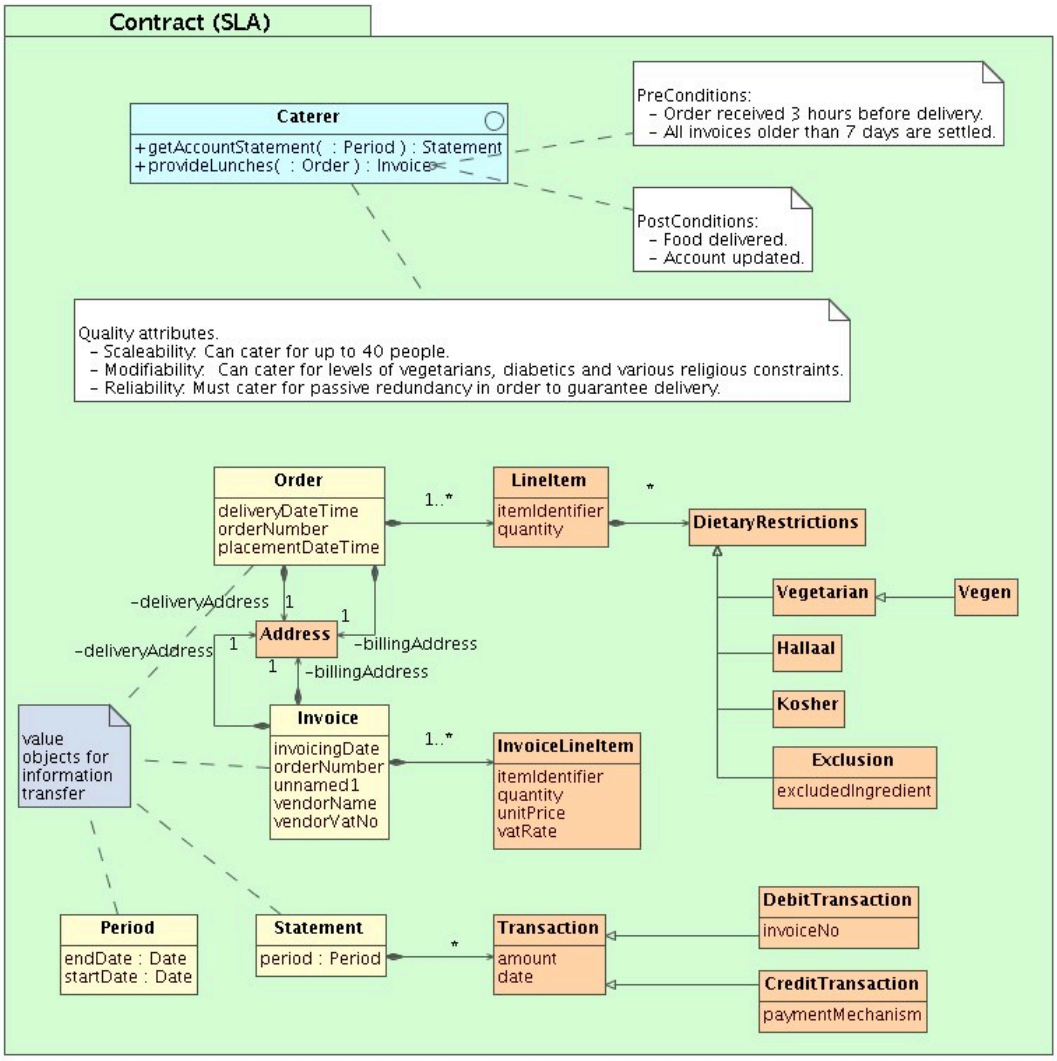
- the *pre-conditions* which must be met before the service provider is able or willing to provide the service (recall that a service provider may raise an exception signalling to the client that the service requested is not going to be supplied).
- a complete list of the post-conditions, i.e. all the deliverables of the service,

The first two aspects can (and should) be handled in the framework of *design-by-contract*. All three aspects are handled in UML by further formalizing the contract skeleton provided by the interface with sequence and activity diagrams which fully specify the interaction between a client and its service providers. We will cover these aspects in detail in the discuss using UML for business and requirements modeling.

4.5.2. SLA for a Caterer

We can use these concepts to define a Service Level Agreement between a client and service providers. Different serbice providers would typically realize the same contract. An example of such a contract is shown in Figure 4.27, “SLA for a caterer”.

Figure 4.27. SLA for a caterer



4.6. Implementing multiple interfaces

A service provider can implement multiple interfaces, thereby simply making more promises (i.e. stating that there are more services which are going to be supplied in a standard way as specified in the interface).

For example, as more banks enter the bank-assurance market, many of them provide the services of both, a `Bank` and an `Insurance company`. One can model that in UML as classes implementing multiple interfaces as is done in Figure 4.28, " Some banks have entered the bank-assurance model providing the services of both, a bank and an assurer while other remain pure banks. ".

Figure 4.28. Some banks have entered the bank-assurance model providing the services of both, a bank and an assurer while other remain pure banks.

4.7. Extending interfaces

UML also supports the concept of one interface extending another. The extended interface requires any classes which realize it to implement all its services as well as all the services of the interface it extends.

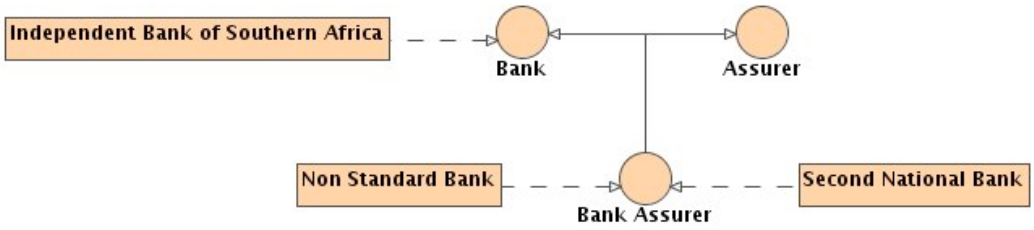
Figure 4.29. Documents are Printable and hence also Viewable, providing both, print and show services.

In Figure 4.29, " Documents are Printable and hence also Viewable, providing both, print and show services. " any class which implements `Printable` must provide both, `print` and `show` services.

4.7.1. Extending multiple interfaces

An interface can also extend multiple interfaces. For example, in Figure 4.30, " Introducing a concept of a bank assurance business with different organizations realizing that concept. " the `BankAssurer` interface specifies that any organization entering the bank assurance markets will have to supply the services of both, a bank and an assurer. The `Non Standard Bank` as well as the `Second National Bank` realize the concept of a `Bank Assurer`, while the `Independent Bank of Southern Africa` still sticks to the clean banking model.

Figure 4.30. Introducing a concept of a bank assurance business with different organizations realizing that concept.



4.8. Benefits of using interfaces

So, in summary, what a the benefits of using inerfaces?

1. Interfaces enable one to localize the interface specification at in one unit, facilitating the decoupling of the service implementation from the interface through which the service is requested. The various implementations can be developed independently after the aggreement has been achieved on the interface specification.

2. Interfaces facilitate a plug-&-play approach to service providers. These service providers may be organizations, business units, hardware components or software components. They thus provide a mechanism through which vendor locking can be avoided.

5. Implementing interfaces

Here we show mappings of UML interfaces onto Java and C++.

5.1. Implementing interfaces in Java

The support for interfaces is Java is a direct mapping of what the UML provides using the same terminology as what is used in the UML.

5.1.1. Defining and interface

An interface is defined as such where a Java interface can only contain abstract methods, constant data fields and nested classes and interfaces. The methods of an interface are implicitly abstract -- they are not declared `abstract`.

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

5.1.2. Implementing an interfaces

The `implements` keyword is used in Java to specify that a class implements the interface. If that class does not provide an implementation of the methods specified in the interface, the compiler will force you to declare the class abstract because instances of the class would be objects which claim to implement the interface but have not yet fulfilled all the requirements laid down in the interface.

Note

The compiler enforces the specification at instance level.

For example, to specify that our `InterestRateCurve` fulfills the requirements laid down for interest rate sources, we specify

```
public class InterestrateCurve implements InterestRateSource
{
  ...
  public double getDiscountFactor(Period period) {...}
  public InterestRate getInterestRate(Period period) {...}
  ...
}
```

5.1.3. Extending interfaces

In Java the keyword `extends` is used to specify that one interface extends another:

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

5.1.4. Using interfaces to provide partial support for multiple inheritance

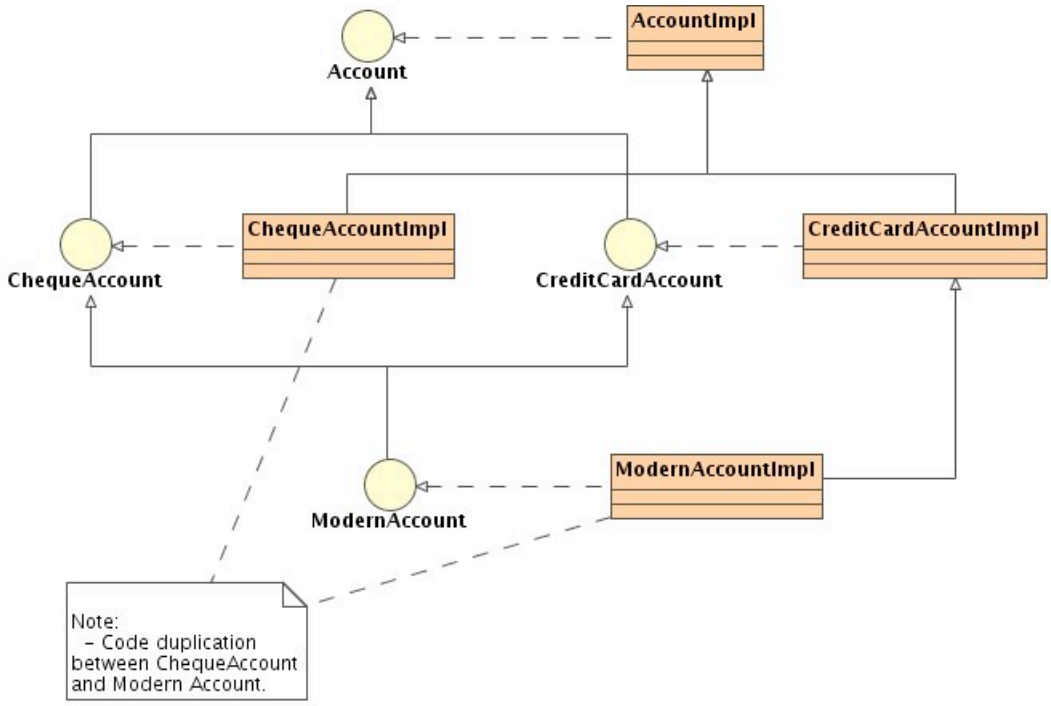
Java does not support multiple inheritance of classes. We can, however, use interfaces to provide the from a user's perspective full multiple inheritance, though, from an implementation perspective we will have to pay the price of some code duplication.

This requires that users decouple via interfaces (which they anyway should) from the actual classes. At interface level we have multiple inheritance. For each interface we define an implementation class and where possible, we let them inherit from each other. However, `ModernAccount` cannot, in Java, inherit from both, `ChequeAccount` and `CreditCardAccount`.

Note

We can reduce the amount of code duplication by encapsulating the shared code in another helper class which is used by both, `ChequeAccount` and `ModernAccount`, but we cannot get rid of it completely.

Figure 4.31. Realizing multiple inheritance in a language which does not support it



The design is shown in Figure 4.31, “Realizing multiple inheritance in a language which does not support it”.

Note

We use the more natural name for the interface and a more convoluted name for the class since most of the code will work at interface level.

5.2. Implementing interfaces in C++

Unlike Java, C++ has no direct support for interfaces. Instead

- interfaces are modeled as abstract classes with only abstract methods,
- an interface is implemented by extending the class which represents the interface, and
- an an interface is extended by defining a class which represents the extended interface as a abstract class with only abstract which extends the class which represents the super-interface.

6. Ports

A port represents an interaction point between a classifier, the context, and its actors. Typically the context makes a set of services available through the port, but may also, at times, request services from other service providers through that port.

6.1. Specifying ports

In UML one can use ports to specify interfacing objects for a context. A port is specified in UML by attaching a port symbol to a class. The port itself is usually an instance of a class. One thus specifies the class when specifying a port and may assign optionally an object name.

Service providers may make the same services availible through multiple ports. Also, at times, a

port may be used to not only accept service requests, but also to send service requests to external service providers.

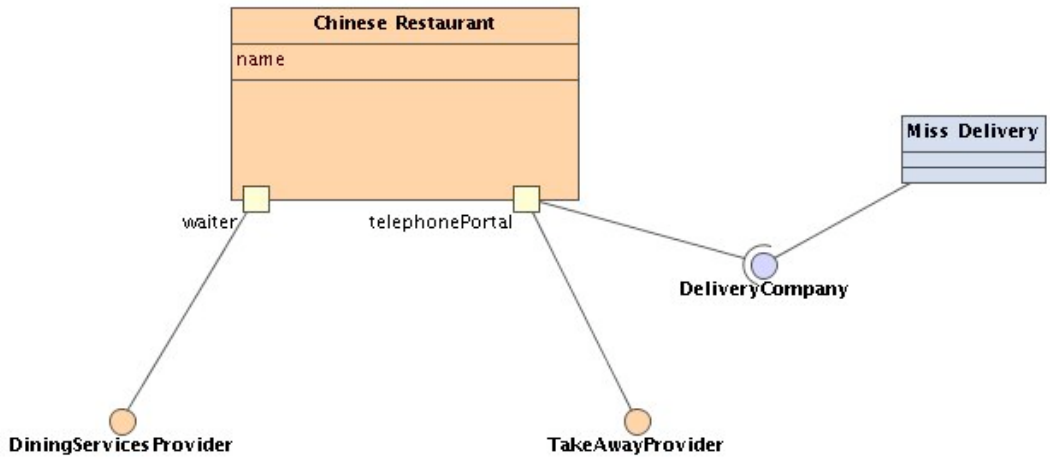
6.1.1. Portals for a restaurant

As an example, lets look at a restaurant. Assume external objects interface with the restaurant through two core ports:

- The waiters port is used to interface with dining clients.
- A telephone port is used
  - by take-away clients to place takeaway orders, and
  - by the restaurant to place orders with external suppliers.

Figure 4.32, “Ports for a restaurant” shows how external objects interface with restaurants through a waiter and a telephone port.

Figure 4.32. Ports for a restaurant

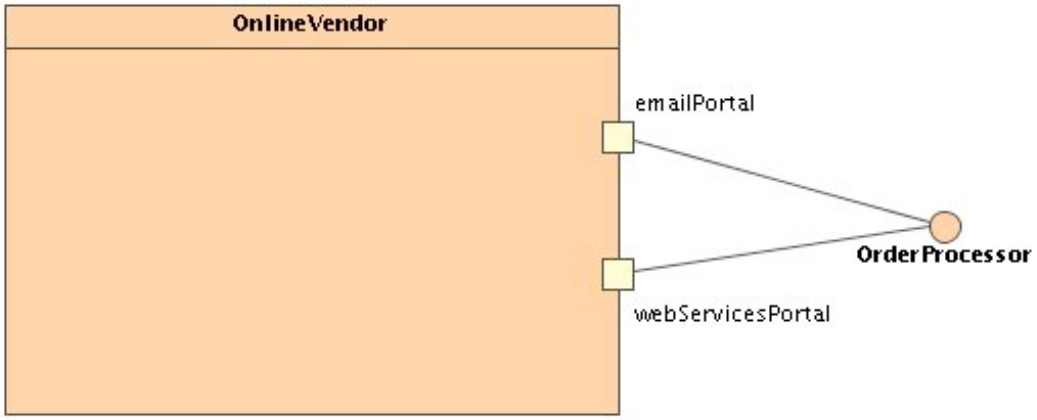


6.2. Benefits of using ports

The benefits of using ports is that one can specify various portals through which services are published. The same services could, at times, even be made available through different ports.

For example, an online vendor may offer the service to process an order via both, a web services and a e-mail portal.

Figure 4.33. Offering the dame services through different portals



7. Composition

Specialization is a very strong relationship -- it is an *is a* relationship where every instance of the subclass is a special instance of the superclass. The next weaker relationship is *composition* which is a *strong has a* relationship. This is one of the core relationships facilitating the recursive decomposition of high-level objects into lower level objects.

7.1. What is composition?

Composition is a *strong has a* relationship where the components

- are elements (parts) of the owner,
- form part of the state of the owner, i.e. a state transition in the component results in a state transition in the owner object,
- can be accessed only through the owner, i.e. the components are encapsulated in the owner,
- the components life span is limited to that of the owner, i.e. if the owner object is destroyed, the components are destroyed too and
- where the owner is responsible for its components.

Furthermore, the owner object takes full responsibilities of its components (i.e. if one of the components fails and the owner does not handle this problem, then the owner object fails).

Example 4.7. Cheque

Let us look at the components of a cheque. The cheque has (among other things) a `payDate` and an `amount`. These are components of the cheque because

- if either the `payDate` or the `amount` of the cheque change, the cheque itself changes (a state transition in the component result in a state transition in the owner),
- you can only change the `payDate` or the `amount` of the cheque by modifying the `Cheque` itself, and
- if the `Cheque` is destroyed, the `amount` and the `payDate` are destroyed too.

7.2. Documenting composition in UML

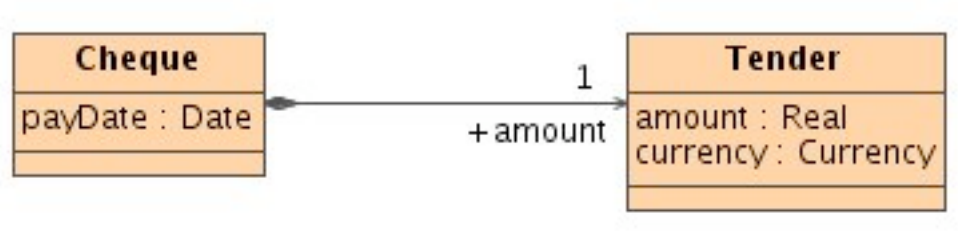
Composition is shown in UML either

- by inserting the component in the attributes block.
- or by drawing a line with a solid diamond one the owner side of the composition relationship.

The first notation allows on to show more details about the component, while the second is often used for its compactness and readability.

In Figure 4.34, " Cheques have as components an amount and a payDate. " we show both notations. The `payDate` component is shown as an attribute, while the `amount` component is shown using a UML composition relationship.

Figure 4.34. Cheques have as components an amount and a payDate.



7.2.1. Specifying role names and multiplicities

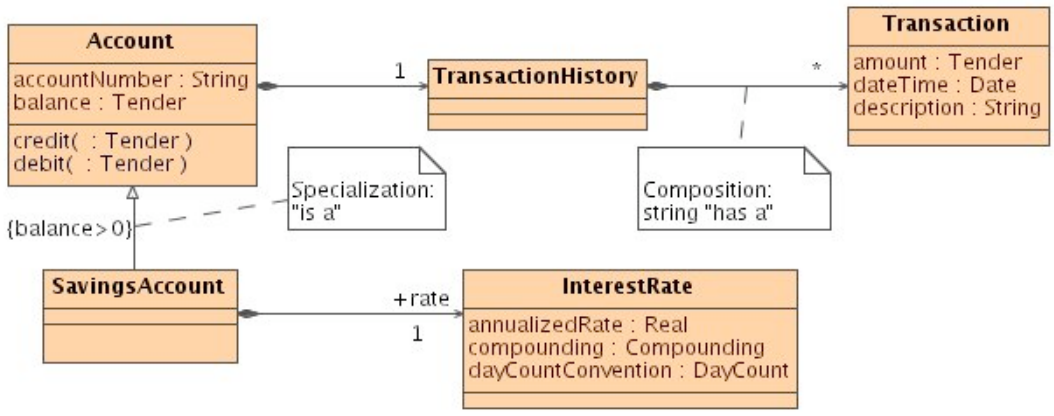
Role names map onto variable names. The instance of the `Tender` which is a component of a `Cheque` plays the role of being the `amount` of the `Cheque`.

Multiplicities are specified on the component side of the composition relationship (the owner side has always multiplicity one, i.e. a component has at most a single owner).

7.2.1.1. Example: Account with transaction history

Figure 4.35, " Accounts have a transaction history. " shows accounts with transaction histories and a savings account as a specialization of an account.

Figure 4.35. Accounts have a transaction history.



7.3. Composition as a relationship enforcing encapsulation

Just to recap, a composition relationship enforces access control, i.e. that the component can only be accessed through its owner. This effectively is a form of encapsulation.

7.4. When not to use composition

Composition is a very strong relationship, enforcing encapsulation, state transition propagation and the owner acquiring responsibility for its components.

If the component needs to be accessed directly (not only through the owner) or if a state transition in the component should not result in a state transition in the potential owner, or if the potential owner should not acquire the responsibility for the performance of the potential component, then composition should not be used. Instead one would typically choose a weaker relationship like aggregation, association or a simple dependency.

8. Implementing composition relationships

Here we show mappings of composition relationships onto Java, C++ and XML.

8.1. Implementing composition in Java

When implementing composition one should be led by the definition of composition which requires encapsulation and that the state of the component may be part of the state of the owner. We thus need to implement that

1. elements can be accessed only through the owner,
2. component do not survive the owner, and
3. that a state transition in the component results in a state transition in the owner.

8.1.1. Enforcing ownership and bounded life-span

To this end we must *ensure that only owners have references to their components*. This will ensure that the first two requirements are satisfied. The last requirement will be satisfied by virtue that of the owner is deleted the last reference is deleted and hence the component will be garbage collected. This is because no object other than the owner or one of its components (which also will be garbage collected) will have a reference to the component.

So, how do we ensure that no object except the owner (or one of its components) ever has a reference to its components. One way to achieve this is to ensure that the owner makes its own copy of the component by either cloning or using a suitable constructor.

Setters would either modify the state of the encapsulated object or would use cloning to replace the encapsulated object by a new instance. Getters would typically provide clients with either

- a read only view on the component via a suitable interface, or
- provide the client their own copy which they can manipulate as they desire (typically again via cloning).

Example 4.8. Enforcing true composition on the components of a cheque

```
import java.util.Date;

public class Cheque
{
  public Cheque(Date date, double amount)
  {
    this.date = (Date)date.clone();
    this.amount = amount;
  }

  public Date getDate() {return (Date)date.clone();}

  public void setDate(Date newDate) {this.date = (Date)newDate.clone();}

  public double getAmount() {return amount;}
  public void setAmount(double newAmount) {amount = newAmount;}

  public Object clone()
  {
    Cheque copy = null;
    try
    {
      copy = (Cheque)super.clone();
      copy.date = (Date)this.date.clone();
    }
    catch (CloneNotSupportedException e)
      {/* never thrown,just removing pre-conditions. */}

    return copy;
  }

  private Date date;
  private double amount;
}
```

8.1.1.1. When can we ommit cloning?

We don't have to clone

- primitives or
- instances of immutable final classes.

The former are always copied when passed as a parameter to a service (i.e. we have a copy already). The latter cannot be modified and furthermre, one cannot substitute a mutable subclass instance because subclassing has been prevented by virtue of the class being declared `final`.

8.1.2. Supporting state change notification

You may want to support support state change notification for your classes, enabling clients to register as observers to your objects. If the observed object has components, the owner will have to register as observer with the components in order to receive state change notification messages from its components. upon receipt of such a state change notification message, the owner would itself issue suitable state change notification events to its observers.

In Java the Java Beans framework provides support for state change notification.

8.2. Implementing composition in C++

The implementation of composition in C++ is virtually identical to that of java where one should also include a polymorphic clone method instead of the copy constructor which is a class service

and hence not resolved polymorphically.

There is, however, the additional aspect of memory management which applies to C++ and any other programming language wich does not support automatic memory management through a mechanism like garbage collection.

8.2.1. Composition as a mechanism for simplifying memory management

Very many application suffer from memory management problems which typically manifest themselves either in the form of protection exceptions or in the form memory leaks. This is true even for many so-called mature applications which have been around formany years.

The reason for this is that memory management is difficult. How does an object know that another object it has been using can be garbage collected? Are there perhaps other objects which have a handle (e.g.a pointer) to that object which will attempt to make use of it at a later stage? Perhaps this object is the only one which can delete the object it has been using since no other object has a handle to it and not deleting it would result in a memory leak. But how does one know this?

The following guidelines should help to transform a near-impossible problem to a manageable problem:

1. Enforce that ever object has ultimately an owner through true composition.
2. Only the owner and its components can reference a component of that object.
3. Only the owner will delete its components. The components can only be used through it (access control) and hence no object but its own components can use that component.
4. Strive for a cleanly layered structure where the owner objects of one layer of granularity take over the memory management responsibilities of the components in the next layer of granularity.

8.3. Mapping composition relationships onto XML

XML does not directly support composition because it provides no alternative access to querying information (has no methods) and encapsulated information would have no use.

9. Aggregation

Aggregation is a *weak has a* relationship. The components of the aggregate object still form part of the state of the aggregate object (i.e. a state transition in the component still results in a state transition in the aggregate object), but *the component is not owned by the aggregate object*. The consequences of this are

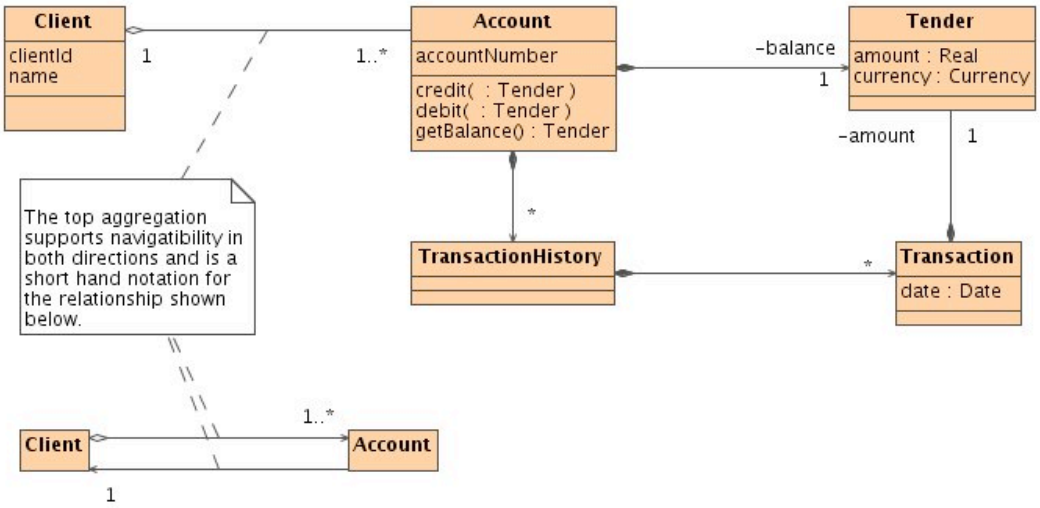
- the component can be accessed directly without going through the aggregate object,
- the component may survive the aggregate object,
- the same component may be a component of multiple aggregate objects,
- the aggregate object does not typically take responsibility for its components.

9.1. UML notation for aggregation

The UML notation for aggregation is similar to that for composition except that a hollow diamond is

used instead of the solid diamond. The notation is illustrated in Figure 4.36, “ Clients have accounts via aggregation. ”.

Figure 4.36. Clients have accounts via aggregation.



Clients may potentially have multiple acccounts. This is a *has a* relationship where a state transition in the clients accounts result in a state transition of the client (e.g. his/her credit worthyness).

However, it is a *weak has a* i.e. aggregation and not composition because the account can be credited and debited directly without going through the client.

Example 4.9. Graphics objects have styles

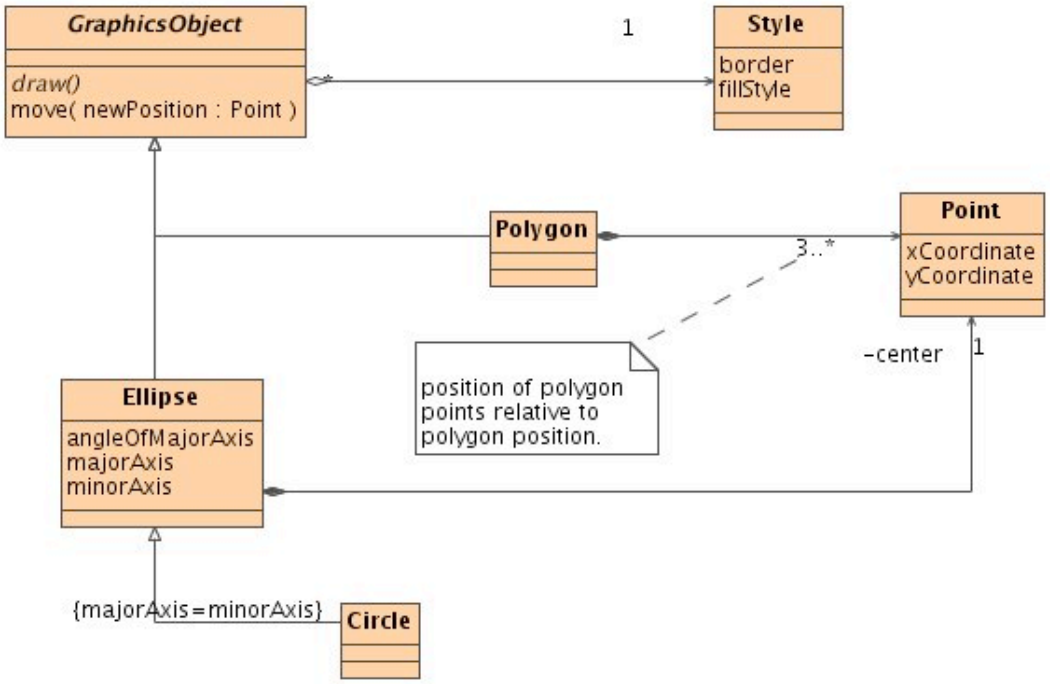
Polygons and ellipses are graphics objects. Polygons have 3 or more points via composition:

- you can modify them only by modifying the polygon,
- changing one of the points implies changing the polygon itself (state transition propagation) and
- if the polygon is deleted, its points are too.

An ellipse has a single point, the `center` of the ellipse as well as a `majorAxis` and a `minorAxis`. Circles are ellipses with the constrained that the major and minor axes are equal.

All graphics objects have a style (aggregation) defining the border and the fill style. Changing the style of a graphics object does result in a change in the graphics object. i.e. state transition propagation still holds. However, the same style object can be applied to multiple graphics object (changing it would change them all) and the style which is associated with a graphics object may continue to exist after the graphics object has been deleted.

Figure 4.37. Graphics objects have a style via aggregation.



10. Implementing aggregation

When implementing aggregation we have to implement that the state changes in a component are observable through the aggregate object.

10.1. Implementing Aggregation in Java

In aggregation the component state is part of the state of the aggregate object. In that context the result of certain query services of the aggregate object will typically change upon a state transition of the component.

10.1.1. State change notification

Often one would want to provide the facility enabling objects to be notified about state changes of some object with state. In Java it is the Java Beans framework which provides support for state change notification. If the observed object has aggregation relationships to other objects, it will have to register as state change listener, catching state change events from these components in order to send relevant state change events to its state change listeners.

10.2. Implementing Aggregation in XML

In the case where the component belongs only to a single aggregate object, aggregation maps onto XML sub elements. In the case where the component may be a component for multiple aggregate objects, the aggregation relationship is implemented in the same way as association (i.e. via keys and keyrefs).

11. Associations

An association is the next weaker relationship after aggregation. It is perhaps the most widely used relationship in UML and UML provides extensive notation for associations.

11.1. What are association relationships?

Specialization is an *"is a"* relationship and composition and aggregation are stong and weak *"has a"* relationships. The former facilitate inheritance and the ability to work at various levels of abstraction while the latter enabled us to assemble complex objects from simpler ones.

The objects of our system have to collaborate to perform the tasks required from the system. To this end they send messages to one another requesting services from one another. In order for an object to send a message to another object it needs a link to that object. A link can be seen as a message path between objects.

An association is a template from which links are generated. We shall see that associations can be modeled as classes and their instances (objects) will be links. For example, if the `PortfolioManager` class has an association to the `InterestRateSource`, then a particular portfolio manager will have a link to particular interest rate source. The portfolio manager can then send a service request message to the interest rate source requesting, for example, the interest rate applicable to an investment over a specified period.

An association can thus be seen as a *"uses"* or *"is associated with"* relationship.

11.1.1. Associations as message paths for client/server relationships

An association resembles a message path between two objects. This enables one object to request aservice from another by sending a service request message to it.

The message path may be *uni-directional* or *bi-directional*. In the case of a unidirectional path one object, the *client*, has a mesage path to another object which plays, in the context of the association, the role of a *service provider*.

In the case of a *bi-direcional association* both objects can send messages to one another. In this case each object may, at times, play the role of a client and at other times pay the role of a server. n this case we do not have a pure client server relationship where one object always plays the role of the client and the other always the role of the server. Instead we have a *request-centric client-server relationship* where the client and server roles are defined within the context of individual service requests.

11.2. UML notation for association

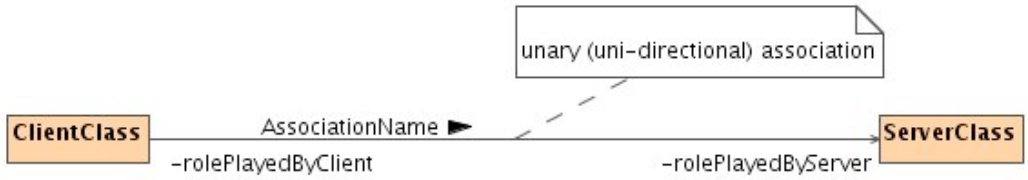
The UML notation for a associations is a solid line with potentially an arrow on one or both ends of the line. A uni-directional association represents a client-server relationship, while a binary association can be seen as a peer-to-peer relationship.

11.2.1. Unary associations for client-server relationships

A unary (uni-directional) association is shown as a line with an arrow on one of the two ends. Messages are sent in the direction of the arrow on the line.

The object which requests the the service (the source of the arrow) represents the client, while the message recipient side (the side of the arrow head) is the server side. One can add multiplicity constraints to either side of the association. The notation is shown in Figure 4.38, " Unary associations are drawn as a solid line with an arrow head on the server side of the relationship. "

Figure 4.38. Unary associations are drawn as a solid line with an arrow head on the server side of the relationship.



11.2.2. Binary associations

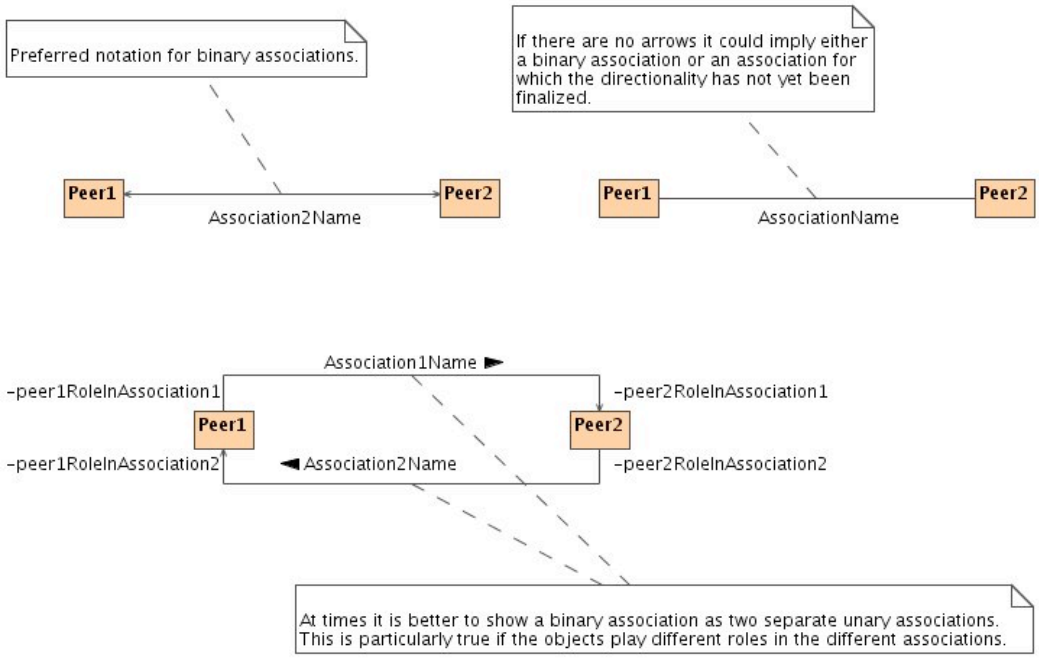
A binary (bi-directional) association represents a bi-directional message path. Either side can request services from each other.

A binary association is drawn in UML either as a solid line with arrows on both ends or as a solid line without arrows. The latter notation can also be used to specify an association for which the directionality has not yet been finalized.

This notation is a little unfortunate because a binary association is significantly more complex than a unary association. Neither of the two classes in a binary association can be tested independently. One would preferably not want to have this as a default and it would have perhaps been better if a binary association was shown exclusively by a solid line with arrows on both ends. A line without arrows could then be exclusively used in cases where the directionality has not yet been finalized.

In Figure 4.39, " UML notations for binary associations. " we show various ways to document a binary association. Conceptually a binary association can often be seen as a peer-to-peer relationship.

Figure 4.39. UML notations for binary associations.



Note

A binary association is simply a short-hand notation for two inverse unary directions. Each can have its own association name and own role names. At times it is better to draw a binary association as two separate unary associations.

11.3. Using verbs to identify associations

Recall that we used the nouns in a textual description of the system as a first step to identifying the objects. After all, if we use a noun it is an object within our conceptual understanding.

Associations can often be identified by looking at verbs. This is because associations provide message paths along which we can send our service requests.

11.3.1. A data acquisition, processing and control system

Consider the following textual description of a system:

> *" A DAPC (Data Acquisition, Processing and Control) system can be used to monitor an external system. Additionally it might control the system. There are persons which operate the DAPC system (the operators) and a person which supervises it (the supervisor). Supervisors and operators can talk to one another. "*

Focusing on the nouns, we might identify the following objects

```
dapcSystem, externalSystem, operator, supervisor
```

with the following abstractions for the classes

```
DAPCSystem, ExternalSystem, Person
```

Let us now try and identify the message paths we require. Simplifying the textual description into the simple sentence structure

Subject*verb*object.

we extract the following core sentences from the textual description:

1. DAPCSystem*monitors*ExternalSystem.
2. DAPCSystem*controls*ExternalSystem.
3. Person (operator)*operates*DAPCSystem.
4. Person (supervisor)*supervises*DAPCSystem.
5. Person (supervisor)*talks to*Person (operator).
6. Person (operator)*talks to*Person (supervisor).

These are the core statements contained in the textual description for the system. In Figure 4.40, " Mapping the textual description onto UML. " we map them onto a UML diagram. Note that the last two sentences mapped onto a binary association. We were, however, a little uncertain about the management style of the supervisor. In particular, it may not yet be clear if there is a binary or only a unary message path between them.

11.3.1.1. Reading sentences off UML diagrams

We can now read off our sentences from the UML diagram. For example

Supervisor (who is a person) supervises DAPCSystem.

DAPCSystem controls ExternalSystem

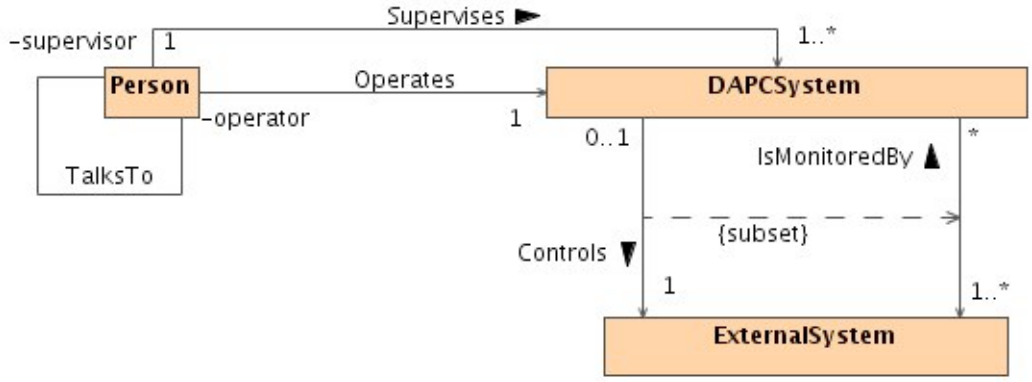
ExternalSystem is monitored by DAPC System.

The little solid arrow designates the direction in which we are to read off the sentences. This is typically, but not necessarily the same direction as what the messages flow.

If the direction of message flow is the same as the direction in which the association label is read,we usually get the normal "*Subject verb object.*" sentence structure. If the directions are different we get a more complex sentence structure (see the third sentence above).

In most cases we would resort to the two arrows pointing in the same direction (i.e. to the simpler sentence structure). In those cases the label arrow may be omitted as is done in the *Operates* association. The label arrow is, however, important for binary associations.

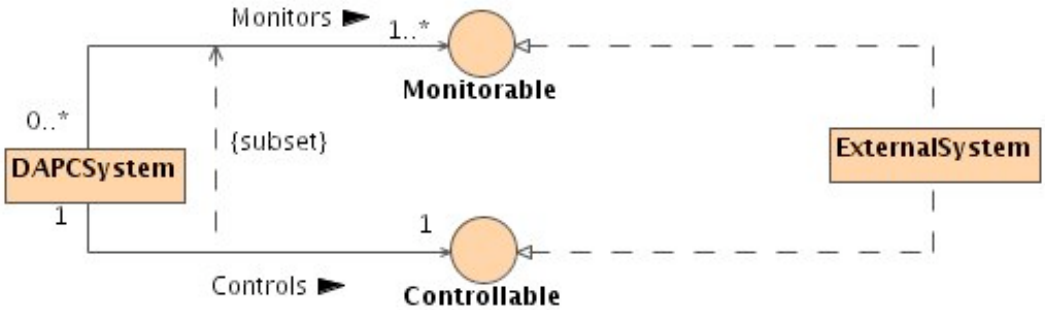
Figure 4.40. Mapping the textual description onto UML.



11.3.2. Decoupling via interfaces

The server side in an association should virtually always be an interface (normally a client would not want to lock into a particular service provider). In Figure 4.41, " Decoupling clients and service providers via interfaces. " we decoupled the DAPC system from the external system via two interfaces defining the client's requirements for the two usage scenarios.

Figure 4.41. Decoupling clients and service providers via interfaces.



11.4. Role names

The different ends of an association can assume different roles. For example, a `Person` associated with the `DAPCSystem` can be either in the role of an `operator` or in the role of a `supervisor`.

Note

The role is part of the association, not part of the class.

The role name, in UML, is specified by a text label at that end of the association which links to the class whose instance assumes the role of the role name. Role name map onto variable names. This is particularly apparent in Section 11.4.1, “Example: Bonds and interest rate sources”.

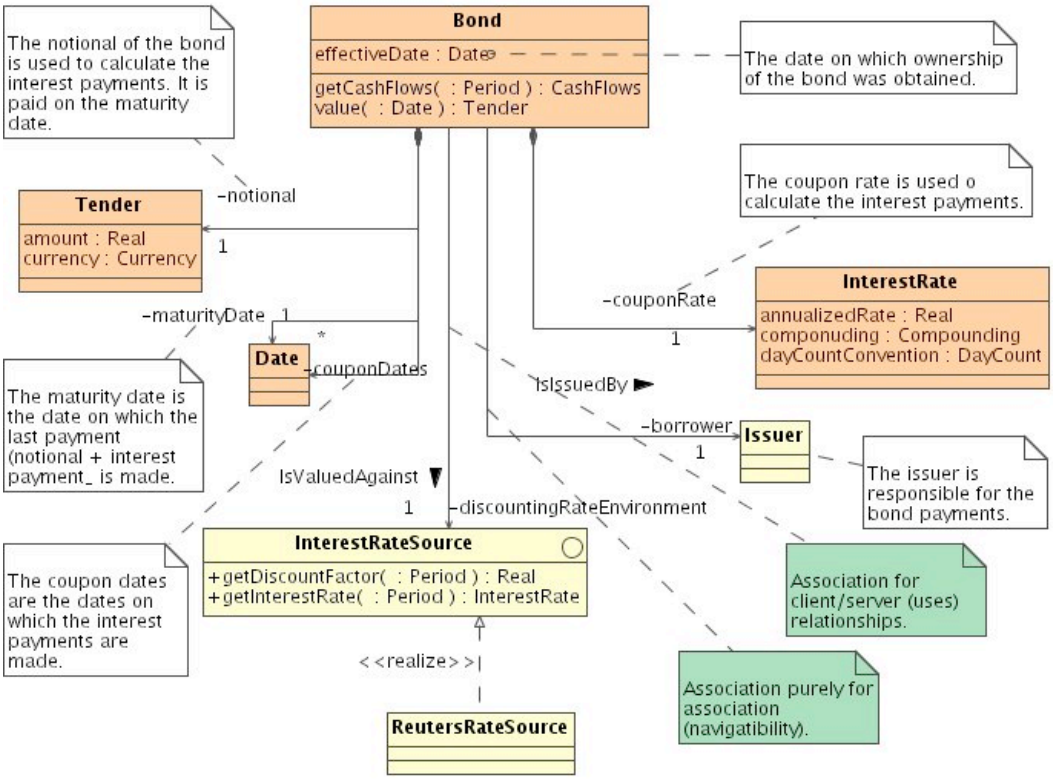
11.4.1. Example: Bonds and interest rate sources

Let us have a look at a second example:

> *“ A bond is issued by an issuer. It has a maturity date, an effective date, a notional, a coupon rate and coupon dates,i.e. dates on which coupons are being paid. To value the bond, its cashflows are discounted against an interest rate environment which can be any interest rate source, i.e. any object which can queried for interest rates (e.g. Reuters). ”*

Going through the motion of identifying the objects, composition and association relationships yields typically something like shown in Figure 4.42, “ Mapping the textual description of a bond onto UML. ”. Note how the variable names map onto role names and note that all elements which are part of the bond contract are shown as components of the bond contract. The interest rate source against which the bond cashflows are discounted, i.e. against which the bond is values, is not part of the contract. This is simply a service provider which is *used* by the bond and hence the relationship is an associatio relationship.

Figure 4.42. Mapping the textual description of a bond onto UML.



11.5. Use interfaces or the server side of associations

We discussed that an association may be viewed as a client/server relationship. Typically clients should not lock into a particular service provider. Instead one wants a situation where you can plug

in any service provider who supplies the required services.

To achieve this the server side of an association should virtually always be an interfaces. this decouples clients fom the actual realizations of service providers and avoids vendor locking.

Furthermore, the client's requirements are clearly documented in an interface. This alone may be reason enough to always insert interfaces between clients and their service providers.

11.5.1. Interfaces in bi-directional associations

One question you might raise is

> *" How does one insert interfaces in a binary association? "*

The solution is actually quite simple. A binary association is anyway just a short hand notation for two unary associations. Extract the unary associations and insert interfaces at the server side of each resultant unary association.

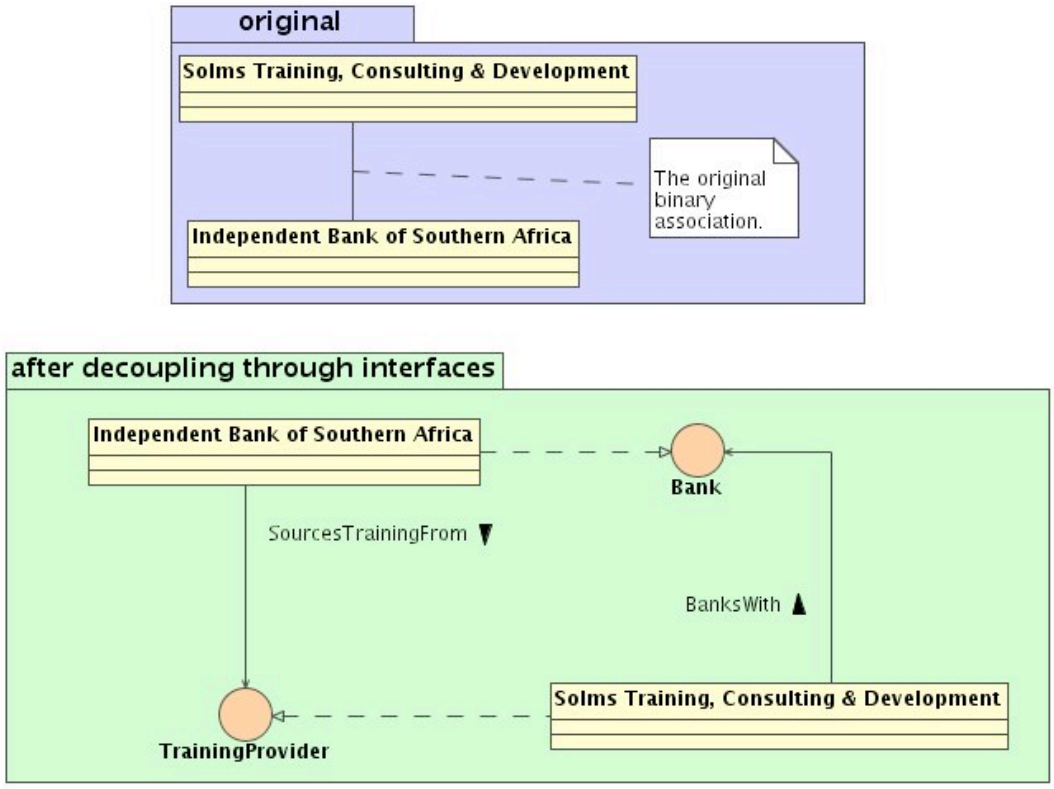
11.5.1.1. Example: Binary relationship between bank and training institution

Let us have a look at an example which demonstrates the benefits of separating a binary association into two unary associations. Take the example of a training institution makin use of the services of a bank which in turn makes use of the training services offered by the training institution.

Separating the binary association into two unary association enables us to

- define the relationships better,
- encapsulate the services they require from each other in two interfaces,
- and use those interfaces to decouple the two organizations so that neither is vendor-locked.

Figure 4.43. Decoupling a binary relationship through two interfaces.



11.6. Association constraints

In the example of the DAPC system we already specified a constraint between the *Controls* and *Monitors* associations, i.e. that the controls links are a subset of the monitors links. Hence we specified that the controlled system is one of the monitored systems.

At this stage we have not yet introduced the *Object Constraint language* (OCL) which will enable us to specify the constraints more formally and reduce the danger of mis-interpretation. The OCL will be covered in Chapter 7, *The Object Constraint Language (OCL)*.

Nevertheless we will look at further constraints between relationships in this section.

Note

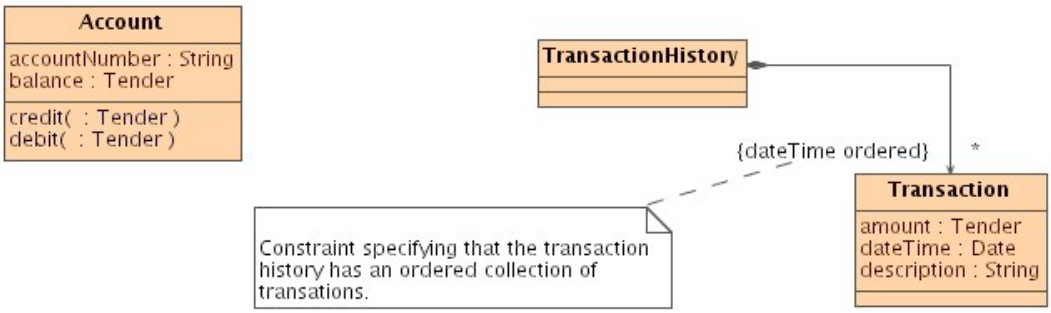
It is often possible to modify the model in such a way that the introduction of an additional constraint is no longer necessary, i.e. that the constraint is automatically satisfied through the model. If this is possible, it usually leads to a cleaner, more natural and more flexible analysis or design solution.

11.6.1. Ordering constraints

If we have a one-to-many or many-to-many relationship (composition, aggregation or association) then instance of one class are associated with a collection of instances of another class. Collections can be ordered or not ordered. By default, collections in UML are not ordered. To specify that a collection is ordered, one places the constraint `{ordered}` on the relevant end of the relationship. in cases where it is not obvious one should specify the attribute on which the ordering is done.

In Figure 4.44, " Using or and xor constraints between relationships. " we specify that a transaction history has an ordered collection of transactions and that the ordering is done on the `dateTime` of the transactions.

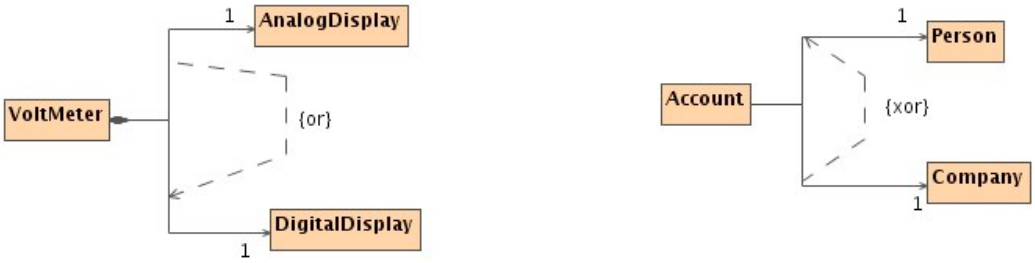
Figure 4.44. Using or and xor constraints between relationships.



11.6.2. Or and xor constraints between relationships

Sometimes a component or a service provider can be one of a few choices. In such cases you might consider to us an `{or}` or an `xor` (exclusive or) constraint between either composition, aggregation or association relationships. An example of this is shown in Figure 4.45, " Specifying an ordered collection via a constraint on the many side of an association or composition relationship. ".

Figure 4.45. Specifying an ordered collection via a constraint on the many side of an association or composition relationship.

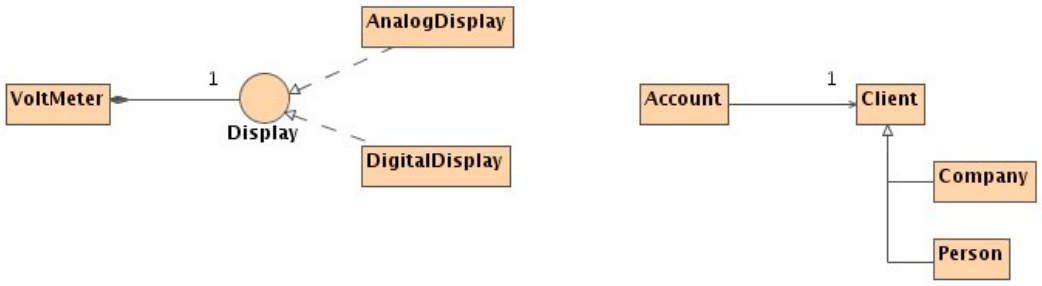


11.6.3. Accommodating or and xor relationships naturally through specialization

Or and xor constraints are conceptually messy and even more messy to implement and to maintain. Having several such constraints in an analysis or design model usually is a sign of a model which can be simplified considerably through working at higher levels of abstraction.

For example, imagine how the `Voltmeter` diagram in Figure 4.45, " Specifying an ordered collection via a constraint on the many side of an association or composition relationship. " scales with increasing number of displays. A direct implementation of such a design would look even worse. However, looking at displays from a higher, more abstract level will lead you to the diagram in Figure 4.46, " Removing or and xor constraints through by introducing more abstract concepts usually leads to a cleaner, more flexible and more scalable model. ".

Figure 4.46. Removing or and xor constraints through by introducing more abstract concepts usually leads to a cleaner, more flexible and more scalable model.



Here we encapsulated the services required by the voltmeter from a display in an interface. The model is not only conceptually cleaner,but can also readily absorb further types of displays.

Similarly, the fact that an account is associated with either a person or a company can be modeled more naturally by introducing the more abstract concept of a client, with persons and companies being different specializations of a client.

11.7. Association classes

We have noted that links are objects and that associations can thus be modeled as classes with stereotype `<<Association>>`. In fact, the solid association line (with or without arrow(s)) can be viewed as a stereotype symbol and the association name as the name of the association class. The standard notation for associations can thus be seen as consistent with the class notation where the class diagram is collapsed into a stereotype symbol.

Since an association can be modeled as a class with links as instances, we can assign attributes and functionalities to them. In this case an association would no longer be simply implemented by a reference or a pointer. We need to define a class for the association.

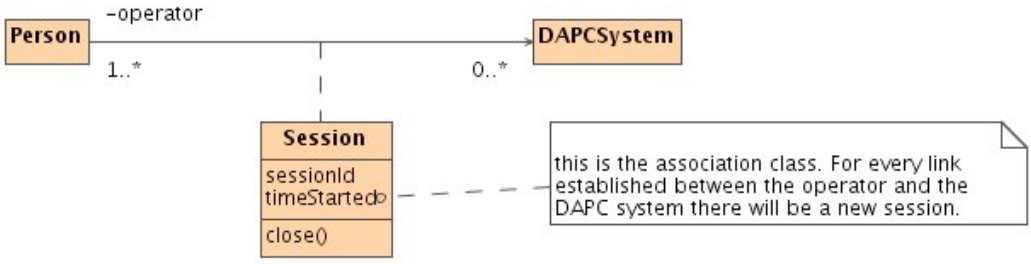
In some cases it is natural to model an association with data attributes, operations (class methods) and sometimes even further associations to other classes. In other words, when we make a link between two objects we obtain another object with attributes and possibly operations. For example, if we associate a `Person` with `DAPCSystem` then the association introduces further attributes (i.e. the session number and the connection time) and further operations (i.e. that the connection can be closed). These new attributes and operations cannot be naturally modeled by encapsulating them in either the client or the server class. Instead we encapsulate them within an association class which is instantiated every time we create a link.

The notation for an association class is a class diagram linked with a dashed line to the association. It is illustrated in Figure 4.47, “ Association classes enable one to assign attributes and services to the association itself. ”.

Note

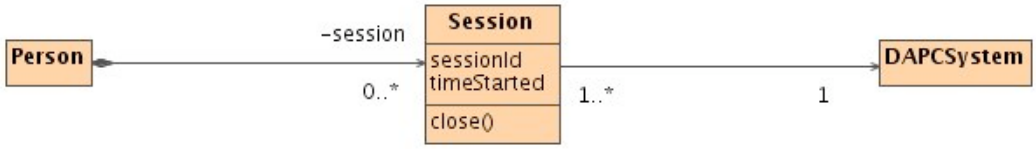
For every instance of the association (every link) there wil be an instance of the association class.

Figure 4.47. Association classes enable one to assign attributes and services to the association itself.



People not very familiar with UML may find the concept of an association class a little difficult. One can, however, map an association class onto more vanilla UML notation (see Figure 4.48, “ Association classes can be mapped onto more vanilla UML notation. ”).

Figure 4.48. Association classes can be mapped onto more vanilla UML notation.

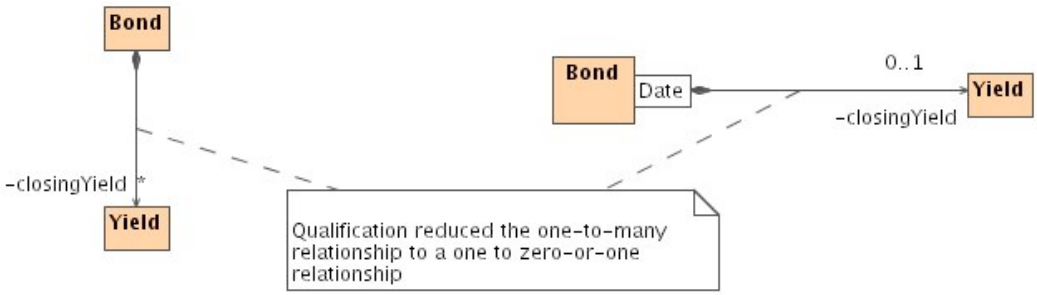


11.8. Qualifications

The unified modeling language also includes support for qualifications of association, aggregation and composition relationships. association. A qualifier is a link attribute which reduces the effective multiplicity of the association. We can thus attach qualifiers to an end of an association which has a multiplicity greater than one.

Let us illustrate the use of qualifiers by an example. For each bond there is a yield quoted at the close of trading, the closing yield. On any given date there is at most one closing yield. There might be none if there was no trading on that date (e.g. on a public holiday). This relationship is shown in Figure 4.49, “ Qualifications reduce the multiplicity of a relationship. ”.

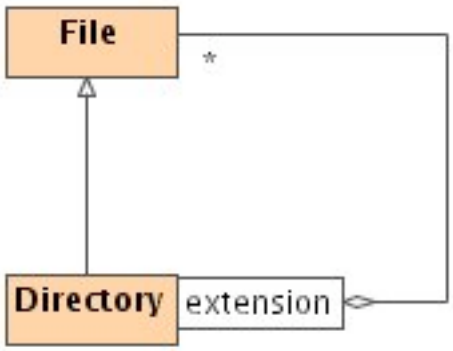
Figure 4.49. Qualifications reduce the multiplicity of a relationship.



In UML qualification is shown as a small rectangle attached to a class in an association which has a non-unity multiplicity in the association. The qualifier might visually give the impression that it belongs to the class, but that is not so. What really is specified is that the client should have a service which returns a subset of its associations.

Qualification reduces the multiplicity of a many-to-one association but it need not reduce the multiplicity to a one-to-one association. For example, giving a directory the qualifier that the extension of a file name is `xml` could resolve all Java files present in the directory.

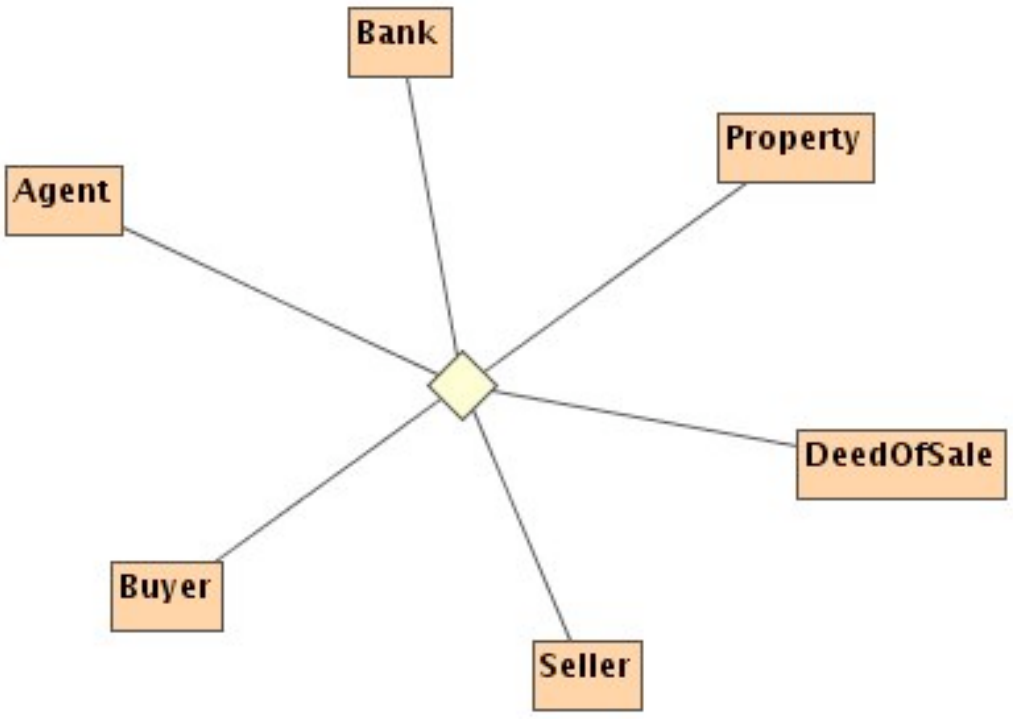
Figure 4.50. Qualifications need not reduce a one-to-many to a one-to-one relationship.



11.9. N-ary associations

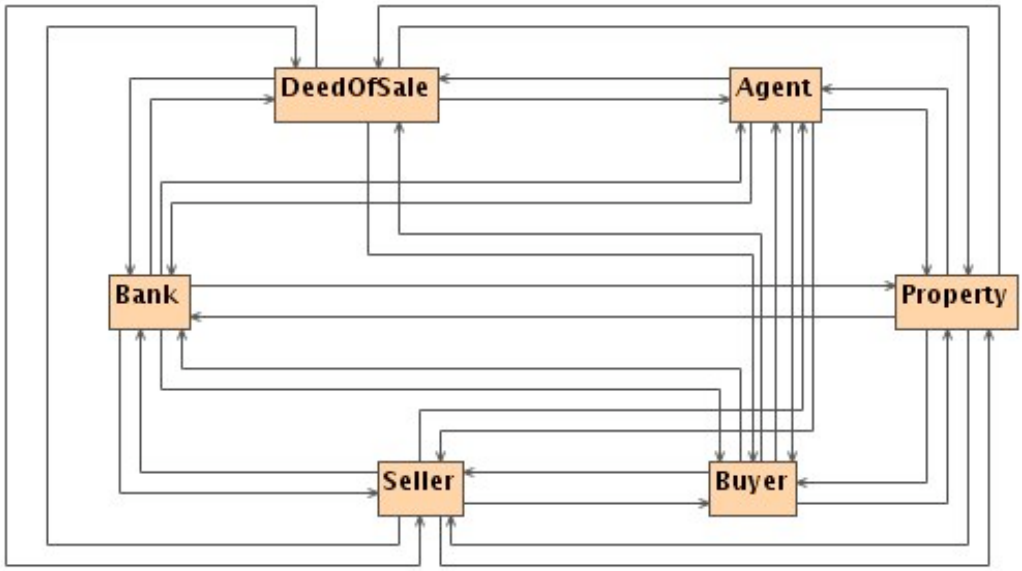
N-ary notations enable you to specify bi-directional message paths between all objects participating in the n-ary association using a single diamond connector between the association lines to the various objects (see Figure 4.51, " UML notation for n-ary associations. ").

Figure 4.51. UML notation for n-ary associations.



N-ary associations are one of the *"sins"* of UML. You many want to consider putting an outright ban on the use of n-ary associations. They leed to excessive complexity which is hidden behind some compact UML notation. Figure 4.52, " N-ary associations provide a compact notation to hide a total mess. " shows the message paths specified in the harmlessly looking diagram of Figure 4.51, " UML notation for n-ary associations. ".

Figure 4.52. N-ary associations provide a compact notation to hide a total mess.

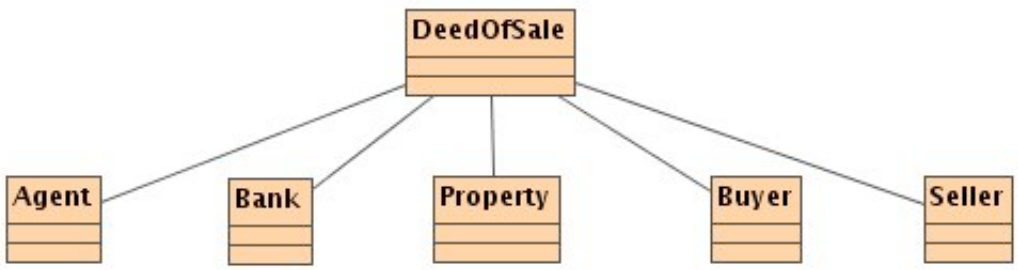


11.10. Simplify N-ary associations by introducing a mediator

If you feel that all the communication paths which you would like to specify in an n-ary association are really required, then you may want to consider using a mediator (one of the classical design patterns).

For example, we may declare the `DeedOfSale` the mediator with binary associations between it and the related objects. Then all communication between these object will go through the mediator (see Figure 4.53, " Simplifying a communication network by introducing a mediator. "

Figure 4.53. Simplifying a communication network by introducing a mediator.



11.11. Avoid spaghetti communication networks

In general one should remain aware of the fact that every association specifies at least one communication path and so do aggregation and composition relationships. We are specifying a communication network.

One of the core dangers is that the resultant communication network is excessively complex and it may be worth to periodically refactor the model and potentially even to apply network optimization techniques to it.

12. Implementing association relationships

Here we show mappings of UML association relationships onto Java, C++ and XML.

12.1. Implementing associations in Java

A unary association is implemented by giving the client a reference to the service provider. If the service provider is decoupled through the use of an interface, the reference is an reference to the interface.

```
import java.util.Date;

public class Cheque
{
  public Cheque(Date date, double amount)
  {
    this.date = (Date)date.clone();
    this.amount = amount;
  }

  public Date getDate() {return (Date)date.clone();}
  public void setDate(Date newDate) {this.date = (Date)newDate.clone();}

  public InterestRateSource getDiscountingRateSource()
  {
    return discountingRateSource;
  }
  public void setDiscountingRateSource
                (InterestRateSource discountingRateSource)
  {
    this.discountingRateSource = discountingRateSource;
  }

  public double getAmount() {return amount;}
  public void setAmount(double newAmount) {amount = newAmount;}

  public Object clone()
  {
    Cheque copy = null;
    try
    {
      copy = (Cheque)super.clone();
      copy.date = (Date)this.date.clone();
    }
    catch (CloneNotSupportedException e)
      {/* never thrown,just removing pre-conditions. */}

    return copy;
  }

  private Date date;
  private double amount;
  private InterestRateSource discountingRateSource;
}
```

You may also want to consider the following guidelines:

- Do not clone (i.e. do not make a copy) of the associated class because then you do not have an association to the real thing, i.e. you will no longer feel the effects of a changing environment.
- Your constructor arguments would typically contain all mandatory components (at least all for which there are no default values), but the association are usually provided via set methods. For example, our cheque may exist without assigning a discounting rate source. We cannot value our cheque then,but we still have the cheque as a legally binding contract.

Binary associations are implemented as two unary associations.

12.2. Implementing association in C++

Association relationships are implemented in C++ in a way which is analogous to the Java implementation except that pointers are used for references. The client should NOT take over any memory management responsibilities of the objects they are using. This should be left to the owners of these objects.

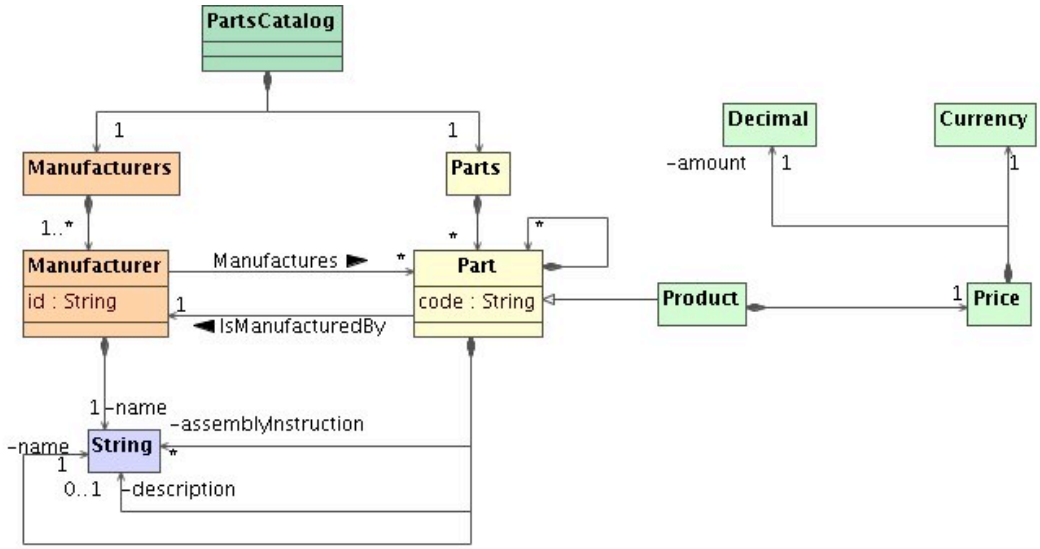
12.3. Implementing associations in XML

Before schemas established themselves as the preferred way of specifying XML document structure, DTDs (Document Type Definitions) were used. In that technology associations were implemented by hosting an `ID` on the destination side of the association and an `IDREF` on the client side.

With the introcution of schemas XML now provides a much more powerful mechnism for specifying associations in XML via keys and key references. The keys need not be unique accross the entire document, but can be scoped within the document. Furthermore, they can be assembled from a combination of elements and/or attributes.

The destination side of an association thus hosts a `key` and the client side hosts a `keyref`.

Figure 4.54. A parts catalog containg composition, specialization and association relationships.



Example 4.10. A parts catalog

The XML mapping of the UML diagram in Figure 4.54, “ A parts catalog containg composition, specialization and association relationships. ” is shown below. Note the mapping of associations onto key/keyref pairs.

```
<xsd:schema targetNamespace="http://www.ManufacturingUnlimited.co.za/PartsCatal
                      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                      xmlns="http://www.ManufacturingUnlimited.co.za/PartsCatalo
                      elementFormDefault="qualified">

  <xsd:element name="partsCatalog">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="parts" type="Parts"/>  <!-- some of which may be pro
        <xsd:element name="manufacturers" type="Manufacturers"/>
      </xsd:sequence>
    </xsd:complexType>

    <xsd:key name="partID">
      <xsd:selector xpath="parts/part"/>
      <xsd:field xpath="@code"/>
    </xsd:key>

    <xsd:key name="manufacturerID">
      <xsd:selector xpath="manufacturers/manufacturer"/>
      <xsd:field xpath="@id"/>
    </xsd:key>

    <xsd:keyref name="manufacturerPartRef" refer="partID">
      <xsd:selector xpath="manufacturers/manufacturer"/>
      <xsd:field xpath="part/@ref"/>
    </xsd:keyref>

    <xsd:keyref name="partManufacturerRef" refer="manufacturerID">
      <xsd:selector xpath="parts/part"/>
      <xsd:field xpath="manufacturer"/>
    </xsd:keyref>
  </xsd:element>
  <xsd:complexType name="Parts">
    <xsd:sequence>
      <xsd:element name="part" type="Part" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="Manufacturers">
    <xsd:sequence>
      <xsd:element name="manufacturer" type="Manufacturer" maxOccurs="unbounded
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="Part">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="description" type="xsd:string" minOccurs="0"/>
      <xsd:element name="assemblyInstruction" type="xsd:string" minOccurs="0" m
      <xsd:element name="part" type="Part" minOccurs="0" maxOccurs="unbounded"/
      <xsd:element name="manufacturer" type="xsd:string"/>
    </xsd:sequence>
    <xsd:attribute name="code" type="xsd:string"/>
  </xsd:complexType>

  <xsd:complexType name="Product">
    <xsd:complexContent>
      <xsd:extension base="Part">
        <xsd:sequence>
          <xsd:element name="price">
            <xsd:complexType>
              <xsd:sequence>
                <xsd:element name="amount" type="xsd:decimal"/>
                <xsd:element name="currency" type="xsd:string"/>
```

```
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="Manufacturer">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="address" type="xsd:string"/>
    <xsd:element name="part" minOccurs="0" maxOccurs="unbounded">
      <xsd:complexType>
        <xsd:attribute name="ref" type="xsd:string"/>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
  <xsd:attribute name="id" type="xsd:string"/>
</xsd:complexType>

</xsd:schema>
```

An example XML document which would be parsed by this schema is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<partsCatalog xmlns="http://www.ManufacturingUnlimited.co.za/PartsCatalog"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.ManufacturingUnlimited.co.za/PartsCatalog
PartsCatalogKeys.xsd">

  <parts>
    <part code="0112">
      <name>PC-Flash</name>
      <description>The ultimate -- at least for this week</description>
      <assemblyInstruction>Plug keyboard in green socket</assemblyInstruction>
      <assemblyInstruction>Plug mouse or rat in purple socket</assemblyInstruct
      <part code="0113">
        <name>GrindAlong Motherboard</name>
        <description>The board your mother wished she had.</description>
        <manufacturer>123</manufacturer>
      </part>
      <manufacturer>123</manufacturer>
    </part>

    <part code="0114">
      <name>MickyMouse</name>
      <manufacturer>123</manufacturer>
    </part>

    <part xsi:type="Product" code="1111">
      <name>Deep Thought</name>
      <description>A model capable of very deep thinking.</description>
      <manufacturer>123</manufacturer>
      <price>
        <amount>231</amount>
        <currency>ZAR</currency>
      </price>
    </part>
  </parts>

  <manufacturers>
    <manufacturer id="123">
      <name>Slap bam</name>
      <address>15 Semble-It Road, Industria</address>
      <part ref="0112"/>
      <part ref="0114"/>
      <part ref="1111"/>
```

```
    </manufacturer>
  </manufacturers>

</partsCatalog>
```

13. Dependencies

A dependency is the weakest and last of the UML static relationships. While association, aggregation, composition and specialization all imply a structural relationship, there may be a dependency between two entities with there being any structural relationship.

13.1. What is a dependency?

A dependency is a relationship which specifies that a change in one entity may affect another entity. It is usually uni-directional so that the reverse statement does not hold.

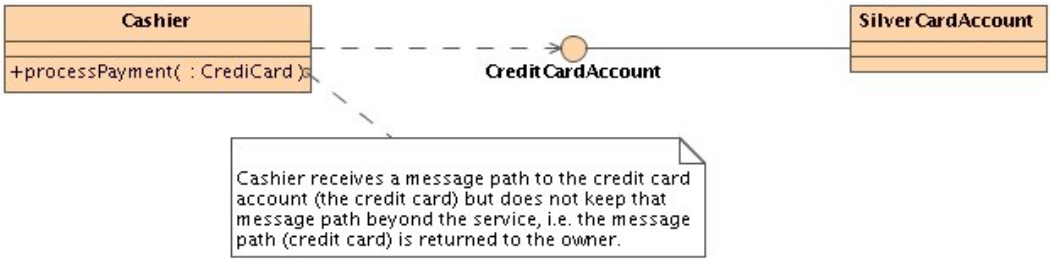
It can be used

- in its own right as a *weak uses* relationship,
- in the context of friendship relationships, or
- as an abstraction of association, aggregation, composition and specialization relationships, i.e. to specify that there is a relationship between two entities without specifying the nature of the relationship.

13.2. UML notation for specifying a dependency

A dependency is specified in UML by drawing a dashed line with an arrow pointing from the class which has the dependency to the class it is dependent upon.

Figure 4.55. Cashiers have a *"weak uses"* relationship with your credit card.



Example 4.11. Cashier has a dependency relationship with your credit card

In order for a cashier to complete his/her work flow, he/she must be able to process a credit card transaction using your credit card as message path to your credit card account. The card is not his/hers (thus the relationship is neither composition nor aggregation) and the teller will (hopefully) not maintain the message path (the credit card) to your credit card account in order to make, at a later stage, further service requests to you credit card account.

13.3. Dependency as a weak uses relationship

Association resembles a *strong uses* relationship where the client maintains a message path to a service provider. This is a structural relationship where the structure provides the message path.

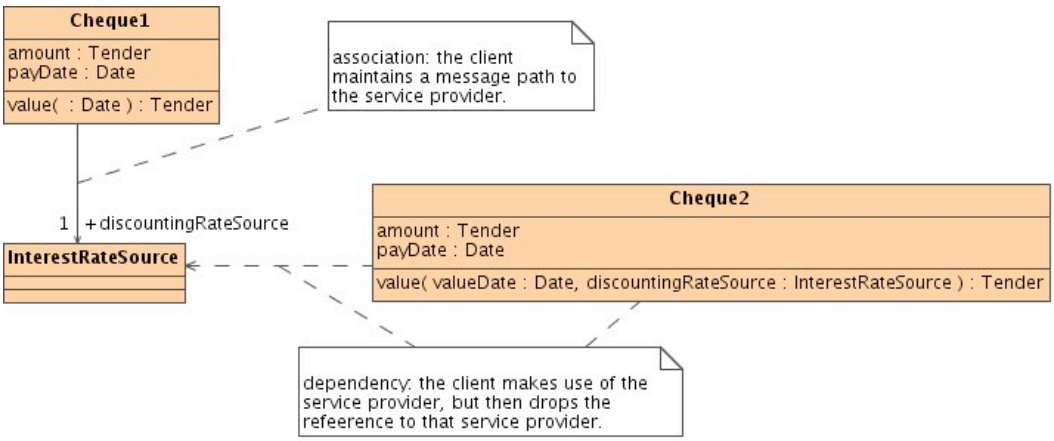
A dependency, on the other hand, may be used to specify a *weak uses* relationship where the client makes use of a service provider, but does not maintain a message path to the service provider.

This is, for example, the case when an operation of one class obtains an instance of another class as argument, makes use of it and then drops the reference to that object. Alternatively a client can create an instance of a class, make use of it and drop the reference to it. That too would be a dependency.

Example 4.12. Cheque has dependency to interest rate source

Figure 4.56, " Association as a strong uses vs dependency as a weak uses relationship. " illustrates the difference between association where the cheque maintains a message path and a dependency where the cheque receives a handle to an interest rate source, uses it for valuation and drops it.

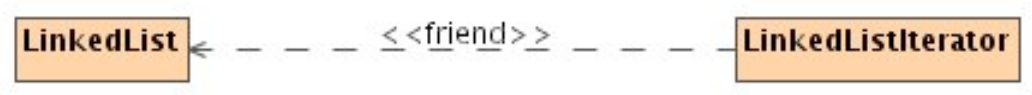
Figure 4.56. Association as a *strong uses* vs dependency as a *weak uses* relationship.



14. Friendship

In UML one class can declare another class a friend, exposing its private members (e.g. private data fields and methods) to the friend. The class who has been declared a friend can thus access the private members of the class from whom it obtained the friendship and through this it has a dependency not only on the public interface, but even on the private implementation details. This is then shown as a dependency with stereotype `<<friend>>`.

Figure 4.57. The LinkedList class declares the LinkedListIterator a friend.



For example, in Figure 4.57, " The LinkedList class declares the LinkedListIterator a friend. ", the

`LinkedList` declares the `LinkedListIterator` a friend, thereby giving it access to its private members (e.g. the *head* node). The iterator is thus dependent on changes to the implementation details of `LinkedList`.

14.1. Implementing frienship in Java

Friendship can be implemented in two ways in Java:

1. By declaring members package scope and placing the two classes within the same package. In this case the friendship will be bi-directions.

2. By making the friend class an inner class. Inner classes have access to the private members of the outer class. This implementation of a friendship relationship is uni-directional and usually the preferred option:

```
public class LinkedList
{
  ...

  public class LinkedListIterator
  {
    ...
  }
  ...
  private Node head;
}
```

There are, however, some consequences to implementing friendship via inner classes. Firstly, instances of inner classes must live within the context of an instance of the outer class. Thus, our linked list iterator would have to live within the context of a particular linked list and would only have access to the private attributes and operations of that particular linked list. In most cases this is exactly what one wants and hence inner classes provide the preferred implementation framework.

14.2. Implementing friendship in C++

C++ has direct support for friendship. One class simply declares another class a friend via the `friend` keyword:

```
class LinkedList
{
  ...
  friend class LinkedListIterator;
  ...
};
```

15. Overview of OO relationships supported in UML

In this section we provide an overview of the UML support for object-oriented relationships between classes and classes and interfaces.

15.1. The 5 relationships between classes

The 5 core relationships between in object-oriented modeling are

- specialization,
- composition,
- aggregation,
- association and
- general dependencies.

These relationships have very precise meaning in object-oriented modeling which we shall review below.

15.1.1. Dependency

The dependency is the weakest relationship where a change in one class (particularly it's public interface) can result in change requirements for the dependent classes.

A dependency relationship is typically direct or indirect uses relationship without there being a client-server relationship maintained. For example, an object which uses another object temporarily in the context of supplying a service is a dependency is dependent on the object whose services it makes use of.

> **Note**
>
> The temporary reference is often obtained by receiving a message path to the object as parameter to a service request. The service makes use of that object, but the object offering the service does not maintain the message path after the service has been provided.

15.1.2. Association

An association relationship is a client-server relationship which is a special type of dependency where one object, the client, maintains a message path (handle) to the other object, the server.

15.1.3. Aggregation

Aggregation is a special form of association where there is still a message path maintained, but where there is a weak has a relationship between the client which is now called the aggregate object and the server which is viewed as a part of the aggregate object.

Aggregation implies state propagation in the sense that a state transition in the component implies a state transition in the owner.

15.1.4. Composition

Composition is a strong has a relationship implying full ownership. It is thus a special form of aggregation were the aggregate object takes full control of the component, i.e.

- The component can only be modified through the owner and
- the component does not survive the owner.

15.1.5. Specialization

Specialization specifies a is a relationship at the instance level. For example, a

- a `CreditCardAccount` is a special type of `Account` and
- a `CorporateClient` is a special type of `Client`.

The two core features of specialization are Substitutability: that we can substitute an instance of a sub-class for an instance of the superclass. For example, if somebody needs an account to, say, subtract some subscription fee,

```
publlic void subtractSubscriptionFee(Account acc)
{
  ...
  acc.debit(fee);
  ...
}
```

we can provide an instance of a `CreditCardAccount` instead and the `debit` service is typically resolved polymorphically. Inheritance: All instance members (attributes and methods) are inherited, though access to some of these may be restricted due to access control specifications (i.e. they have been declared private).

However, specialization can also be seen as a string form of composition where the instance of the subclass has full ownership of an instance of the superclass. A state transition in the superclass instance (the inherited fields) does result in a state tansition for the subclass instance and the superclass instance does not live beyond the subclass instance.

15.2. Relationships between classes and interfaces

UML supports 3 relationships between classes and interfaces, one for service providers and two for clients.

15.2.1. Service providers and interfaces

The only relationship between the service provider role and an interface is the *realization* or *implements* relationship. A service provider may publish the intent of being able to realize the services specified in an interface.

15.2.2. Clients and interfaces

Clients make use of interfaces either via

- an *association* which would imply that the client maintains a message path to a service provider realizing the interface in order to be able to make further service requests to that service provider in the future, or
- a *dependency* which would imply that the client does not maintain a message path to a service provider realizing the interface.

15.3. A Precise Summary of the UML relationships

We have discussed the object-oriented relationships defined in UML. They range from a very strong *"is a"* relationship of specialization to the very *"weak uses"* relationship of a dependency.

In this section we recap their exact meaning and show that the UML relationships are really specializations of each other with a *dependency* being the most general relationship to a *specialization* which is the most specialized.

Figure 4.58, " The core UML relationships are specializations of each other. " shows the relationships between the core object-oriented relationships. It points out that

- **Association is a special form of a dependency.** The client still makes at times use of the ser-

vice provider, but now the client maintains a message path rigrizing the client-server relationship.

- **Aggregation is a special form of association.** The aggregate object still uses its components and maintains a message path to them, but now a state transition is propagated through to the aggregate object (i.e. a state transition in the component implies a state transition in the aggregate object).

- **Composition is a special form of aggregation.** In a composition relationship the owner maintains a message path to its components and a state transition in the component is still propagated through to the owner. But now the owner applies full access control on its components (i.e. the component can only be accessed through its owner). Furthermore, the component does not survive the owner.

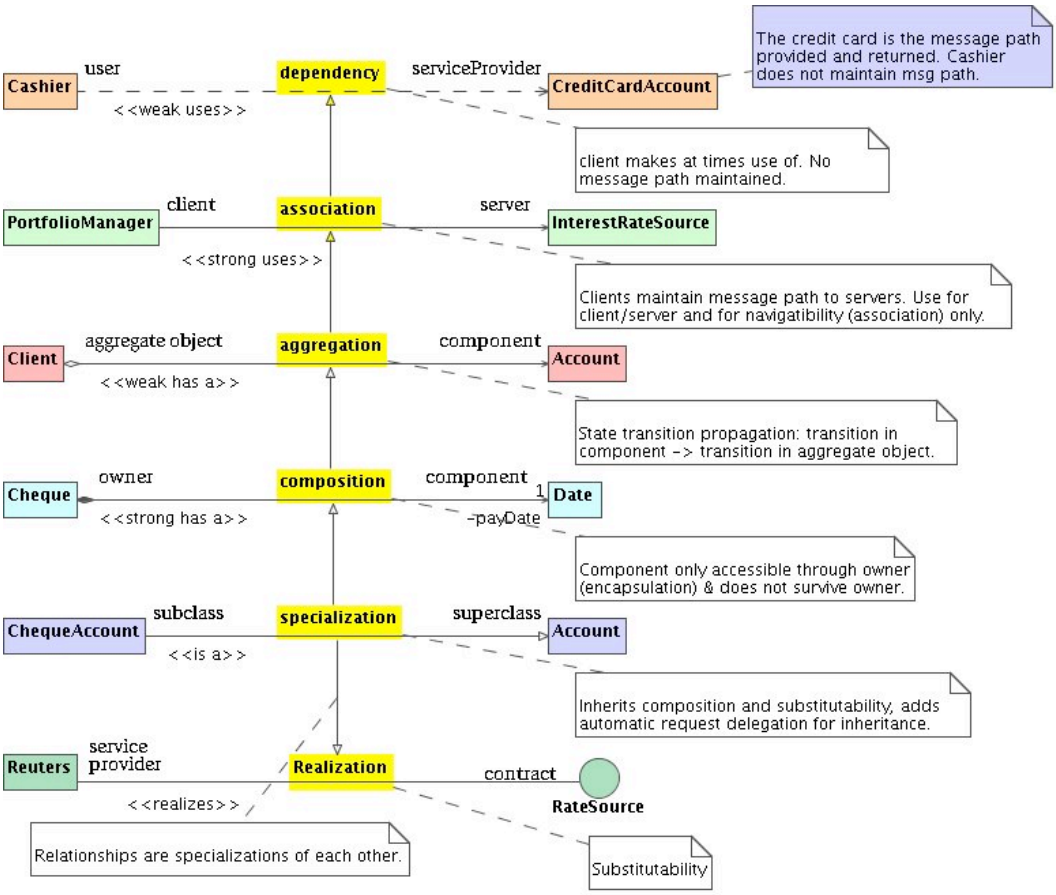
- **Specialization is a special form of composition and realization.** That specialization is a special form of composition may be surprising. Let us test it. The instance of the subclass still maintains a message path to the instance of the superclass it specializes and a state transition in the instance of the superclass it specializes does result in a state transition of the instance of the subclass (if the account balance changes the cheque account it is also changes). Furthermore, the instance of account the cheque account is, can only be accessed through the cheque account and does not survive the latter. Hence specialization is indeed a special form of composition. But what does it add? It adds inheritance of the services of the superclass and substitutability with instances of the superclass -- this is then also what is sacrificed if one maps specialization onto composition.

  ## Note

  Note that specialization is of course also a special form of realization.

- **Realization.** Here a class is a special realization of a contract (interface or abstract superclass).

Figure 4.58. The core UML relationships are specializations of each other.



15.3.1. Shopping for relationships

Figure 4.58, “ The core UML relationships are specializations of each other. ” can be used as a shopping list when you go out and “buy” relationships between classes. Look at what is required. Does the client have to maintain a message path? If yes, then you require at least an association relationship. If not, then a dependency may surfice (or perhaps there is no need to relate them at all).

Next you can check whether a state transition in the server should imply a state transition in the client. If so, you need to upgrade from an association to an aggregation. Otherwise an association may surfice.

Next you can ask yourself whether the aggregate object needs to take full control of the compoenent or whether other objects should be allowed to access the component directly. You can also question whether it makes sense if the component survives the owner. If the answer to these questions is yes, stick with aggregation. If not, you may need at least a composition relationship.

Finally you ask yourself whether you need substitutability. If so you'll have to upgrade the somposition relationship to specialization and you will get inheritance as a bonus. If not, stick with composition.

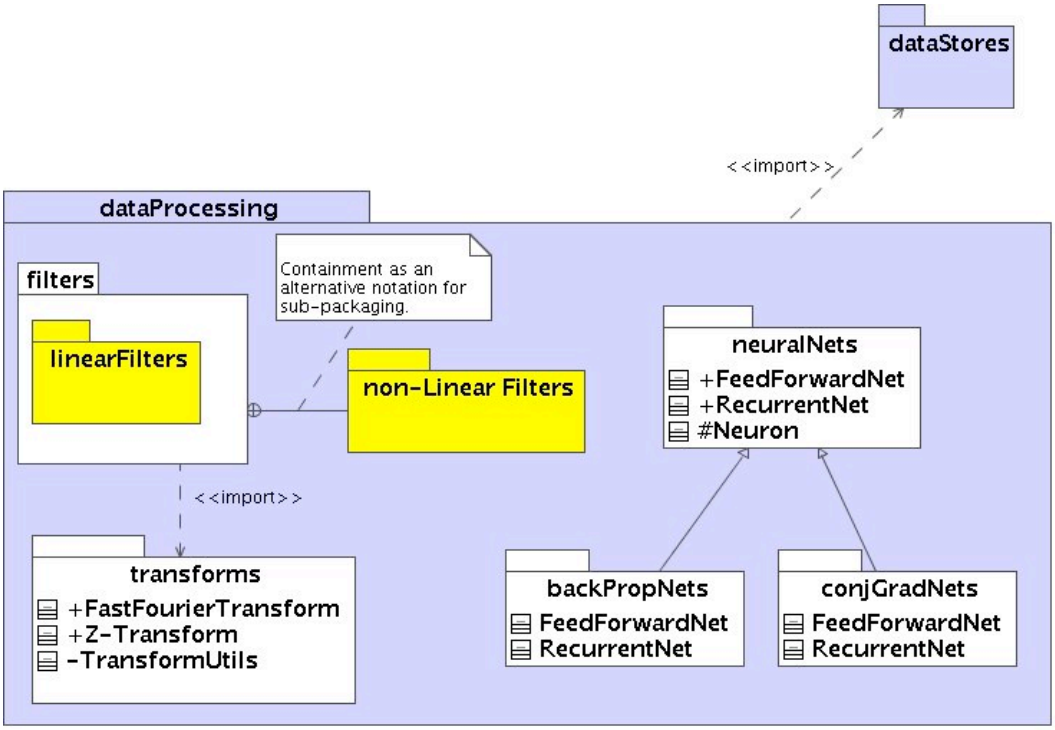
16. Packaging

A package in UML is a grouping of UML model elements. These can include class diagrams and object diagrams, further packages as well as other UML model elements like Use Cases or Sequence Diagrams. Packages can themselves be packaged into higher level packages. Elements contained in a package may either be visible only from within the package, or may be externally accessible (i.e. from within other packages).

16.1. UML notation for packaging

A package in UML is shown as rectangle with a tab, not unlike certain filing cards (see Figure 4.59, “ The relationships between packages are dependency, sub-packaging (nesting of packages) and package specialization. ”). The package name may be supplied either on the tab or in the body of the package rectangle.

Figure 4.59. The relationships between packages are dependency, sub-packaging (nesting of packages) and package specialization.



16.2. Exported and internal elements of a package

The elements of a package are specified in the same way we specify elements of a class, either as attributes or using the composition relationship. Public elements are exported from the package while private elements are internal elements which are only accessible from within the package. If no visibility s defined for an element it is assumed to be public.

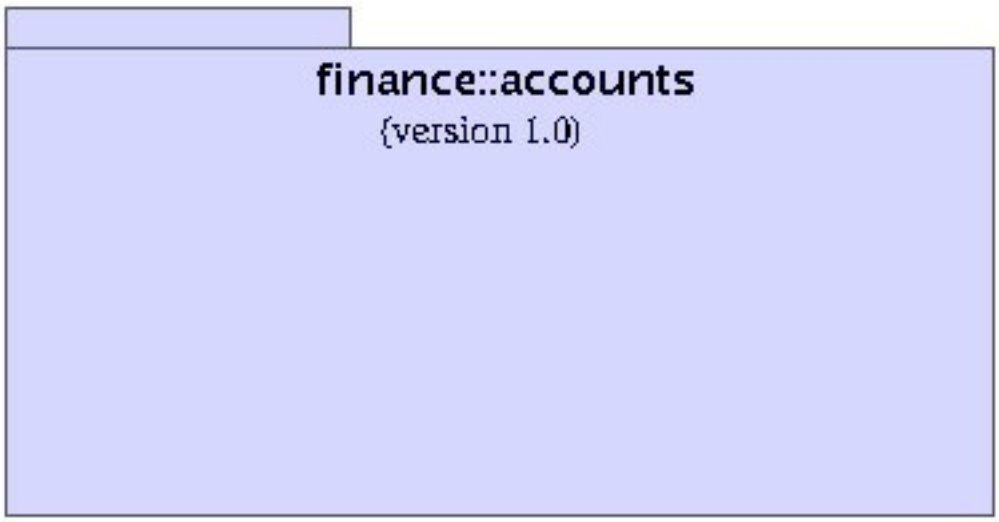
For example, `FourierTransform` and `Z-Transforms` in Figure 4.59, “ The relationships between packages are dependency, sub-packaging (nesting of packages) and package specialization. ” are exported from the `transforms` package, while `TransformUtils` is a class which is internal to the package.

16.3. Nested packages

A package can contain further sub packages. This is specified in UML in one several ways. Firstly the component package can be drawn inside the body of the container package. For example, in Figure 4.59, “ The relationships between packages are dependency, sub-packaging (nesting of packages) and package specialization. ”, the packages `filters` and `neuraNets` are components of the package `dataProcessing`.

Figure 4.60. Nested packages may also be specified using the scope resolution

operator, ::.



A nested package can also be denoted in UML without showing its container package. A scope resolution operator similar to that used in C++ is used. The UML diagram in Figure 4.60, " Nested packages may also be specified using the scope resolution operator, ::. " shows this notation and also demonstrates how version information can be supplied for a package. This is particularly relevant when one wants to show that a system (or package or class) depends on a particular version of another package.

16.4. Importing packages

A package or a class can import an entire package or extract an exported element (interface, class or object) from a package. The importing results in a dependency and is graphically depicted via a dependency relationship with an optional `<<imports>>` stereotype.

For example, in figure Figure 4.59, " The relationships between packages are dependency, sub-packaging (nesting of packages) and package specialization. ", the package `filters` imports `transforms` and the package `transforms` and the package `dataProcessing` imports the package `dataStores`.

16.4.1. Importing is transitive

Importing is transitive in the sense that if package `a` imports package `b` which in turn imports package `c`, then `a` imports `c` too. Exporting, on the other hand is not transitive.

16.5. Package specialization

Package specialization is in many ways similar to class secialization. The sub-package inherits all the elements of the more general super-package. It can add further elements and it can replace inherited elements with ts own version of those elements.

Note

The substitution principle also applies to packages, i.e. a generalized package can be replaced by any of the more specialized packages.

For example, the `backPropagationNets` and `conjGradNets` packages in Figure 4.59, " The relationships between packages are dependency, sub-packaging (nesting of packages) and package

specialization. ” are specializations of the `neuralNets` package. They replace the `FeedForwardNet` and `RecurrentNet` with their own implementations where either the backpropagation or the conjugate gradients optimization method is used for the learning process. The `Neuron` is inherited by both packages, but, as it is defined with protected visibility, it is not exported for public use by either package.

16.6. Package stereoTypes

UML defines 5 standard stereotypes for packges:

- `<<system>>` represents a package containing the entire system being modeled.
- `<<subsystem>>` represents a package containing a subsystem modeled.
- `<<stub>>` represents a package that contains a proxy for the public contents of another system. Stub packages enable one to package the interface to a system separately so that another team can use the stub while the system is being developed. A stub package can be developed early and represents the client view of the system.
- `<<facade>>` represents a package which provides a facade to another package, often publishing the elements in a more intuitive or simpler form. A facade is one of the classical design patterns.
- `<<framework>>` is a package for a class library which facilitates developing code for a particular domain. Examples include user-interface frameworks, persistence frameworks and communication frameworks.

16.7. How to group elements into packages

Any elements which may be re-used should be packaged in a package structure representing a hierarchical tree structure of domains which are addressed by that element.

For example, a numerical root solver, `NewtonRaphsonRootSolver` using the Newton-Raphson method for finding roots of a multi-dimensional function could be packaged in

```
math::numeric::rootSolvers::multiDimensional
```

and its fully quialified name would be

```
math::numeric::rootSolvers::multiDimensional::NewtonRaphsonRootSolver
```

In order to make elements globally unique and facilitate re-use accross organizations one usually inserts the inverted domain name in front of the conceptual package structure:

```
za::co::solmstraining::math::numeric::rootsolvers::multidimensional::NewtonRaph
```

Any elements which should explicitly NOT be re-used should be grouped into a package for the system they are exclusively meant for.

17. Implementing packaging

Here we show mappings of UML packages onto Java, C++ and XML.

17.1. Implementing packages in Java

UML packages map virtually directly onto Java's package support. A UML package maps onto a Java package which maps onto a directory if the file system is used as source repository. Public, and private elements of a package would map onto Java package elements (e.g. classes and interfaces) declared with public and package (friendly) scope respectively. The latter is the default visibility level in Java.

In Java a dot is used for the scope resolution operator. Elements of a file are assigned to a package by inserting a package statement at the top of the file. For example, the `FastFourierTransform` class shown in Figure 4.59, " The relationships between packages are dependency, sub-packaging (nesting of packages) and package specialization. " would be defined in Java via

```
package dataProcessing.transforms;

public class FastFourierTransform { ... }
```

and its fully qualified name would be

```
dataProcessing.transforms.FastFourierTransform
```

Package specialization is not supported in Java. To map package specialization onto Java one could first define a stub package for the package interface. The super and sub package classes would then both implement the interfaces specified in the stub.

One can import a single class from a package via

```
import dataProcessing.transforms.FastFourierTransform;
```

or the entire package (all its elements) via

```
import dataProcessing.transforms.*;
```

17.2. Implementing packages in C++

C++ uses namespaces for packaging. Namespaces can be nested and the scope resolution operator is the same as in UML.

For example, the `FastFourierTransform` class shown in Figure 4.59, " The relationships between packages are dependency, sub-packaging (nesting of packages) and package specialization. " would be defined in C++ via

```
namespace dataProcessing
{
  namespace transforms
  {
    class FastFourierTransform { ... };
  }
}
```

and its fully qualified name would be

```
dataProcessing::transforms::FastFourierTransform
```

Package specialization is not supported in C++. To map package specialization onto C++ one could first define a stub package for the package interface. The super and sub package classes would then both implement the interfaces specified in the stub.

One can import the elements of a package via

```
using namespace dataProcessing::transforms;
```

17.3. Implementing packages in XML

UML packages map XML namespaces. To assign elements of a file to a namespace one uses the `targetNamespace` attribute of the schema element.

To access elements from a package one imports the namespace either into the default prefix (no prefix: the elements can then be accessd directly) or into a particular prefix through which the elements will be accessed.

For example, below we define elements for a package

```
http://www.ManufacturingUnlimited.co.za/PartsCatalog
```

and import those elements directly into the `parts` prefix.

```
<xsd:schema targetNamespace="http://www.ManufacturingUnlimited.co.za/PartsCatal
       xmlns:xsd="http://www.w3.org/2001/XMLSchema"
       xmlns:parts="http://www.ManufacturingUnlimited.co.za/PartsCatalog"
       elementFormDefault="qualified">

  <xsd:complexType name="PartsCatalog">
    ...
  </xsd:complexType>

  <xsd:element name="partsCatalog" type="parts:PartsCatalog"/>
  ...
</xsd:schema>
```

Below we have an XML instance document which imports the parts-catalog namespace into the default prefix. The elements are then used directly without referring to them through a prefix:

```
<?xml version="1.0" encoding="UTF-8"?>
<partsCatalog xmlns="http://www.ManufacturingUnlimited.co.za/PartsCatalog"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.ManufacturingUnlimited.co.za/PartsCatalog
PartsCatalogKeys.xsd">

  <parts>
    <part code="0112">
      <name>PC-Flash</name>
      <description>The ultimate -- at least for this week</description>
      <assemblyInstruction>Plug keyboard in green socket</assemblyInstruction>
      <assemblyInstruction>Plug mouse or rat in purple socket</assemblyInstruct
      <part code="0113">
        <name>GrindAlong Motherboard</name>
```

```
        <description>The board your mother wished she had.</description>
        <manufacturer>123</manufacturer>
      </part>
      <manufacturer>123</manufacturer>
    </part>
...
  </parts>
</partsCatalog>
```

18. Metaclasses

<xi:include></xi:include>

18.1. What is a metaclass?

Recall that an object is an instance of a class. In a similar way one could go to a further level of abstraction and view classes themselves as instances of other classes -- so called metaclasses. A *metaclass* is thus a class whose instances are themselves classes. The attributes and operations of a metaclass hence become class attributes and class services. Some programming languages (notable *Smalltalk* and *CLOS*) support metaclasses as a language construct. These languages use metaclasses to define class attributes and class services.

A class operation is a service offered by the class itself. Hence class services are requested by sending a message to the class, not to instances of the class.

18.2. Constructors

Many implementation languages support the concept of a constructor which is responsible for creating instances of the class.

From a user perspective constructors are class services. You requesting the class to create an instance for you (how, otherwise would you create the first instance, if it was an instance, i.e. non-static, service).

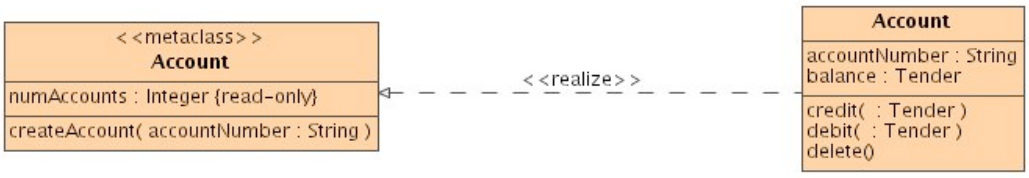
However, from an implementation perspective the constructor body is an instance service initializing the instance itself. Effectively, a constructor is a package containing

1. a class service (static function) which creates an instance of the class, and
2. an instance service initializing the instance itself (the constructor body).

18.3. UML notation for metaclasses

The instances of metaclasses are classes. In UML one uses a class diagram with stereotype `<<metaclass>>` to represent a metaclass. The diagram in Figure 4.61, “ A metaclass defining the class (static) members of an Account class. ” shows a metaclass for `Account` specifying the class service for instance creation. This is a class service since you are not going to request one account to create you another account (how would you create the first account?). Object creation or instantiation is thus provided through a service offered by the class itself (constructors are implicitly static methods). The metaclass also defines a class attribute keeping track of the number of instances of the class.

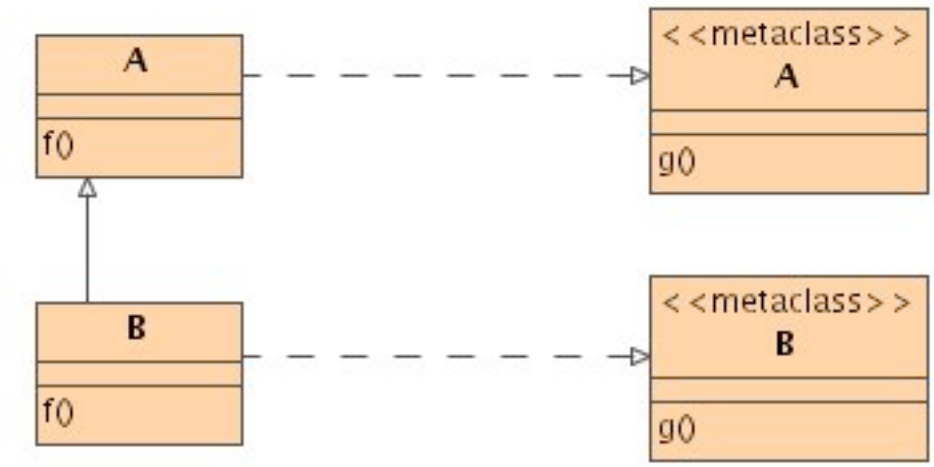
Figure 4.61. A metaclass defining the class (static) members of an Account class.



18.4. Class services are not resolved polymorphically

The operations defined in a metaclass are operations (services) offered by the class itself and not by instances of a class. Even though many programming languages like Java and C++ do allow class services to be requested from their corresponding instances, these class services are typically bound (linked) at compile-time.

Figure 4.62. Class services are not resolved polymorphically.



For example, Figure 4.62, " Class services are not resolved polymorphically. " specifies a class `A` with service `f` and a subclass `B` which overrides that service with its own implementation. Here `f` is an instance service (offered by instances of the classes `A` and `B`) which will be resolved polymorphically, i.e.

```
A a = new B();
a.f();
```

will resolve the instance service as specified in the class `B`.

Figure 4.62, " Class services are not resolved polymorphically. " also shows the corresponding metaclasses which provide a service `g`. The service `g` is thus a class (static) service. That service will, however, not be resolved polymorphically, i.e.

```
A a = new B();
a.g();
```

will not resolve the service `g` as specified in the metaclass `B`, but instead the one specified in `A`, even though the reference `a` actually refers to a `B`.

19. Implementing meta-classes

Here we show mappings of composition relationships onto Java, C++ and XML.

19.1. Implementing metaclasses in Java

Java does not support meta-classes directly. Instead it provides `static` attributes and services which are class attributes and services. For instantiating constructors are used which are implicitly class services (static). For example, the diagram in Section 18, “Metaclasses” would be implemented as follows:

```
public class Account
{
  public Account(String accountNumber)  /* constructors are implicitly static *
  {
    ...
    ++numAccounts;
  }

  public static int getNumAccounts() {return numAccounts;}

  public void finalize() {--Account.numAccounts;}

  private static int numAccounts = 0;
  ...
}
```

All the methods shown, except the `finalize` method, are actually class services (i.e. `static`). The constructor is implicitly so. The class attribute, `numInstances`, is initialized upon loading the class. Each instance of the class has, in Java, access to the class attributes and services. In the `finalize` method, which is called by Java's garbage collector before an instance is released from memory, the counter for the number of instances is decremented.

19.2. Implementing metaclasses in C++

The C++ implementation is similar to the Java implementation:

```
class Account
{
  public:
    Account(const String& name, const Account& account)
    {
      ...
      ++numAccounts;
    }

   virtual ~Account() {--Account::numAccounts;}

   static int getNumAccounts() {return numAccounts;}

  private:
    static int numAccounts;
  ...
}

Account::numAccounts = 0;
```

Note that the destructor `~Client()` is used instead of Java's `finalize` and that the class variable, `numInstances`, must be initialized at global scope.

20. Inner classes

The benefits of using inner classes is appreciated more and more. They have found particular favour in the Java and XML communities where many developers often introduce as many inner classes as stand-alone classes.

20.1. What is an inner class?

Inner classes are classes whose instances exit only within the context of instances of other classes. It is thus a class whose instances are packaged within instances of the outer class and have access to the private instance members of the outer class.

20.2. Why use inner classes?

The core reasons for using inner classes are scoping and encapsulation.

20.3. Specifying inner classes in UML

An inner class is specified in UML using the containment symbol.

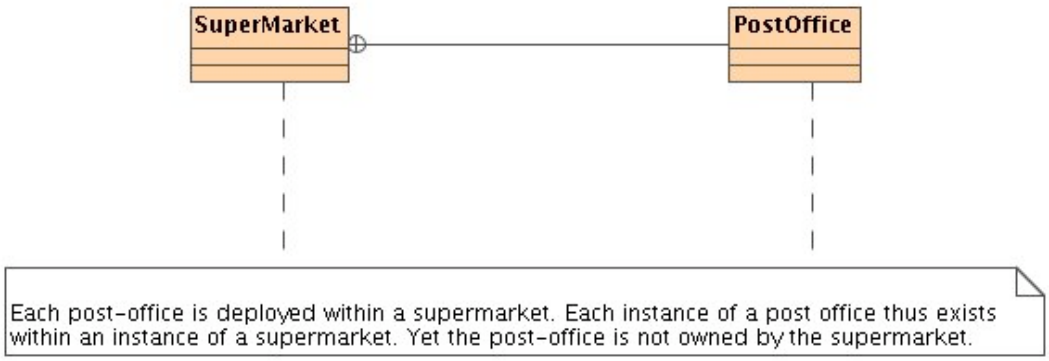
20.3.1. Example: Embedded service provider

As an example from business modeling, consider a business model where post-offices are not deployed as stand-alone service providers, but where each post-office is embedded within a supermarket. In this case

- Each instance of a post office exists within an instance of a supermarket.
- The post office is not owned by the supermarket not composition).
- The state of the post office is not part of the state of the supermarket.
- The supermarket does not use the post-office as a service provider.

Instead, this is a form of packaging instances of one class within an intsance of another.

Figure 4.63. Post-offices being packaged with supermarkets



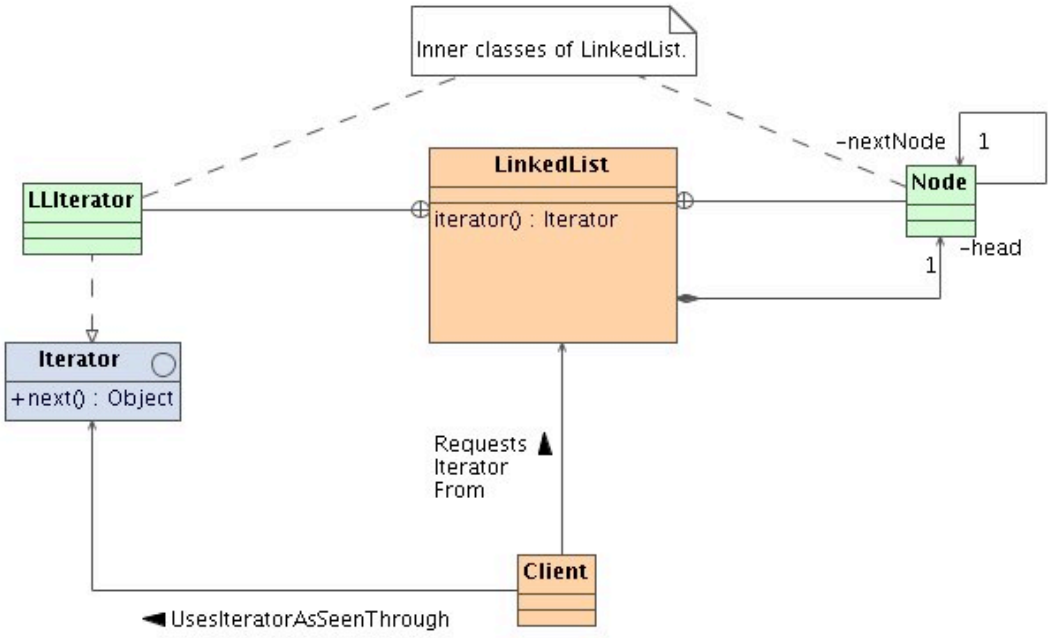
Potentially the post-office may have access to private components of the supermarket. For example, the post-office could use the rest-rooms reserved for staff of the supermarket.

20.3.2. Example: Iterators and nodes as inner classes

Iterators resemble a form of a pointer which provides access to the collection's elements and supports iterating across these. The iterator for a particular collection is tightly bound to that collection.

For example, the iterator for a linked list needs access to the nodes of the list and in particular to the head node of the list. The existence of the iterator makes sense only in the context of the esistence of the collection itself. Each instance of the iterator thus exists within an instance of the linked list and has access to its private members.

Figure 4.64. Iterator and Node defined as inner classes



Even though the iterator class would be defined as a private inner class, clients may request an iterator for a collection receiving a message path (reference) to an iterator which does not expose the private class itself, but only provides access to it via the `Iterator` interface.

21. Template types

Template types are also known as *parametrized types* or as *generic types*. Specific classes are generated from template types by binding the parameters of the template class. In this way a template type defines a whole family of classes. The parameters passed to a template in order for it to generate a class can be other classes, constant objects or operations.

21.1. When should you consider using a template type?

Template types are particularly useful if several classes or data types share common algorithms (operation bodies). For example, sorting algorithms are largely independent of the data types they are sorting. Similarly, linear algebra operations like matrix multiplication, vector addition and dot product etc. are also largely independent of the data type of the elements, i.e. whether it is a vector of real or complex numbers.

21.2. Vectors

A vector is a special kind of array used in mathematical modeling for which there are certain mathematical operations defined. For example, we can add two vectors, take the dot product of two vec-

tors or calculate the norm of a vector. These operations do not, in general, make sense for arrays.

However, we do not want to define a whole collection of vector classes, one for each data type to be stored in a vector, i.e. we do not want to define separate classes for vectors of integers, vectors of floating point numbers, vectors of complex numbers and say vectors of rational numbers. Instead, we want to define a single vector template which receives the data type of the vector elements and the length (dimensionality) as parameters. Classes are created from the template by *binding it to other classes, integral constants or even operations*. In the case of our vector template we would bind the vector template with a class for the vector elements and an integer constant representing the dimensionality of the vector. For example, if we wanted to generate the class `VectorOfRational` from a `Vector` template, we would bind the template to the class `Rational`.

A template type thus defines a family of classes, each class being specified by binding the parameters of the parameterized type to specific classes, constants or operators. Template types are particularly useful when we have a group of related classes with a very high degree of structural and functional similarity.

Parametrized types reduce duplication of design structures and make it easier to achieve consistency in the family of related classes. They also reduce the maintenance burden, i.e. only the parametrized type has to be changed and the whole family of classes is consistently redefined. Programming languages which support templates (e.g.\ C++) will automatically generate the required classes from the template, i.e. developers only have to modify the template class and the compilation step will generate the classes generated by binding the template arguments.

21.3. UML notation for template types

In UML a template class is depicted by a class diagram which has a parameter block in the top right hand corner of the class diagram. The parameter block defines the elements which must be bound to the template in order to generate a class.

Figure 4.65. A vector template and various classes generated from it.

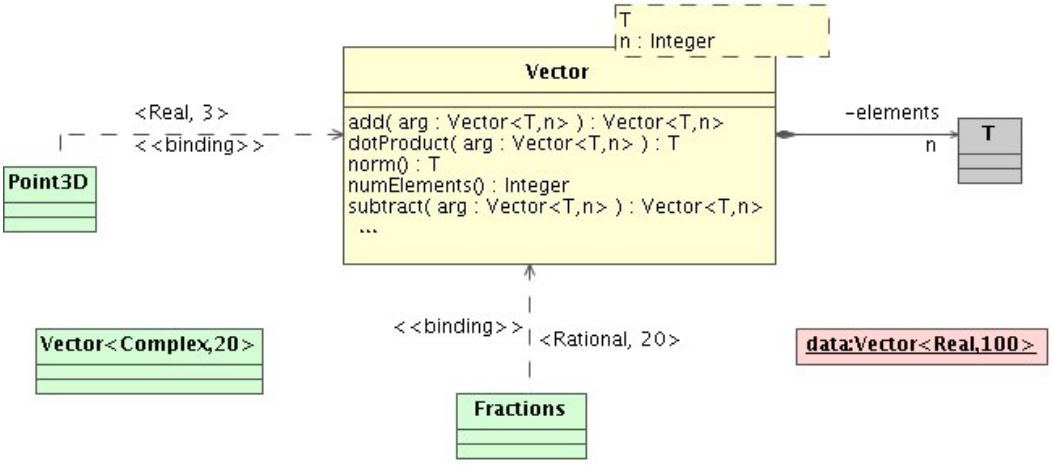


Figure 4.65, " A vector template and various classes generated from it. " shows a vector template with a number of classes generated from it.

The dashed rectangle in the top right corner specifies that one has to bind the `Vector` template to a class `T` and an integer constant `n` in order to generate one of the family of related classes specified by the parametrized type. We then use this template to generate the class `Point3D` by binding it to the class `Real` and the integer constant 3 For this purpose we use a binding dependency, i.e. a with the stereotype `<<binding>>` and the template arguments.

Alternatively we can define a specific class by a class diagram with the template name and the tem-

plate arguments in the name field of the class diagram. For example

```
Vector<Complex,20>
```

specifies the class generated from the template by binding it to the data type `Complex` and the integer constant, `n`, to 20, i.e. it defines a vector of 20 complex numbers.

To define an instance of one of the classes generated by the template we can use an object diagram like the one shown in the bottom right-hand corner of Figure 4.65, " A vector template and various classes generated from it. ". Here `data` is a vector of 100 real (floating point) numbers.

21.4. Implementing template types

Several languages including C++ support template or parametrized types. In such languages one only has to define the template and request the compiler to generate any of the family of related classes specified by the template. This can reduce code duplication significantly and hence the size of the code that has to be maintained. Furthermore, templates ensure that the various classes in the family of template classes are consistent.

21.4.1. Implementing template types in Java

Java approximates templates via generics. However, this is a language construct used to provide type-safe access to generic types without the template type existing at run-time. This introduces a number of significant limitations. ain particular, one is not able to directly instantiate the template type (though there are work arounds using factories and interfaces).

21.4.1.1. A generic resource pool

As an example,let us look at a resource pool where each resource component should at any time be used by only one user. This could be a pool of database connections, a pool of connections to messaging services or a pool used by JCA adapters for connecting to servers supporting proprietary protocols, or simply object or thread pools.

However, we do not want to define pool classes for all different types of resources. We also want to have the pool type safe, i.e. that we can only put the correct type of resource objects into a particular pool. This removes the necessity for type validation and casting of the objects received from the pool.

The pool will have to create new instances of the template type upon increasing the pool size. Since we cannot use constructors to create instances of the parameter types, we have to look at cloning, effectively using the prototype pattern.

> **Note**
>
> To this end developers of the resource class must make certain that the clone method is implemented in such a way that it creates new, functionally equivalent resources. For example, the clone of a database connection must be a separate connection through which database queries can be made.

21.4.1.1.1. The prototype interface

```
package za.co.solms.utils.resource.pooling;

public interface Prototype<T> extends Cloneable
{
  public T clone();
}
```

21.4.1.1.2. The resource pool contract

```
package za.co.solms.utils.resource.pooling;

/**
  * Contract for resource pools (e.g. connection or thread pools).
  * The pool must be initialized with a prototype and an initial
  * pool size before the pool can be used.
  */
public interface ResourcePool<R extends Prototype<R>>
{
  /**
   * Provides and locks a handle to a particular resource.
   */
  public R getResource();

  /**
   * Returns the size of the pool.
   */
  public int getSize();

  /**
   * Returns the current number of available resources.
   */
  public int getNumFreeResources();

  /**
   * Releases the provided resource back to the pool It is the
   * user's responsibility to drop all references to that instance
   * directly after having released it.
   */
  public void releaseResource(R resource);

  /**
   * Resizes the pool to contain the specified number of instances.
   * Pool sizes are increased by cloning the prototype.
   */
  public void resize(int numResourceInstances);

  /**
   * Initializes the pool for use by providing the prototype which
   * is cloned when increasng the pool size.
   * The pool is initialized to the specified pool size.
   */
  public void init(R resource, int initialPoolSize);
}
```

21.4.1.1.3. An implementation for a basic resource pool

```
package za.co.solms.utils.resource.pooling;

import java.util.Set;
import java.util.HashSet;
import java.util.Iterator;

public class BasicResourcePool<R extends Prototype<R>>
             implements ResourcePool<R>
{
  public R getResource()
  {
    if (prototype == null)
      throw new IllegalStateException
                          ("Pool not yet initialized with prototype.");

    boolean resourceAvailable = false;
    synchronized (availableResources)
    {
      // Spin lock for extracting resource
```

```
    while (!resourceAvailable)
    {
      resourceAvailable = true;
      if (availableResources.isEmpty())
      {
        try
        {
          availableResources.wait();
        }
        catch (InterruptedException e)
        {
          if (availableResources.isEmpty())
            resourceAvailable = false;
          else
            resourceAvailable = true;
        }
      }
    }
    Iterator<R> iter = availableResources.iterator();
    R resource = iter.next();
    availableResources.remove(resource);
    return resource;
  }
}

public int getSize() {return resources.size();}

public int getNumFreeResources()
{
  return availableResources.size();
}

public void releaseResource(R resource)
{
  if (resources.contains(resource))
  {
    synchronized(availableResources)
    {
      availableResources.add(resource);
      availableResources.notify();
    }
  }
  else
    throw new IllegalArgumentException
                      ("Resource instance not in pool.");
}

public void resize(int numResourcesInPool)
{
  if (numResourcesInPool < 1)
    throw new IllegalArgumentException
                       ("Pool size may not be less than 1");

  int instancesToAdd = numResourcesInPool - resources.size();
  if (numResourcesInPool > 0)
  {
    for (int i=0; i<instancesToAdd; ++i)
    {
      R resource = (R)((R)prototype).clone();
      add(resource);
    }
  }
  else if (numResourcesInPool < 0)
  {
    synchronized (availableResources)
    {
      synchronized (resources)
      {
        for (int i=instancesToAdd; i<0; ++i)
```

```
          {
            // Spin lock for removing resource from pool

            while (availableResources.isEmpty())
              try
              {
                availableResources.wait();
              }
              catch (InterruptedException e) {}

            R resource =  getResource();
            resources.remove(resource);
          }
        }
      }
    }

    private void add(R resource)
    {
      synchronized(resources)
      {
        synchronized(availableResources)
        {
          resources.add(resource);
          availableResources.add(resource);
          availableResources.notify();
        }
      }
    }

    public void init(R prototype, int initialPoolSize)
    {
      this.prototype = prototype;
      add(prototype);
      resize(initialPoolSize);
    }

    private Set<R> resources = new HashSet<R>();
    private Set<R> availableResources = new HashSet<R>();
    private R prototype;
}
```

21.4.1.1.4. Using the basic resource pool

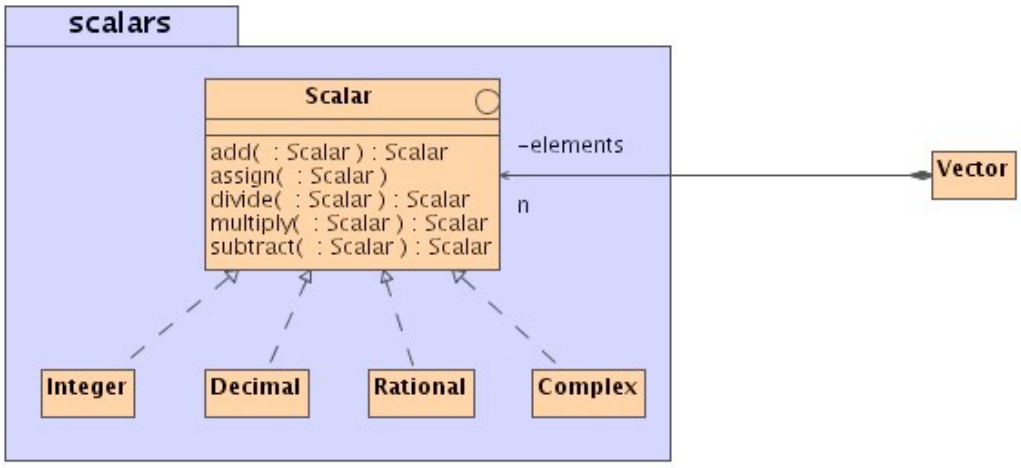
```
package za.co.solms.utils.resource.pooling;

import java.util.*;

public class ResourcePoolTest
{
  class A implements Prototype<A>
  {
    public A clone()
    {
      A copy = null;
      try
      {
        copy = (A)super.clone();
      }
      catch (CloneNotSupportedException e) {/* never thrown */}
      return copy;
    }
  }

  public void run()
  {
```

```
    System.out.println("Creating pool");
    resourcePool = new BasicResourcePool<A>();
    System.out.println("Pool state: " + resourcePool.getNumFreeResources()
        + " available from a total of " + resourcePool.getSize());
    System.out.println("Initializing pool");
    resourcePool.init(new A(), 5);
    System.out.println("Pool state: " + resourcePool.getNumFreeResources()
        + " available from a total of " + resourcePool.getSize());

    new Thread()
     {
       public void run()
       {
         try
         {
           Thread.currentThread().sleep(4000);
         }
         catch (InterruptedException e) {}
         System.out.println("Now resizing");
         ResourcePoolTest.this.resourcePool.resize(7);
         System.out.println("Pool state: "
            + ResourcePoolTest.this.resourcePool.getNumFreeResources()
            + " available from a total of "
            + ResourcePoolTest.this.resourcePool.getSize());
         try
         {
           Thread.currentThread().sleep(4000);
         }
         catch (InterruptedException e) {}
         Iterator<A> iter = as.iterator();
         A a = iter.next();
         as.remove(a);
         System.out.println("Releasing 1 resource");
         resourcePool.releaseResource(a);
       }
     }.start();


    as = new HashSet<A>();
    for (int i=0; i<8; ++i)
    {
      System.out.print((i+1) + ": Retrieving resource: ");
      A a = resourcePool.getResource();
      as.add(a);
      System.out.println(a);
    }
  }

  public static void main(String[] args)
  {
    new ResourcePoolTest().run();
  }

  private ResourcePool<A> resourcePool;
  private Collection<A> as;
}
```

21.4.1.2. Using interfaces in place of templates or generics

An alternative to templates/generics which, in many cases is preferable to the latter is to use interfaces.

Figure 4.66. Scalar defines the minimum requirements for elements of linear algebra classes like Vector or Matrix. Scalar::Integer, ..., Scalar::Rational implement Scalar and can hence be used as elements.



Consider, for example, our vector template specified. The Java implementation could be provided by a `Vector` class obtaining a collection of elements as arguments. The minimal requirements could be defined in a `Scalar` interface as is done in Figure 4.66, " Scalar defines the minimum requirements for elements of linear algebra classes like Vector or Matrix. Scalar::Integer, ..., Scalar::Rational implement Scalar and can hence be used as elements. ". The vector operations can now be implemented, since it is known that the elements supply the required functionality. Note that the composition link refers to a interface implying that the components may be any objects of any class implementing the interface.

21.4.2. Implementing template types in C++

C++ provides direct support for parametrized types via templates. The syntax is very close to that used in UML. Below we show an implementation of the parametrized vector type of Figure 4.65, " A vector template and various classes generated from it. ".

```
template <class T, int n> class Vector
{
  public:
    Vector<T,n> operator+ (const Vector<T,n>& vec) const;
    Vector<T,n> operator- (const Vector<T,n>& vec) const;
    T           dotProduct(const Vector<T,n>& vec) const;

    T norm() const;

  private:
    T* _array;
};

typedef Vector<float,3> Point3D;
```

Here we defined the parametrized `Vector` type and we generated the class `Point3D` by binding the `Vector` template to the type `float` and the integer constant `3`. Note that instances of the class acquire an additional private integer constant, {\bf n}, as data attribute.

We cannot create instances (objects) of the template type `Vector` without binding the parameters. However, we can create instances of `Point3D` and we can create instances of `Vector` if we bind the template parameters:

```
Point3D point3D;

Vector<Complex,20> data;
```

The compiler will generate the specified classes from the template type. This is done at compile/link time and there are no run-time overheads.

22. Exercises

1. Select one of the use cases for the system for an e-commerce retailer for which you developed the use case view and

   - identify the core objects,
   - their abstractions as classes,
   - their attributes and services,
   - and the relationships these classes have amongst each other.

Chapter 5. The Dynamic View

1. Introduction

So far we looked at

- *use case diagrams* which identified
  - the external objects interfacing with the system and
  - the use they get out of the system and at

  object and class diagrams which are used to document the system structure including
  - the objects and their abstractions as classes,
  - the attributes of the objects,
  - the services they supply and
  - the relationships they have with other objects.

In this chapter we will look at the dynamics of the system. This dynamics will occur within the context of the static structure. Ultimately we need a process to identify the structure required to support the dynamics (e.g. the business or system processes).

1.1. UML diagrams for the dynamic model

For now we will discuss the diagrams UML provides to document the dynamics of a system, i.e. how the objects collaborate to realize the use cases. There are two groups of diagrams:

1. *Interaction diagrams* look at how the objects interact in the context of the collaboration realizing the use cases.
2. *Behavior diagrams* focus on what the objects do in the context of the processes realizing a use case.

1.1.1. Interaction diagrams

Interaction diagrams focus on the interactions between objects in the context of processes realizing use cases. They show the sequence of messages exchanged between the objects. UML supports two types of interaction diagrams:

- *Sequence diagrams* which show the object participating in the collaboration along one axis and time on another.
- *Collaboration diagrams* which show the messages being sent along message paths provided by the system structure with the time ordering of the messages specified by a numbering system.

Sequence diagrams are particularly intuitive and can be used to explain business and system processes even to people who may not be familiar with the notation. Collaboration diagrams are less intuitive, but they will prove very useful as a transition path from the dynamic to the static model.

1.1.2. Behavior diagrams

While interaction diagrams focus on the communication between the objects, behaviour diagrams focus on what the actions do, the states they are in, the state transitions and the events which cause them. UML supports two types of behavior diagrams:

- *Activity diagrams* which look at the activities done by the objects participating in the processes realizing the use case, at the events which cause the transitions, at the concurrencies within these processes and the synchronization points.
- *State charts* show the full state dynamics of an object. UML state charts are derived from Harel state charts.

An activity diagram for a single object can be viewed as a subset of a state chart where only the activity aspect of the states is taken into account. However, an activity diagram can also show the activities across objects and provides a simple, intuitive graphical notation for business processes and processes.

2. Sequence Diagrams

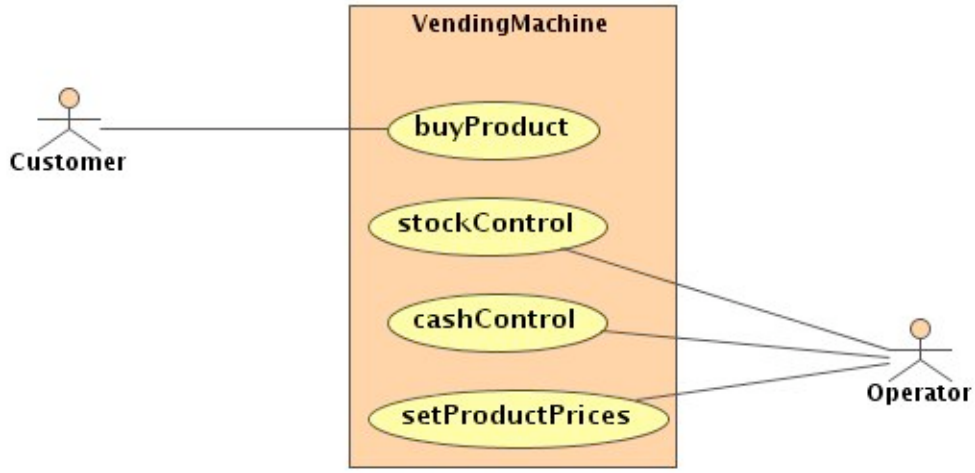
We have encountered sequence diagrams already in the use-case view where we treated the system we are modeling as a black box and showed the messages exchanged between the system and its actors. In this section we shall use sequence diagrams to show *how the components of a system interact to realize a use case*. In this context we will cover further features of sequence diagrams including

- generic sequence diagrams,
- object life-cycle management,
- message types,
- timing constraints and
- concurrencies in sequence diagrams.

2.1. A vending machine example

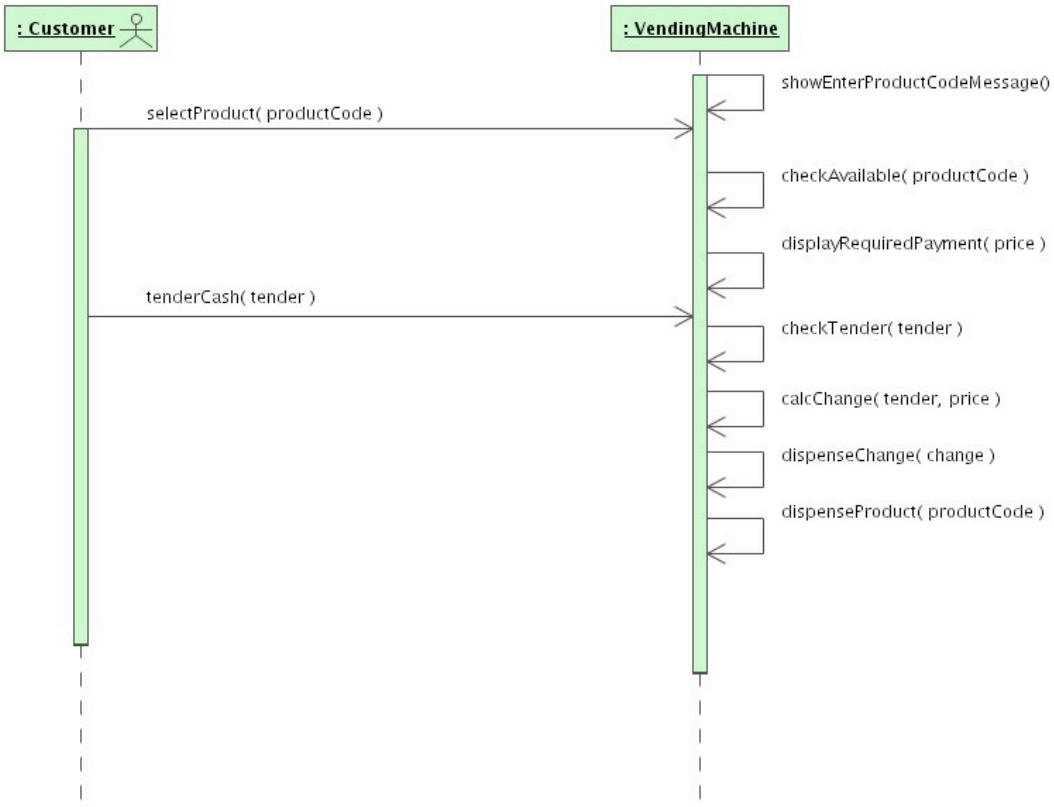
Let us, as a simple example, consider a vending machine.

Figure 5.1. Use case diagram for a vending machine.



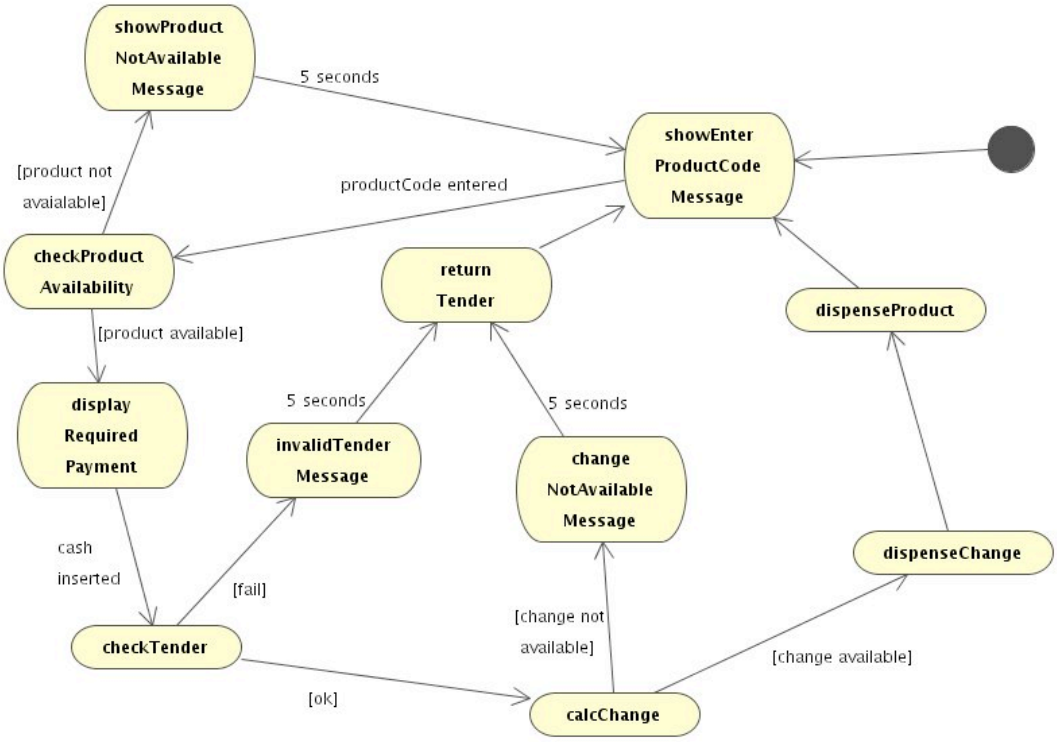
Looking at the buy-product use case we could come up with the following high-level sequence diagram showing the messages exchanged between the client and the vending machine:

Figure 5.2. A sequence diagram for the buy-product use case.



We can introduce a high level activity diagram showing multiple scenarios of buying a product from a vending machine.

Figure 5.3. An activity diagram for the buy-product use case.



2.2. Responsibility identification and allocation for the buy-product use case of a vending machine

In Figure 5.4, “ Identifying the responsibilities which need to be addressed for the buy-product use-case. ” we identify the core responsibilities which need to be addressed for the *buy-product* use case.

Figure 5.4. Identifying the responsibilities which need to be addressed for the buy-product use-case.

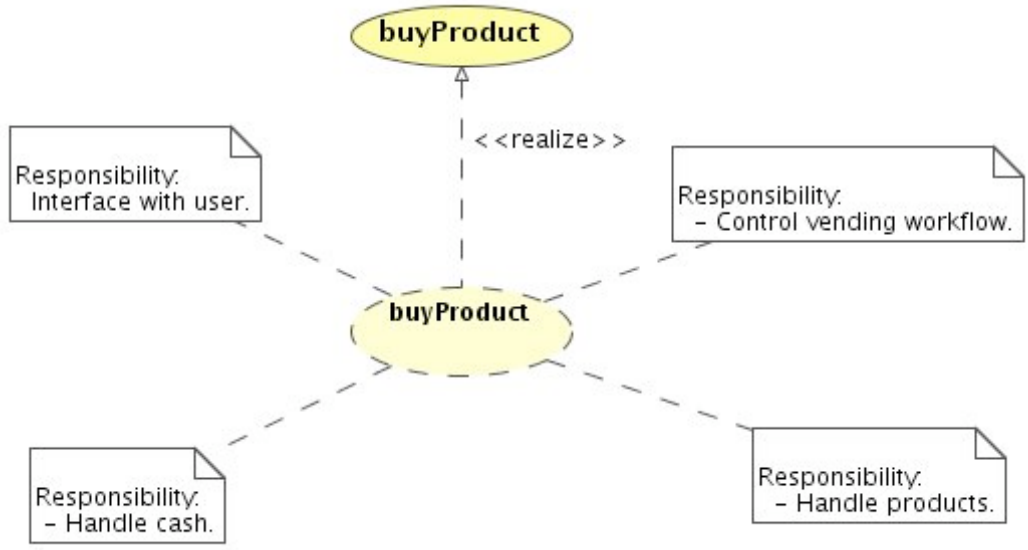


Figure 5.5, “ Allocating the responsibilities which need to be addressed for the buy-product use-case

to core system components. ” shows the core components of a vending machine, each of which acquires one of the core responsibilities which need to be addressed to realize the *buy-product* use case.

Having identified the responsibilities, we can now go ahead and assign them to core system components or business units.

Figure 5.5. Allocating the responsibilities which need to be addressed for the buy-product use-case to core system components.

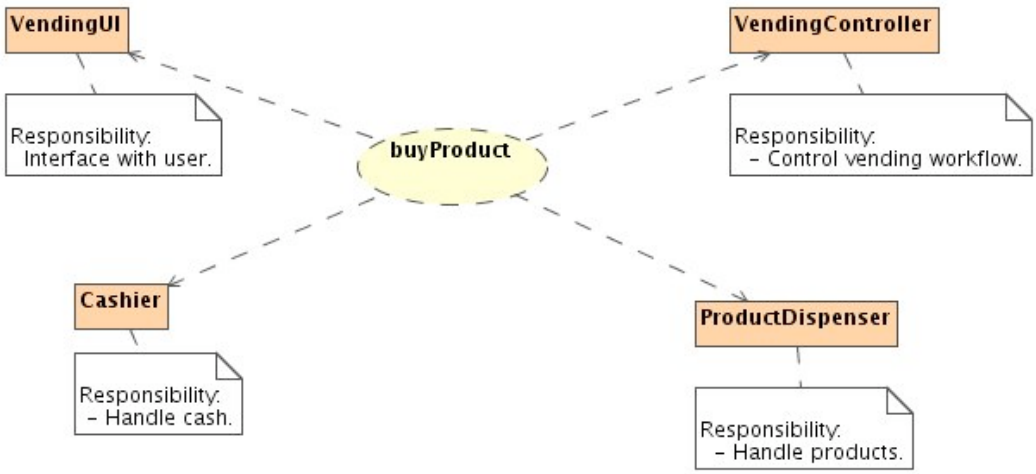


Figure 5.5, “ Allocating the responsibilities which need to be addressed for the buy-product use-case to core system components. ” shows the core components of a vending machine, each of which acquires one of the core responsibilities which need to be addressed to realize the *buy-product* use case.

2.3. Sequence diagrams showing the interactions of the core components

Having identified the core responsibilities and having allocated them to core components or business units, we are now in a position to analyze how the core business units or components collaborate to realize the use case. For this we use, once again, a sequence diagram, but this time the objects are the actors plus the core components of the system.

Reconsider the vending machine example used earlier. We used an abstract collaboration which resembles a collaboration of objects which realizes a use case. However, before identifying the objects themselves, we identified the core responsibilities which needed to be addressed to realize the use case. We then assigned these responsibilities to objects, giving each object a well-defined responsibility focus.

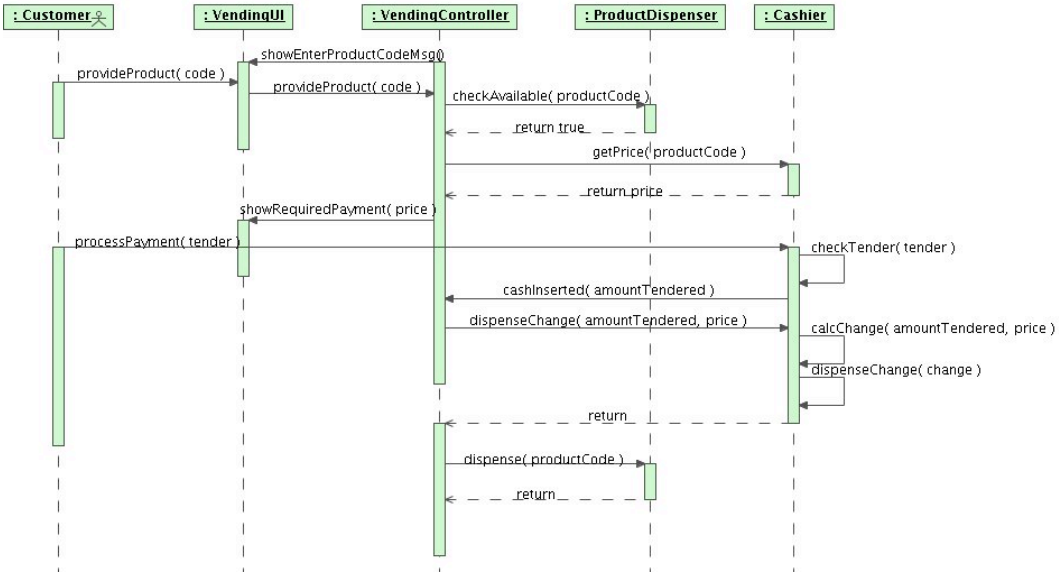
This led us to identify 4 high-level components of a vending machine:

- the user interface,
- the controller,
- the product dispenser and
- the cashier.

The next step is to look at how these high-level components interact to realize the use case. Fig-

ure 5.6, “ A sequence diagram for the buy-product use-case, showing how the core components of the vending machine collaborate to realize the use case. ” depicts a particular scenario (the one where the purchase is successful) of the *buy-product* use case.

Figure 5.6. A sequence diagram for the buy-product use-case, showing how the core components of the vending machine collaborate to realize the use case.



2.4. Generic sequence diagrams

Figure 5.6, “ A sequence diagram for the buy-product use-case, showing how the core components of the vending machine collaborate to realize the use case. ” depicts an instance sequence diagram which shows a particular scenario, i.e. a particular sequence of messages exchanged between the objects. The particular flow of messages depicted in the sequence diagram of Figure 5.6, “ A sequence diagram for the buy-product use-case, showing how the core components of the vending machine collaborate to realize the use case. ” is only realized if a set of conditions is satisfied. One can use generic sequence diagram to show all possible message paths.

2.4.1. Branching

To show multiple paths within a single sequence diagram one uses branching. Branching is thus used to show a number mutually exclusive paths where, in each scenario only one path is chosen.

In principle, one can use a generic sequence diagram to specify a whole range of instance sequence diagrams (scenarios). An *instance sequence diagram* is thus a particular path through a *generic sequence diagram.*

2.4.1.1. Guard conditions

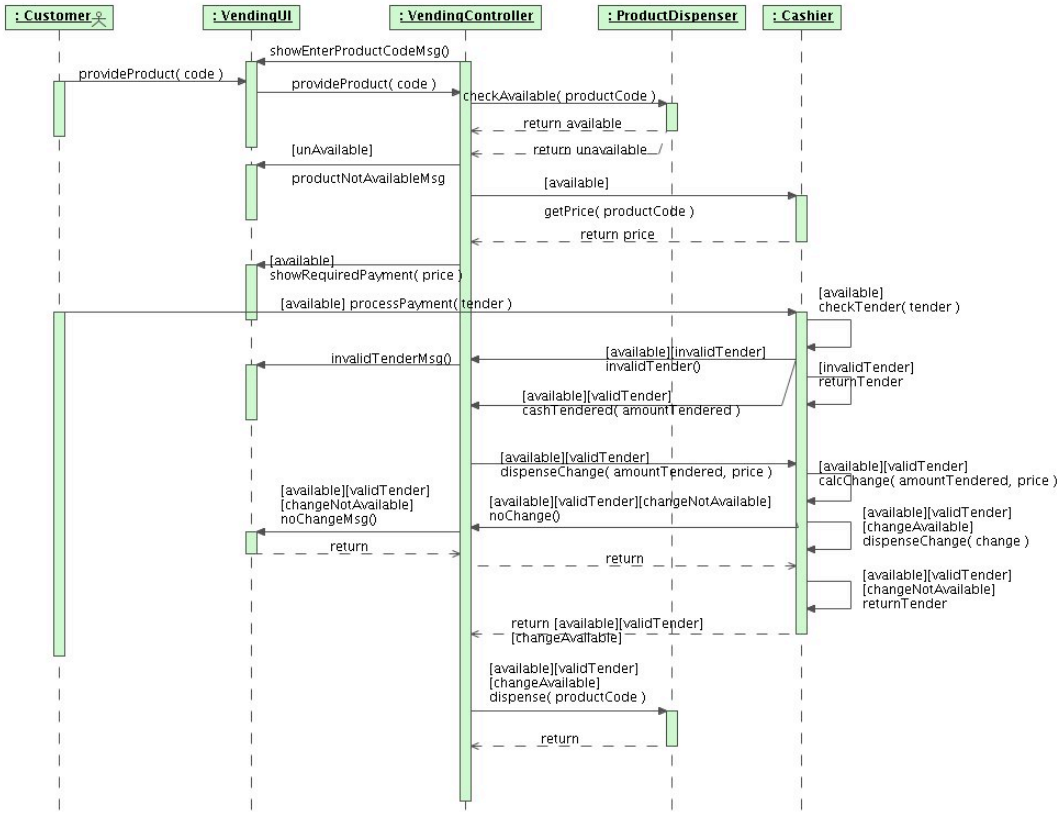
The different paths through a generic sequence diagrams are followed under different conditions. To specify a conditional path, one assigns a *guard condition* to a message arrow.

Guard conditions are shown within square brackets and the message name is placed either after the guard condition or on the other side of the message arrow.

2.4.1.2. Generic sequence diagram for the vending machine

Figure 5.7, “ A generic sequence diagram showing multiple scenarios for the buy-product use-case. ” depicts a generic sequence diagram for a vending machine.

Figure 5.7. A generic sequence diagram showing multiple scenarios for the buy-product use-case.



2.4.2. Disadvantages of generic sequence diagrams

Looking at Figure 5.7, “ A generic sequence diagram showing multiple scenarios for the buy-product use-case. ” you will most probably find that the diagram is not very intuitive. Nor is it really well-defined. We actuallly need to carry all guard condition down through the remainder of the diagram to make the diagram unambiguous. The extra complexity often defeats the core purpose of a sequence diagram: that of clearly, simply and intuitively documenting aspects of a workflow and the ability to use the diagram to validate the requirements with the client and users who may not be very proficient in UML.

Sequence diagrams were developed to document a particular sequence of messages, i.e. a particular scenario. Its used should be largely confined to that of documenting single scenarios.

2.4.3. Alternatives to generic sequence diagrams

So, if sequence diagrams are not that well suited to document multiple scenarios, what should we use instead. of course we can draw sequence diagrams for the rest of our natural life to document all the different scenarios. Alternatively we can use activity diagrams to show multiple scenatios within a single diagram.

Activity diagrams are much better suited for documenting multiple paths in a clear, simple and non-ambiguous way. Furthermore,

2.5. Object life cycle modeling with sequence diagrams

So far the object which paricipated in the interactions depicted in the sequence diagrams excisted accross all time. Often we need to show objects which are created and destroyed within a workflow.

2.5.1. Life lines and activity bars

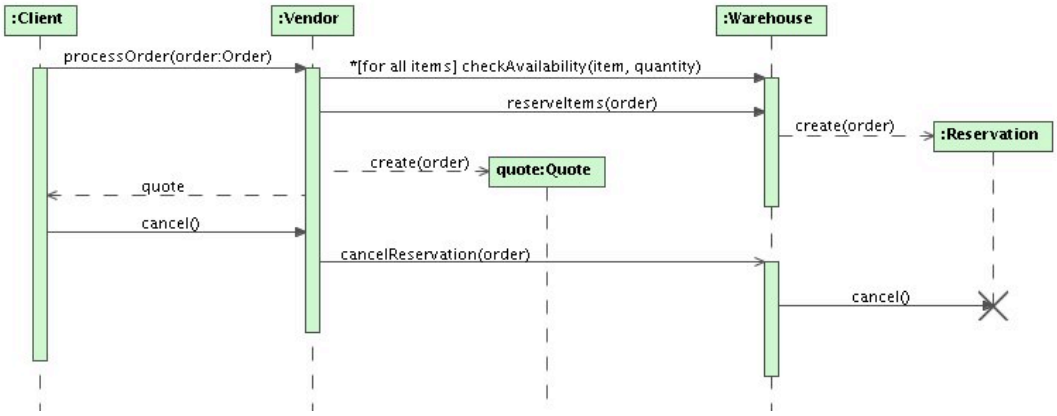
Let us first revisit the sequence diagram shown in Figure 5.6, " A sequence diagram for the buy-product use-case, showing how the core components of the vending machine collaborate to realize the use case. ". Below each object there is a dashed line. This is the *life-line* of the object, indicating the period over which the object exists.

On top of these life lines we see bars. These are the *activity bars* indicating the period over which the period is active, e.g. busy processing a service request.

2.5.2. Object creation and destruction

To show the request for the creation of an object, we draw a a message ending on the actual object diagram. For example, in Figure 5.8, " A sequence diagram showing the creation and destruction of an order processor. " we show how the `WareHouse` creates a `Reservation`, for an `order`.

Figure 5.8. A sequence diagram showing the creation and destruction of an order processor.



At a later stage, after the client received a quote for the order and rapidly pressed the cancel-order button, the warehouse destroys the `Reservation`. A message requesting the destruction of the object terminates its life-line. Additionally a cross is added to mark the destruction of the object more clearly.

2.5.3. Iteration

Figure 5.8, " A sequence diagram showing the creation and destruction of an order processor. " also makes use iteration. We indicate an iteration using a `*` while the condition for continuing with the iteration is shown in a standard conditional (within square brackets).

2.6. Message types

Messages can be sent by clients in different ways. The client could, for example, wait for a reply or not expect a reply at all. Different message types thus specify different client behavior upon sending a message. The server side may be unaware of the way the client sent the message.

Figure 5.9. Types of messages supported in UML.

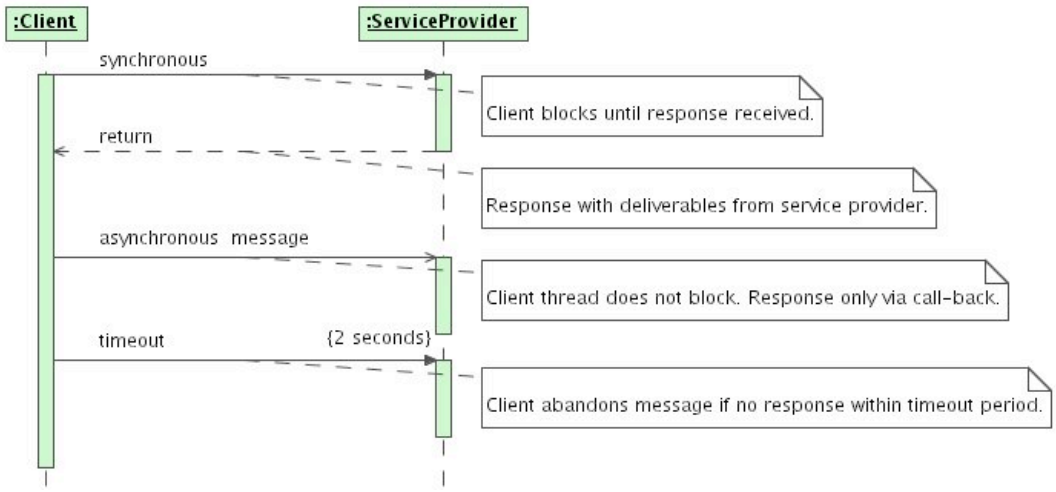


Figure 5.9, " Types of messages supported in UML. " shows the message types supported in UML.

2.6.1. Synchronous messages

A {\em synchronous message} is one where the processing of the client halts until a response from the service provider has been obtained. The sender resumes its own execution thread as soon as the message has been accepted.

2.6.2. Timeout messages

A *timeout message* is one where the sender waits for a certain period and if the receiver has not accepted the message within the specified timeout period, the message is abandoned from the clients perspective, i.e. the client will continue under the assumption that the requested service could not be supplied.

Note

The service provider may, in principle still continue to process the service request.

2.6.3. Asynchronous messages

An *asynchronous message* is one where the sender sends a message and immediately continues with its own execution thread without waiting for a response. The sender does not yield control to the receiver and the two objects would typically perform their respective actions concurrently.

2.6.4. Returns

A return represents the response to a synchronous service request. It may contain certain deliverables for the client (return values and output parameters) or simply act as acknowledgement that the requested service has been supplied.

In either case the service provider does not require a message path to the client. Consider, the example of placing an order via telephone. The client has a message path to the service provider (the telephone number). He/she uses this message path to send a sychnronous service request to the service provider. The client will wait (block) until he/she received a response. The service provider can provide the response to the client without having a message path to the client (he does not need to know the client's telephone number to provide the response).

2.7. Implementing different types of messages in Java

Let us now look how we would map the UML message types onto Java code.

2.7.1. Synchronous messages

A synchronous message is a standard blocking call and simple maps onto a Java method call:

```
serviceProvider.doThis(info)
```

2.7.2. Simple calls

A simple call is in principle under-specified. However,in the absence of any further information it would typically map onto a standard synchronous method call.

2.7.3. Asynchronous messages

An asynchronous message is a message which is sent in a seperate execution thread, enabling the main execution thread to continue with its processing. It can be implemented in a compact way using an anonymous inner class:

```
new Thread(){public void run(serviceProvider.doThis(info));}.start();
```

Here we define an anonymous subclass of `Thread`, override it's `run` service, create an instance of this thread, and finally we register with the thread scheduler

2.7.4. Sychronous call with immediate return

A sychronous call with immediate return is implemented on the client side via a standard synchronous call. The server, on the other hand, should try andreturn before the timeout period. Often the server would defer procesing to a seperate execution thread.

2.8. Further timing features for sequence diagrams

So far we have applied timing constraints to messages. Sequence diagrams are, however,well suited for specifying further timing aspects of workflows.

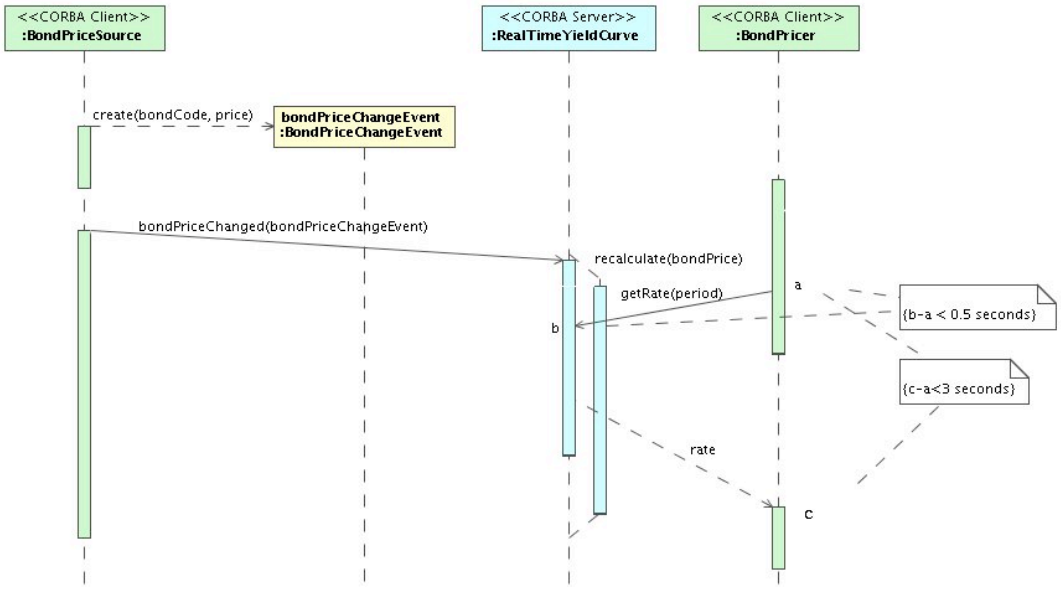
2.8.1. Non-instantaneous messages

Recall that time is along one of the axes in a sequence diagram (often the vertical axis). If we draw a message horizontally, then it is assumed that the message is instantaneous, i.e. that it will be received at the same instant as it was sent (from the perspective of the model).

If we want to indicate that it takes time to deliver a message or a response, the message arrow mat slope downwards, explicitely indicating that the message is going to be received after it as been sent.

For example, in the distributed CORBA system show in Figure 5.10, " Sequence diagram with concurrencies, non-instantaneous messages and timing constraints. ", the remote service request are shown as non-instantaneous messages.

Figure 5.10. Sequence diagram with concurrencies, non-instantaneous messages and timing constraints.



2.8.2. Timing constraints

We can attach labels onto the time access and specify constraints using these labels. For example, in Figure 5.10, “ Sequence diagram with concurrencies, non-instantaneous messages and timing constraints. ”, we attached the label `a` and `b` to the time access and specified that the period between these two time instants must be no more than halh a second.

2.9. Concurrencies

Concurrencies involve simultaneous activities. Concurrencies may be accross objects or within the same object. Every time you see multiple activity bars at the same instant, you have a concurrency.

2.9.1. Concurrent activities within an object

In Figure 5.10, “ Sequence diagram with concurrencies, non-instantaneous messages and timing constraints. ” we have concurrencies accross objects as well as concurrencies within the same object. In the latter case the object does multiple tasks simultaneously. This is shown in UML by a splitting of the life-line with multiple parallel activity bars hosted on these paralell life lines.

In our example, the real-time yield curve can provide the latest available rate, even while it is recalculating its state.

2.9.2. Concurrencies versus branching

The only difference between branching and concurrency is that in the case of branching the arrows have mutually exclusive guard conditions. If the guard conditions are not mutually exclusive, the diagram specifies concurrency.

2.10. Making classes safe for concurrent access

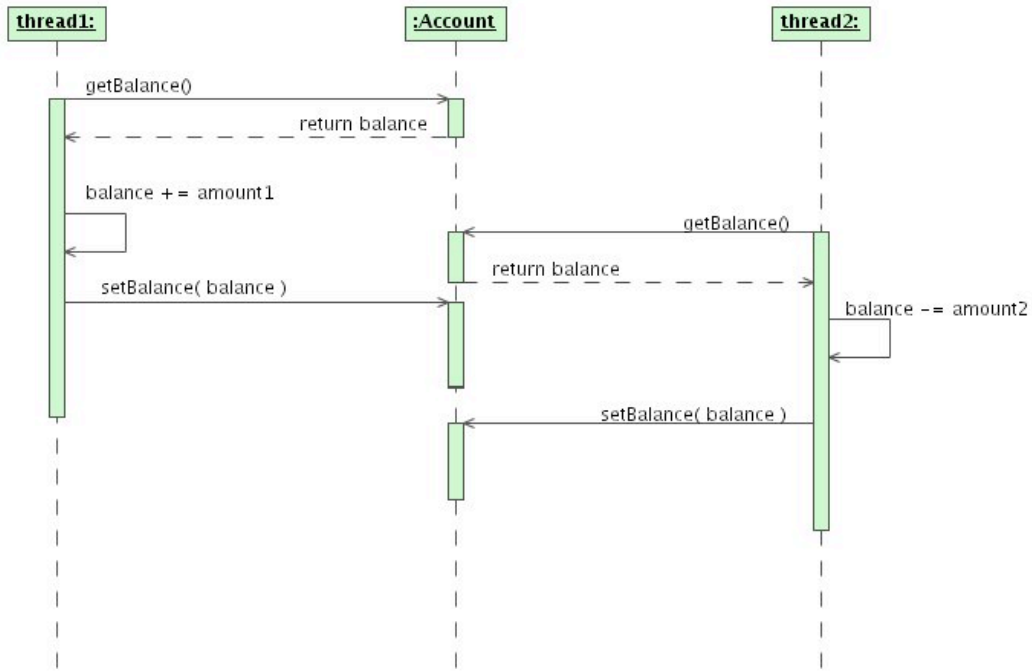
If there are concurrent processes, the resources used by these concurrent processes can get corrupted. We need a protection mechanism which prevents this. Technically we say that we need to make the classes thread-safe.

2.10.1. Why do we need access control?

Consider Figure 5.11, “ Corruption of an object's due to concurrent access. ” which shows two execution threads accessing a shared resource, an account. The first thread wants to credit the account

with `amount1`, while the second thread wants to, say, debit it with an `amount2`. The first thread starts by querying the balance,calculates the new balance and updates it.

Figure 5.11. Corruption of an object's due to concurrent access.

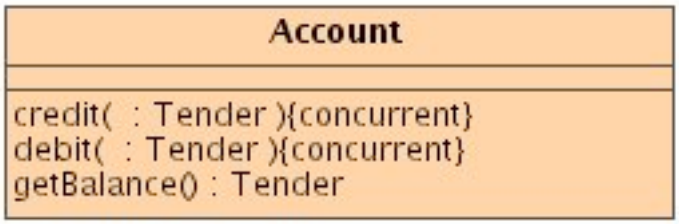


While the first thread is busy processing, the second thread concurrently perform the debit process. It queries the balance, calculates the new balance and updates it. Notice, however, that the first transaction is effectively since the second thread get the balance before the first transaction is completed.

2.10.2. Protection against corruption due to concurrent access

In order to protect a resource against corruption due to concurrent access, we need to serialize at least some of its services. In our case, no credit or debit transaction may start while another credit or debit service is busy.

Figure 5.12. The concurrent constraint



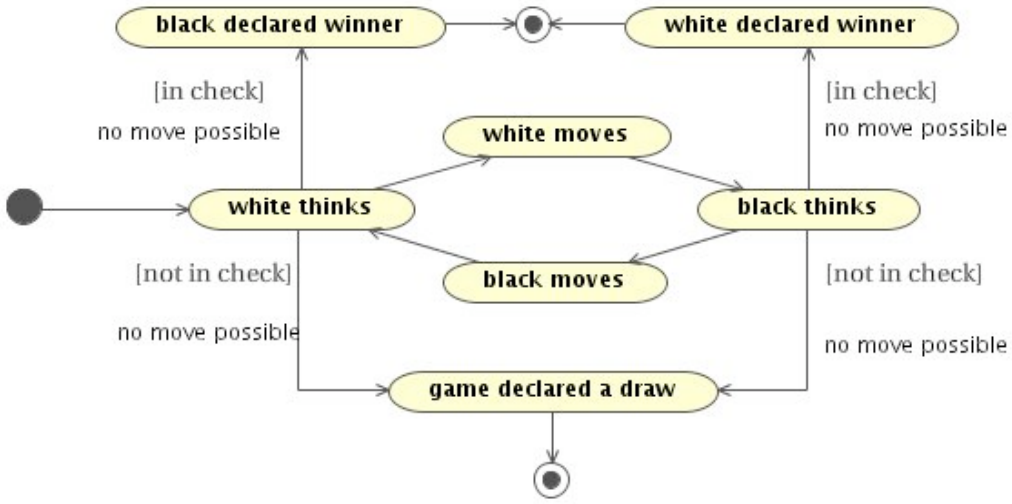
UML provides a `{concurrent}` constraint to specify that a certain services will be provided by a service provider in a way which is safe under concurrent access.

3. Activity diagrams

3.1. Exit states

An activity diagram may have only a single entry state, but may have multiple or exit states. There are no further activities performed after the exit state in the context of the process documented in the activity diagram. An exit state is drawn as a filled circle with an empty circle around it.

Figure 5.13. Entry and exit states.



In Figure 5.13, “ Entry and exit states. ” we show the activities of a game of chess with transitions to the exit state from either a draw or if one of the players wins.

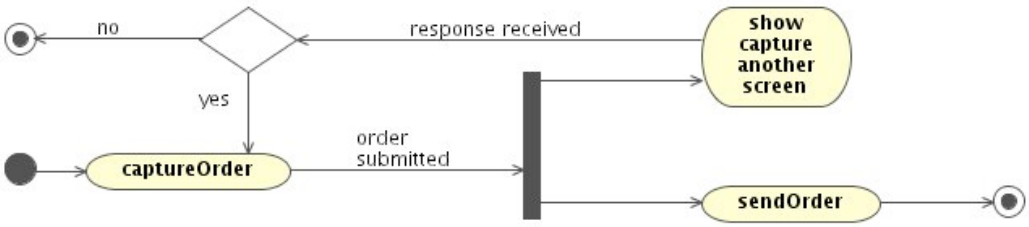
3.2. Forking and synchronization

3.2.1. Forking

If one execution thread spawns another thread or process, the thread is said to fork. A fork thus represents the splitting of a single flow of control into two or more concurrent flows of control.

One execution thread can terminate while others are still processing. This is shown by drawing a transition from the final state of the thread to the exit state. Figure 5.14, “ Forking into concurrent activities. ” shows the forking af the execution thread after the order has been submitted into one thread which sends the order and another prompting the user whether he/she would like to capture another order. Consequently the capturing and sending of orders can occur concurrently

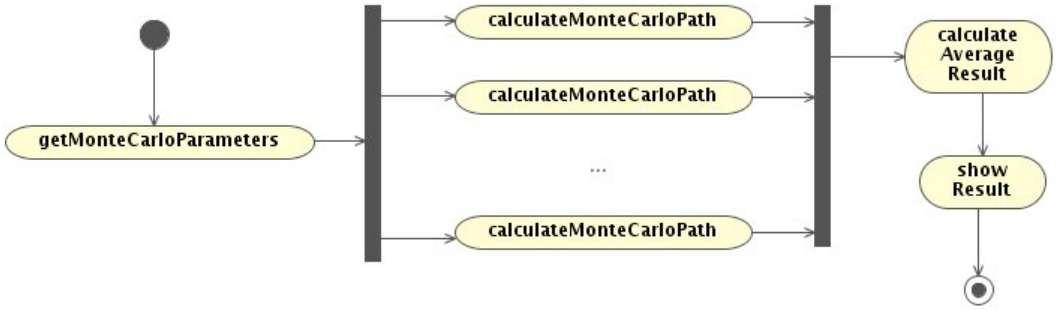
Figure 5.14. Forking into concurrent activities.



3.2.2. Synchronization

Similarly, multiple exection threads can join into a single execution thread. At the join the concurrent flows synchronize, i.e. all threads of activity wait until all incoming threads have reached the join. Thereafter execution continues with the single flow of control leaving the join.

Figure 5.15. Forking and joining via synchronization bars.

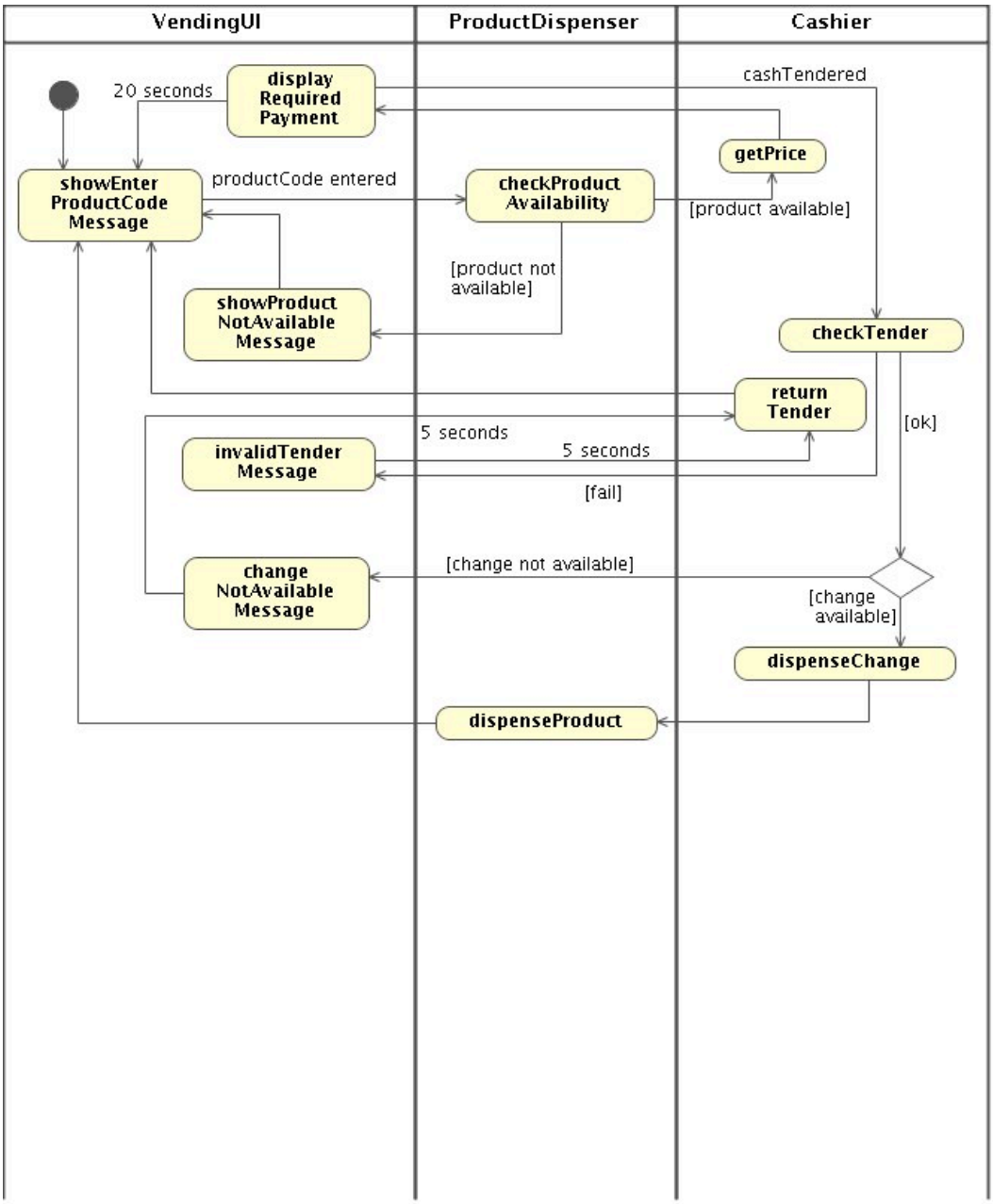


For example, the activity diagram in Figure 5.15, " Forking and joining via synchronization bars. " shows how concurrent implementation of a Monte Carlo (random path) simulation. After the Monte Carlo parameters have been obtained the thread forks into multiple threads, each calculating a single Monte Carlo path. Each path has to be completed before the concurrent threads or processes recombine into a single execution thread calculating the average over the paths and displaying the result.

3.3. Activities across objects: swim-lanes

At times one may want to show activities across objects. This can be particularly useful for documenting business processes which are executed across business units. For this UML provides the notation of swim-lanes which packs the class diagrams for the objects participating in the collaboration which realizes the use case next to each other along either the horizontal or vertical axis.

Figure 5.16. Activity diagram showing how the core components collaborate to realize the buy-product use-case of a vending machine.



Example 5.1. Vending machine

Figure 5.16, " Activity diagram showing how the core components collaborate to realize the buy-product use-case of a vending machine. " shows how the core components of a vending machine (drawn as objects along the horizontal axis) collaborate to realize the buy-product use case.

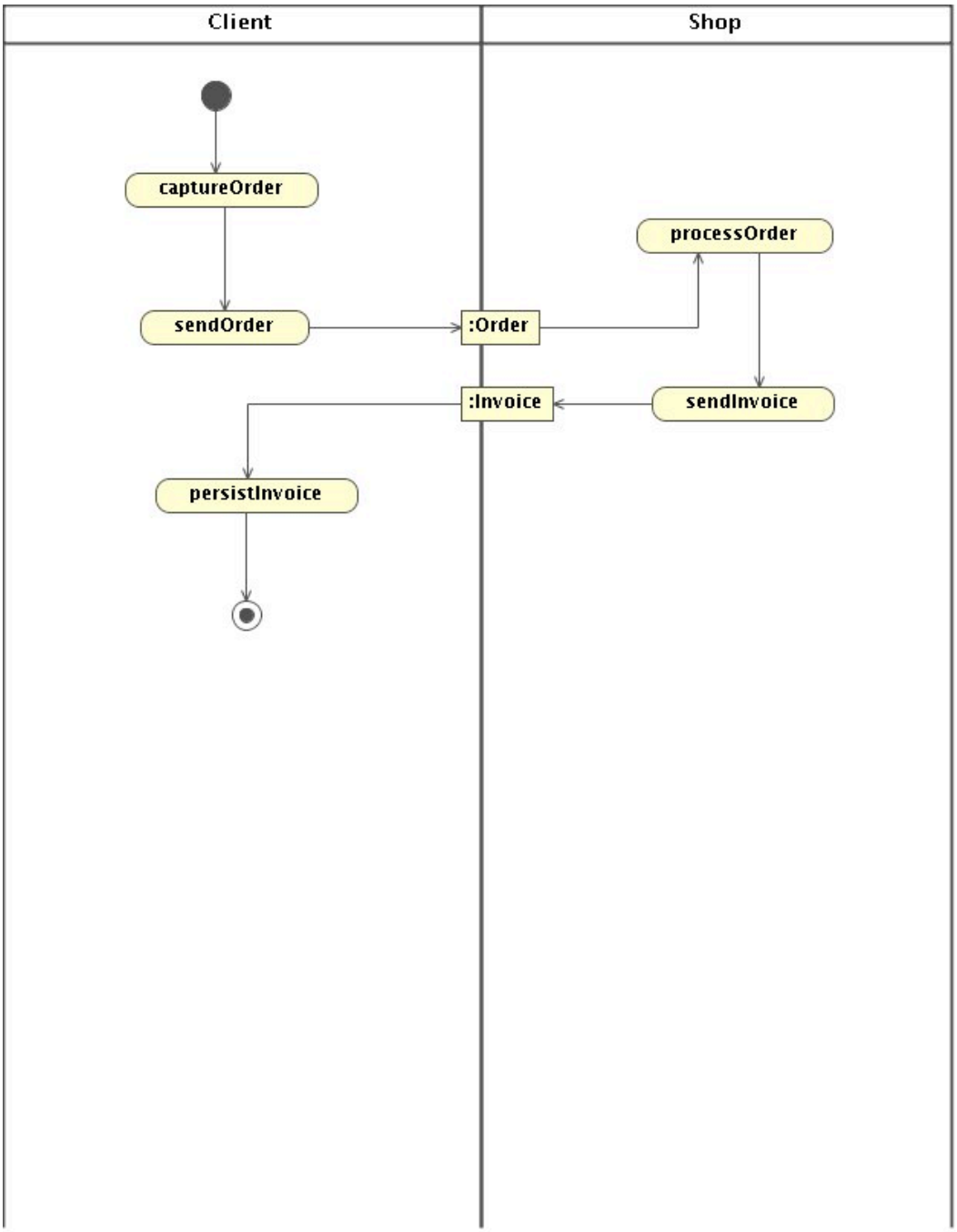
3.4. Showing object flow in an activity diagram

sequence diagrams show object flow as either

- the parameters in a service request sent from one object to another, or
- the return values of a service request.

So far we have not shown object flow in activity diagrams. UML does support object flow in activity diagrams by inserting object diagrams into the transition.

Figure 5.17. Showing object flows in an activity diagram.



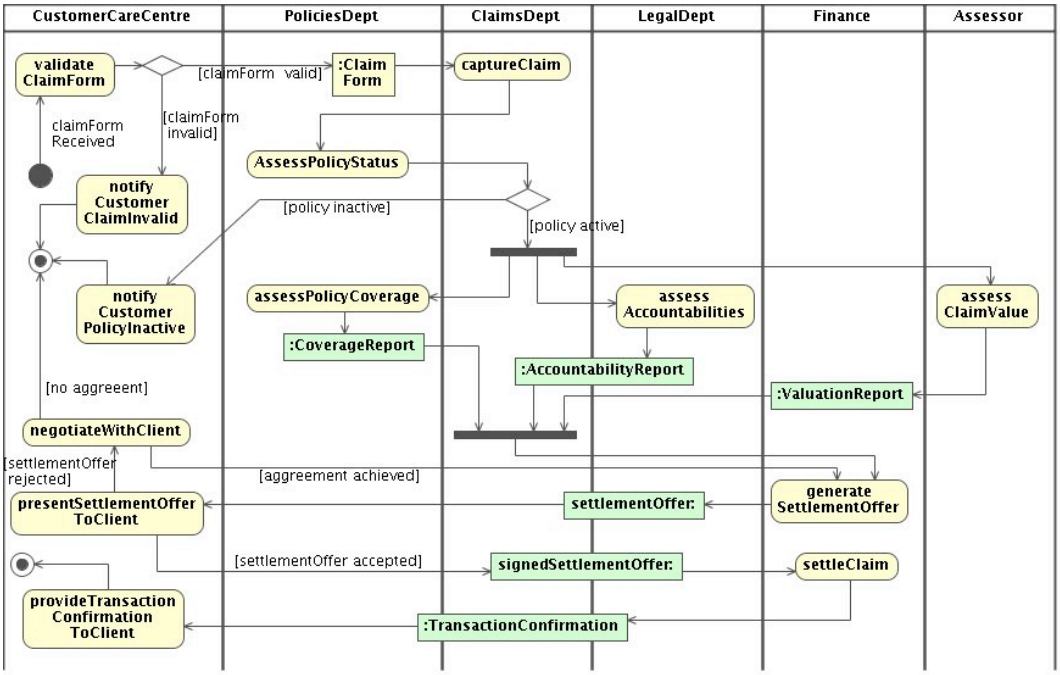
3.5. Processing an insurance claim

In this section we shall look at an example which uses the notation we have discussed so far. We shall use an example of modeling a business process, showing activities across different business units and concurrencies within the business process.

Example 5.2. Processing an insurance claim

We now need to look at how these business units collaborate to realize the use case. We can start by drawing sequence diagrams showing scenarios which are representative for the interactions between these business units. After that we would look at an activity diagram showing all the possible scenarios as well as potential concurrencies and the synchronization points required to synchronize these concurrent processes. he process is depicted in Figure 5.18, " Activity diagram showing how the business units collaborate to realize the process-claim use-case. ".

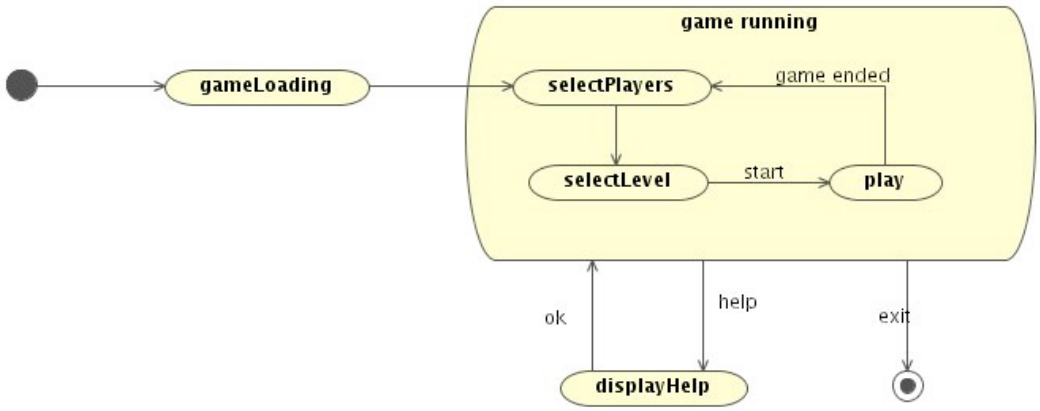
Figure 5.18. Activity diagram showing how the business units collaborate to realize the process-claim use-case.



3.6. Nested activities

At times one needs to show transitions which is common to a whole range of states. Adding all these transition would, however, make the activity diagram difficult to read. To show transitions which are common accross a number of states in a clean,intuitive way UML supports the concept of nested states.

Figure 5.19. Showing common transitions via nested activities.



For example, Figure 5.19, " Showing common transitions via nested activities. " shows that help and exit are available in all states of the game,i.e. whenever the game application is running.

4. State charts

Activity diagrams show one aspect of a state, namely the activity (or activities) performed while the object is in a state. State charts can be used to show the full aspect of the states including local variables of a state. They are used to model the object/system as a state machine. Such state machines are often used to validate a system.

4.1. States

The state of an object or a class is defined by the value of its attributes, the associations to other objects and the operations it currently performs. A state has a duration and thus occupies a period of time.

The UML notation for a state is a rectangle with rounded corners. A state may have a number of attributes and an object may perform a number of actions while it is in that state. The state attributes and actions are specified in separate compartments similar to the compartments for the attribute and operations lists of a class diagram. For example, while the operator is prompted for his/her password a system would be in the `TypingPassword` state. The state diagram for this state is shown in Figure 5.20, " State diagram in UML. ". The left hand diagram shows a simple state. In the right-hand diagram the state is expanded to show a state attribute as well as the actions performed on entry, exit, help event and while the object is in the specified state.

Figure 5.20. State diagram in UML.

The name of the state is shown in bold letters in the top compartment of the state diagram. The name is optional. Unnamed states are *anonymous*, but they are still distinct. If, on the other hand, two states have the same name then they are the same state. It is sometimes useful to place the same state in various places on a state diagram in order to simplify the graphical representation.

An *action* is an operation and it usually involves some computation or the traversing of some al-

gorithm. The actions performed while an object is in a particular state are specified in the last compartment of the state diagram. A *do* action is an ongoing process performed while an object is in a particular state. Actions may be interruptible by external events. For example, our *do*-action can be interrupted by a *help* event which performs the action *display help*. The actions performed on *entry* or *exit* of a state are, however, non-interruptible. They are specified by a *entry* and *exit* events. The UML notation for actions is

```
event-name argument-list '/' action-expression
```

The last compartment holds the state variables. State variables can be class attributes. Alternatively they may be local variables used by the object only while it is in a particular state. State attributes are optional.

4.2. Messages and events

Objects communicate by sending messages to one-another. A message is a one-way transmission of information from one object to another. A *message* is a request for an object to perform an action and the receipt of a message is a special type of an *event*.

In general an event is some or other noteworthy occurrence at some instant in time. In the context of a state diagram an event is usually an occurrence that may trigger a state transition.

UML differentiates between call-events, signal-events, time-events and change events.

4.2.1. Call events

Call events are implemented as synchronous messages, i.e. as standard synchronous method calls. The client sends the message to the server and waits until either the service is completed or until an exception is thrown, notifying the client that the server is unable to provide the service requested.

A call event is specified by the event name and the call arguments list enclosed within parenthesis.

4.2.2. Signals

A signal-event is an event triggered by a signal. A signal event is the receipt of a named asynchronous event. The event source dispatches the signal but does not wait for a response. Instead it continues with its own execution thread directly after sending the signal.

To specify that a particular event is a signal event we attach a `<signal>>` stereotype to the event.

A very large class of signals is that of exceptions. Exceptions are modeled in UML as signals, that is as named asynchronous events. Note that although Java supports exceptions, they are implemented by default as synchronous calls. This is unsatisfactory except in those cases where the exception handler can return virtually immediately. In most cases the exception handler (i.e. the listener) should launch a separate execution thread for processing the exception.

Other signals should be implemented in a similar fashion. The recipient of the signal should launch a new execution thread to process the signal asynchronously.

4.2.3. Time events

A time-event is an event triggered by the passing of time. In UML one can specify the elapsed time directly (e.g. “15 seconds”). By default this specifies the elapsed time since entering the current state. Alternatively one can specify a time-event as a condition (e.g. “[time=16h00]”, or “[14 seconds since entry into state X]”).

4.2.4. State change events

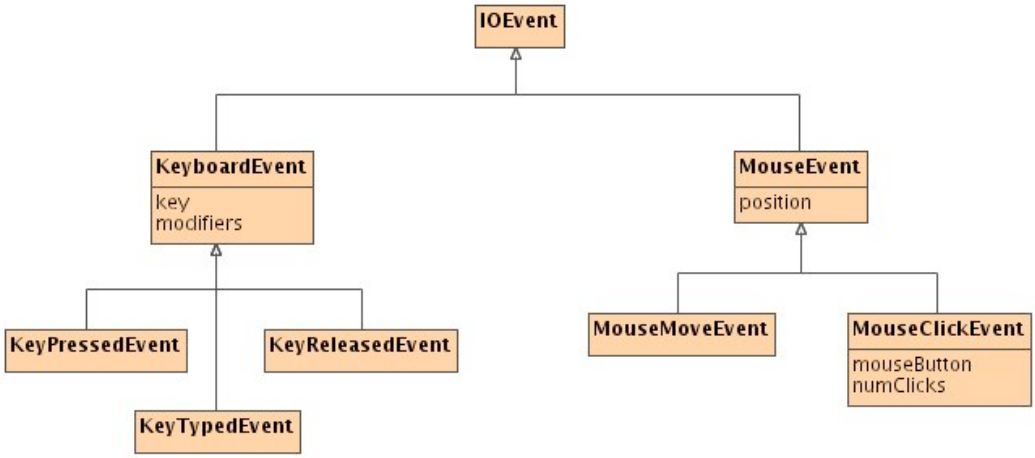
State change events represent an event generated due to an event source undergoing a state trans-

ition. Generally such events are asynchronous events and hence state change events are special kinds of signals. Similarly, time events can be modeled as a special kind of state change event where the state of the clock changes from, for example, `before noon` to `after noon`.

4.2.5. Event specialization

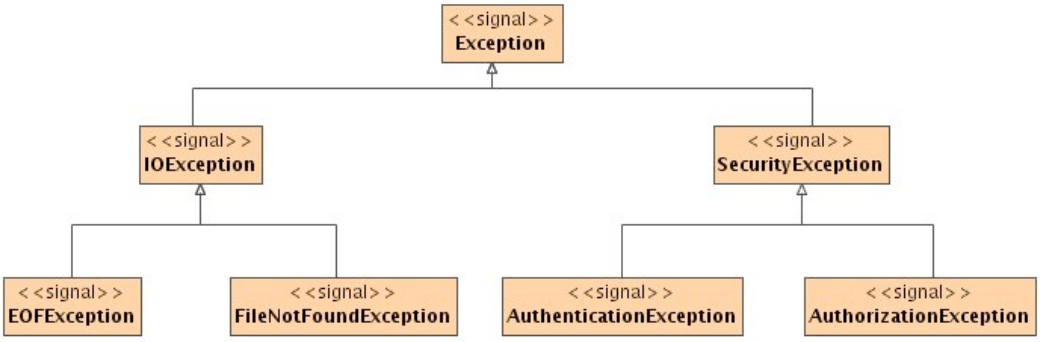
An event can be viewed as an instance of a class. We can thus define event classes and use standard specialization and aggregation relationships between events.

Figure 5.21. Event specialization.



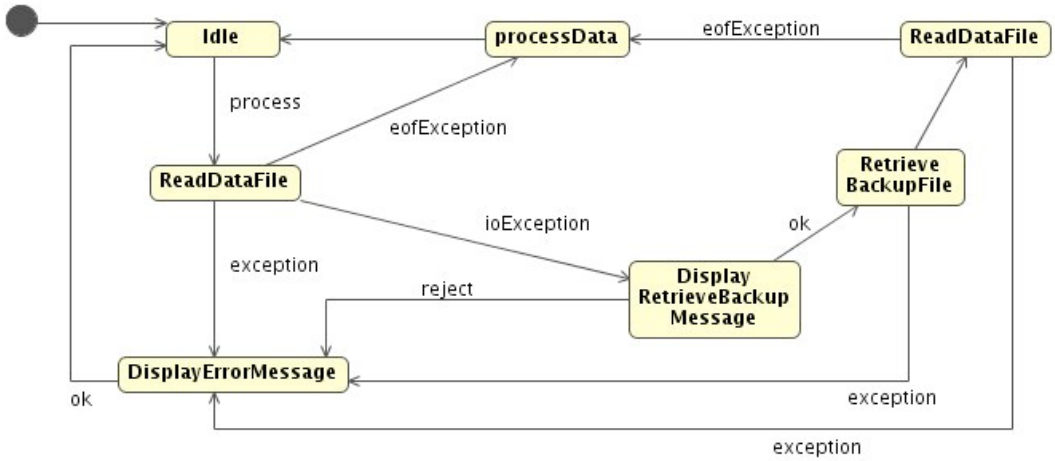
Similarly we can have a specialization hierarchy for exceptions.

Figure 5.22. Exception hierarchy.



Event specialization is particularly useful for specifying and handling abstract events. For example, the specialization hierarchy for exceptions shown in Figure 5.22, “ Exception hierarchy. ” allows us to handle exceptions at various levels of abstraction as is done in Figure 5.23, “ Exceptions at different levels of abstraction guide process flow. ”.

Figure 5.23. Exceptions at different levels of abstraction guide process flow.



One of the advantages for introducing event hierarchies is that it enables a design using events at various levels of abstraction. Consider, for example, the case where we want to read a data file to its end. In this case we might want to use `EOFExceptions` as a signal telling us that we can start processing the data. If we encounter any other `IOException` (`EOFExcetion` is an `IOException` we might want to retrieve a backup file. If there is any other exception we don't know what to do. In this case we simple pop up a dialog box displaying the error message.

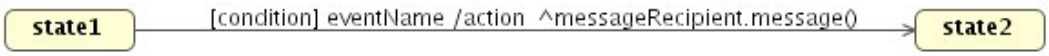
4.2.6. External versus internal events

External events are events passed between the system and its actors. They are used during the requirements specification phase as well as for high level design diagrams showing some interaction with external objects.

Internal events are simply events resembling some occurance at some time. They resemble an event caused and received by objects internal to the system and typically transport some information within the system.

4.2.7. Full signature for state transition label

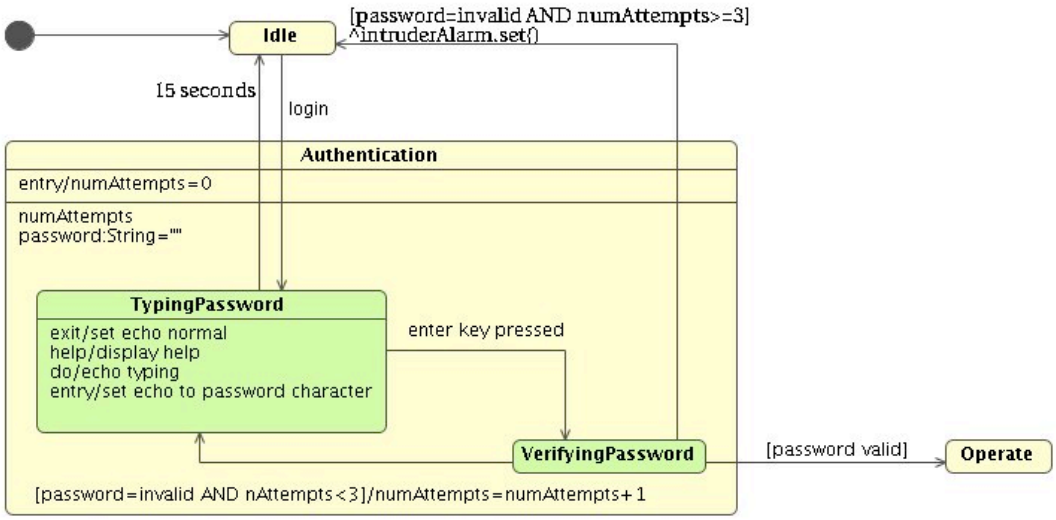
Figure 5.24. The complete UML signature for state transitions.



The complete event signature in UML is

```
[guard condition] eventName /actionUponStateTransition ^object.sendMessage()
```

Figure 5.25. A simple control system.



5. Communication diagrams

We have documented *abstract collaborations* which abstractly name a collaboration which will realize the use case.

We have then identified the responsibilities which need to be addressed and assigned them to core system components or core business units.

After that we looked at how these objects collaborate using sequence diagrams and activity diagrams.

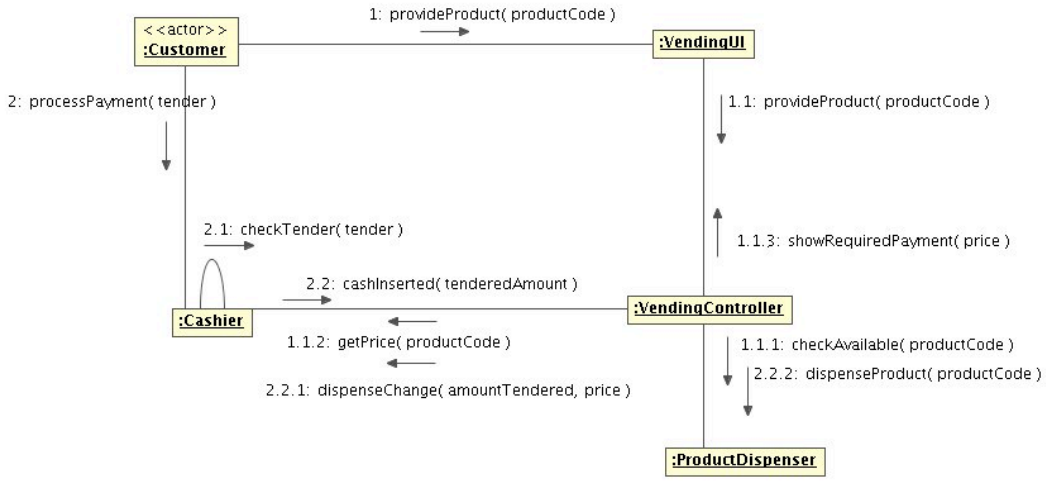
We now need a *convenient transition from the dynamic model to the static model*. This is where a *communication diagram is really useful*. It shows the message paths required for the objects to communicate as well as the messages sent along these message paths. The time-sequencing of messages is depicted via a numbering system.

5.1. Example: Communication diagram for a vending machine

Figure 5.26, “ Communication diagram showing how the core components collaborate to realize the buy-product use-case. ” shows a communication diagram corresponding to the sequence diagram shown earlier.

It shows the message paths required for the objects to communicate as well as the messages sent along these message paths. The time-sequencing of messages is depicted via a numbering system.

Figure 5.26. Communication diagram showing how the core components collaborate to realize the buy-product use-case.

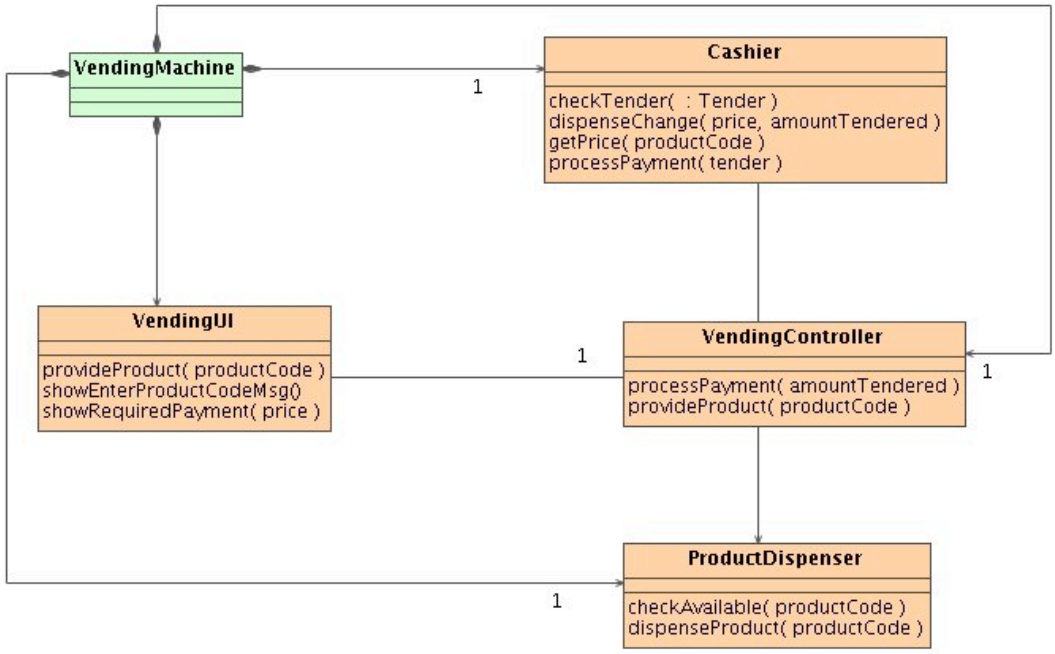


6. The context of the collaboration

The communication diagram now provides a very smooth transition to the static model at the level of abstraction as defined by the responsibilities identified early in the design phase.

The collaboration diagram shows the service request messages sent to objects. These objects must obviously provide these services. Furthermore, the message paths for these messages must exist between the objects. The mapping of this information onto the static model (i.e. the context of the collaboration) is shown in Figure 5.27, “The context of the collaboration”.

Figure 5.27. The context of the collaboration



The communication diagram thus directly leeds to

- the identification of the message paths required between the components at a particular level of granularity, and

- the services these components need to offer.

This information provides then the *static context of the collaboration, i.e. that subset of the static model at that level of granularity which is required to support the collaboration realizing the use case.*

Chapter 6. Deployment View

1. Introduction

We have modeled the business use cases, business processes and the business structure supporting the business processes as well as the business entities. We have, however, not yet looked at the environment in which we are going to deploy the business (or a particular business unit of an organization).

For example, we could decide to deploy the procurement arm of a global antique dealer in London, with retail outlets in New York, Cape Town, Sydney and Paris.

We need to model the environments in which the retail outlets are being deployed (for example, a shopping centre) as well as the physical realization of the integration paths (e.g. shipping organization between procurement and the retail outlets).

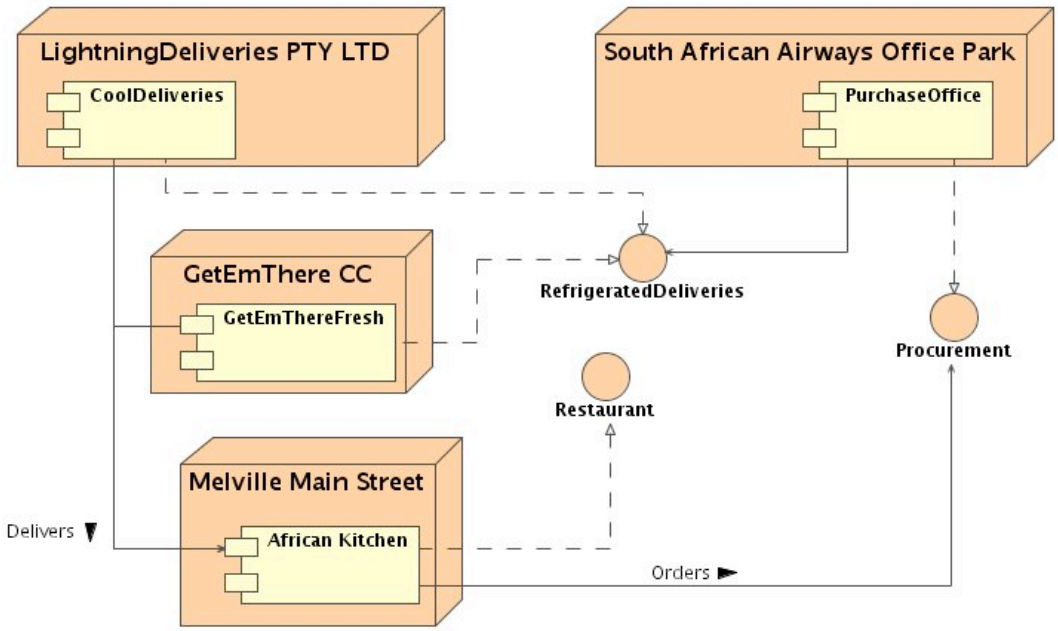
2. Showing deployment aspects

Not surprisingly, we will use UML deployment diagrams to specify the deployment environment of a business or a business unit, i.e. how the core components of a business, the core business units are deployed and what the physical realizations of the integration and communication channels are.

Here a node represents a deployment host. For example, a retail outlet could be deployed in a shopping centre. Than the node is the shopping centre.

Let us revisit our restaurant. Assume we are going to deploy the restaurant in Melville Main street while procurement is close to the airport (in the South African Airways Office Park) in order to manage the sourcing of the fresh seafood and other raw materials our restaurant requires (see Figure 6.1, " Deploying the restaurant and its procurement office. ").

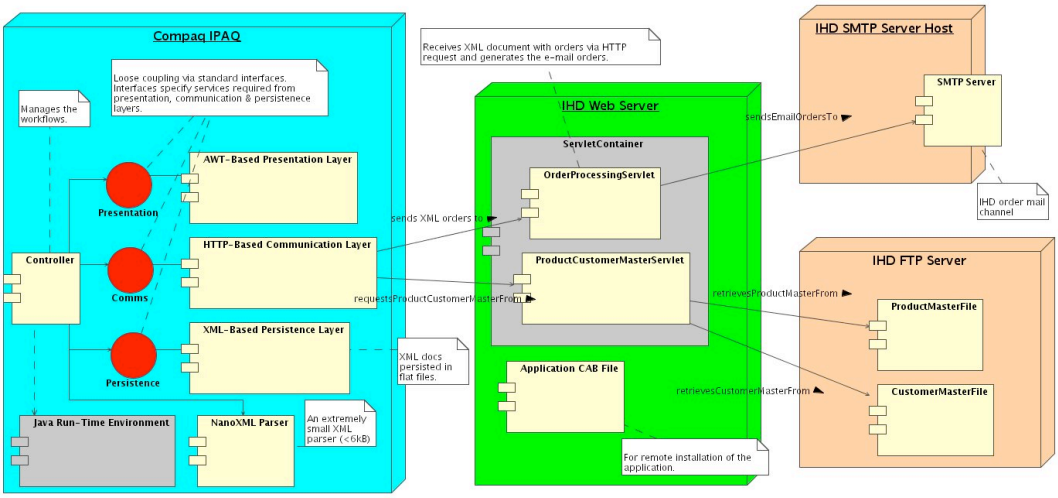
Figure 6.1. Deploying the restaurant and its procurement office.



Orders are placed with procurement via telephone and the orders are delivered via some or other refrigerated deliveries service. Both, *Lightning Deliveries* and *GetEmThere* have services, *CoolDeliveries* and *GetEmThereFresh* which provide the services specified in the interface *RefrigeratedDeliv-*

eries. Our business is decoupled from the physical realizations of that service and only binds via a standard contract defined in the *RefrigeratedDeliveries* interface.

Figure 6.2. Deploying for the MobileOrder PDA application.



Chapter 7. The Object Constraint Language (OCL)

1. Introduction

Defining the constraints in our UML diagram in informal text often leads to incomplete or imprecise constraints. Furthermore, such constraints are often open to mis-interpretation. In order to address this deficiency, UML has absorbed a formal constraint specification,the *Object Constraint Language* (OCL).

1.1. Where does the OCL come from?

The OCL was developed in 1995 by a team led by Jos Warmer and Steve Cook from IBM. This development was part of a business modeling initiative done by IBM. This is prior to UML, but IBM submitted a request to include the OCL as the official constraint specification language for UML and this was adopted by the OMG with UML version 1.1.

Since then the UML team has used the OCL extensively to rigorize UML's meta-model by formalizing UML rules and constraints.

1.2. Applications for OCL

Natural the OCL can be applied to anywhere where we need to formalize a constraint. Some common application include

- Formalizing guard conditions in UML sequence, activity and state diagrams. OCL can thus be used to specifying navigation rules through activity,state and sequence diagrams.
- Formalizing the requirements specification using pre- and post-conditions as well as invariants (in the context of *Design by Contract*).
- Specifying derived variables -- their value is constrained to be equal to some function of other variables.

1.3. Some core features of OCL

OCL was designed to be a very simple language which is sufficiently formal to enable the non-ambiguous declaration of constraints. Some of its core features are

- **OCL is a declarative language.** OCL expressions are without side effects. The evaluation of an OCL expression thus leaves the entity it applies to (its *context*) unchanged.
- **OCL provides syntax for navigating object graphs.** Since the OCL is an *object constraint language* the constraint applies to elements of object graphs. To effectively specify constraints the OCL supports syntax for navigating object graphs when constructing constraint expressions.
- **OCL is a strongly typed language.** All OCL operands apply only to predefined OCL types.
- **Each expression results in a value.** Applying an OCL expression to a context results in a value which can be used within the logic of a UML diagram or for further processing within OCL.

1.4. Types of constraints

The OCL defines 3 basic types of constraints derived from the *Design bu Contract* approach of Bertrand Meyer:

- **Invariants.** Class invariants specify constraints which must at all times be met by all instances of a class. One may specify more complex invariants in OCL which have to be evaluated across the object graph.
- **Preconditions.** Preconditions specify constraints which must be met before a service becomes available.
- **Postconditions.** Postconditions are constraints which must be met after a service has been completed.

1.5. The context of a constraint

The context is the entity to which the constraint applies. This may be an object (i.e. to any instance of a class to which the context refers) or a service (method) offered by instances of a class.

1.6. Constraint expressions

Ultimately a constraint expression must evaluate to either *true* or *false*.

2. Invariants

An invariant is a constraint which must hold at all times. The invariant is associated with a class and the invariant must hold at all times for all instances of that class.

For an invariant, we have to

1. specify the context to which the invariant applies followed by
2. the specification of one or more invariants.

The syntax is thus

```
context ContextClassName
inv: invarianceExpression
```

2.1. Positive balance constraint for savings accounts

Recall that we constrained the balance of savings accounts to be positive. We did this by applying a constraint to the specialization link. Let us now formalize that constraint using the OCL:

```
context SavingsAccount
inv: self.balance > 0
```

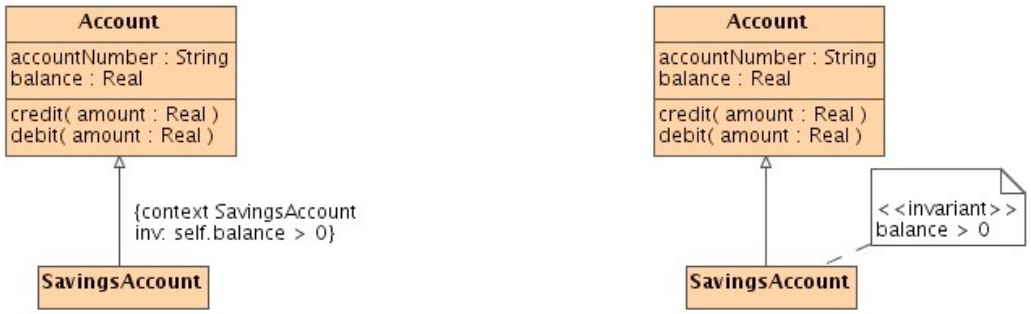
Here `self` is a self reference which is a reference to that instance of the context to which the constraint is being applied. We could have omitted the `self` since this is the default for OCL.

This constraint may be rendered by different UML tools in different ways, though they should all provide a mechanism to enter the formal OCL constraint. MagicDraw renders the constraint by default in OCL. The constraint is packaged with the UML element to which it is attached. In our case the constraint is attached to specialization link and rendered next to it (left diagram in Figure 7.1, “ The balance of savings accounts is constrained to be always positive. ”).

Other UML tools may render the constraint in a constraint comment with stereotype,

`<<invariant>>` as shown in the right diagram of Figure 7.1, " The balance of savings accounts is constrained to be always positive. ".

Figure 7.1. The balance of savings accounts is constrained to be always positive.



2.2. OCL operators

OCL provides a range of logical, relational,arithmetic and string operators which can be used in constraint expressions.

Table 7.1. OCL Logical Operators

operator usage	operand type	result type	description
`b1 or b2`	Boolean	Boolean	logical or
`b1 and b2`	Boolean	Boolean	logical or
`b1 xor b2`	Boolean	Boolean	logical exclusive-or
`not b1`	Boolean	Boolean	logical not
`b1 = b2`	Boolean	Boolean	logical equals
`b1 <> b2`	Boolean	Boolean	logical not-equal-to
`b1 implies b2`	Boolean	Boolean	logical implies
`if b1 then expr1 else expr2`	Boolean	type of expr1 or expr2	logical if

Table 7.2. OCL Relational Operators

operator usage	operand type	result type	description
`x1 = x2`	Integer or Real	Boolean	relational equals
`x1 <> x2`	Integer or Real	Boolean	relational not-equals

operator usage	operand type	result type	description
`x1 < x2`	Integer or Real	Boolean	relational less than
`x1 > x2`	Integer or Real	Boolean	relational greater than
`x1 <= x2`	Integer or Real	Boolean	relational less than or equal to
`x1 >= x2`	Integer or Real	Boolean	relational greater than or equal to

Table 7.3. OCL Arithmetic Operators

operator usage	operand type	result type	description
`x1 + x2`	Integer or Real	Integer or Real	addition
`x1 - x2`	Integer or Real	Integer or Real	subtraction
`x1 * x2`	Integer or Real	Integer or Real	multiplication
`x1 / x2`	Integer or Real	Real	division
`x1 div x2`	Integer	Integer	integer division
`x1 mod x2`	Integer	Integer	modulo (remainder)
`x1.abs`	Integer	Integer	absolute value
`x1.max(x2)`	Integer or Real	Integer or Real	maximum of `x1` and `x2`
`x1.min(x2)`	Integer or Real	Integer or Real	minimum of `x1` and `x2`
`x1.round`	Real	Integer	rounding
`x1.floor`	Real	Integer	the largest integer less than `x1`

Table 7.4. OCL String Operators

operator usage	result type	description
`s1.size`	Integer	length of string
`s1.toLower`	String	convert string to lower case
`s1.toUpper`	String	convert string to upper case

operator usage	result type	description
`s1.subString(i1,i2)`	String	sub-string
`s1 = s2`	Boolean	string comparison

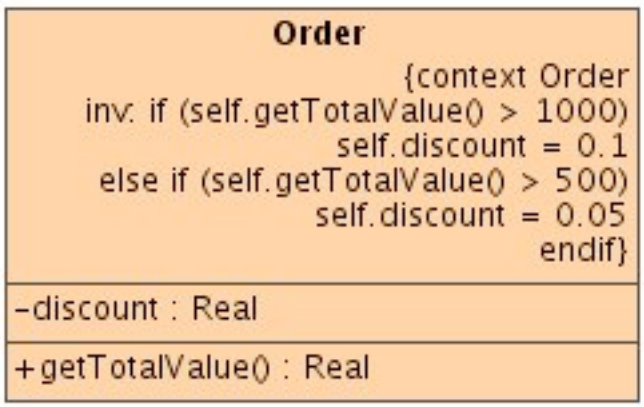
2.3. Conditionals and operations

OCL supports if statements for conditional constraints. Furthermore, a constraint may make use of the services available from the context (or from any other object in the object graph as we shall see shortly).

For example, assume the context of a constraint is the `Order` class with structure as shown in Figure 7.2, " Customers get a 5% discount for orders above R500.00 and a 10% discount on orders above R1000.00 "

```
context Order
inv: if (self.getTotalValue() > 1000)
            self.discount = 0.1
     else if (self.getTotalValue() > 500)
            self.discount = 0.05
     endif
```

Figure 7.2. Customers get a 5% discount for orders above R500.00 and a 10% discount on orders above R1000.00

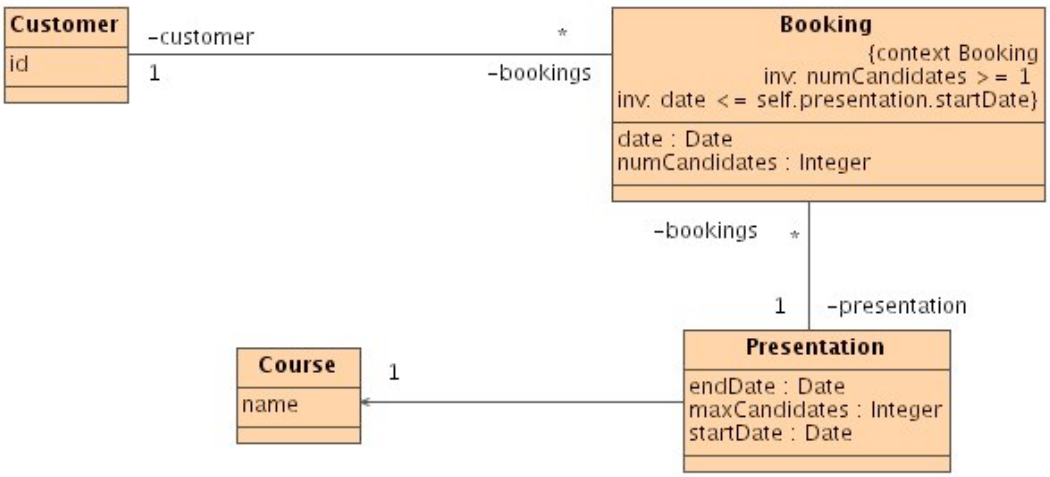


2.4. Navigating object graphs

We can navigate object graphs by simply using a dot for the element access operator. For example, in Figure 7.3, " Booking for a presentation must be done before the start of the presentation. " we apply a constraint which is enforced across elements in an object graph. In particular,we want to enforce that a booking for a presentation of a course must be made before the presentation starts. To apply this constraint to bookings we need to navigate from `Booking` to `Presentation`. This is done in Figure 7.3, " Booking for a presentation must be done before the start of the presentation. " using the OCL element access operator.

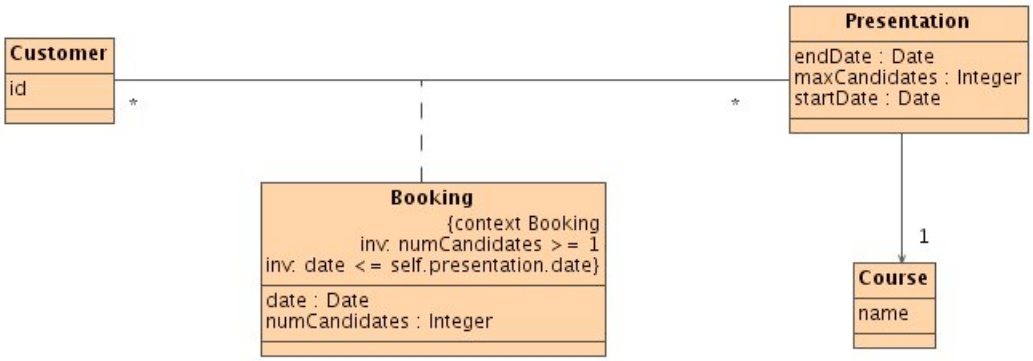
Figure 7.3. Booking for a presentation must be done before the start of the

presentation.



Note that we applied two invariance constraints to the context. A neater UML diagram which states the same is shown in Figure 7.4, " Using an association class for a booking. ". Here we use an association class which still provides the same navigation paths. t emphasis the fact that a booking is created every time a link between a customer and a presentation is instantiated.

Figure 7.4. Using an association class for a booking.

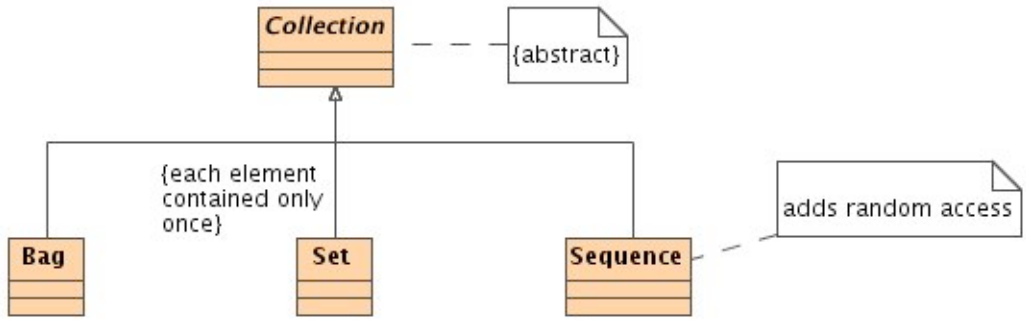


2.5. Constraints involving collections

In the case of one-to-many and many-to-many relationships we are faced with collections. OCL supports a range of collections operators which can be used in OCL constraints.

OCL supports the collections `Bag`, `Set` and `Sequence`. The commonalities between them (common operators) are encapsulated in the abstract supertype `Collection`.

Figure 7.5. Class hierarchy for OCL collections.



The three concrete collections are thus

- **Bag.** A bag is the most general concrete collection in OCL. It is an unordered collection of elements and may contain duplicate elements.
- **Set.** A sets is still an unordered collection, but it removes any duplicate elements -- it is a singleton collection.
- **Sequence.** A sequences is an ordered collections which provides random access and may contain duplicates.

OCL collection fields and methods are accessed via the arrow operator, `->`. For example, reconsider Figure 7.3, " Booking for a presentation must be done before the start of the presentation. ". If we wanted to limit the number of bookings to less than the maximum number of participants we could attach the following constraint to the context, `Presentation`:

```
context Presentation
inv: self.bookings->size <= maxCandidates
```

Here `size` an attribute provided by OCL for collections. It returns the number of elements in the collection to which it is applied. Below we discuss some of the more commonly used collection functions.

2.5.1. Collection operators

OCL provides a range of operators for collections. These are useful when specifying constraints which apply to the many side of a one-to-many or many-to-many relationship.

Table 7.5. OCL Collection Operators

operator usage	result type	description
`col1->size`	Integer	number of elements in the collection
`col1->isEmpty`	Boolean	return `true` if the collection is empty
`col1->notEmpty`	Boolean	return `true` if the collection is not empty

operator usage	result type	description
`col1->count(obj1)`	Integer	number of occurrences of object `obj1`
`col1->includes(obj1)`	Boolean	returns `true` if `obj1` is contained in `col1`
`col1->includes(col2)`	Boolean	returns `true` if all objects in the collection `col2` are contained in `col1`
`col1 = col2`	Boolean	returns `true` if the two collections have identical elements
`col1->union(col2)`	Collection	returns the union of the two collections
`col1->intersection(col2)`	Collection	returns the intersection of the two collections
`col1->including(obj1)`	Collection	returns a collection which includes the object `obj1`
`col1->excluding(obj1)`	Collection	returns a collection which excludes the object `obj1`
`col1->select(expr1)`	Collection	returns a sub-collection where all elements satisfy the expression `expr1`
`col1->collect(expr1)`	Collection	returns a collection where each element is the result of applying the expression `expr1` to the corresponding element in `col1`
`col1->exists(expr1)`	Collection	Boolean
`col1->forAll(expr1)`	Collection	Boolean
`col1->sum()`	Collection	Integer or Real

Table 7.6. OCL Bag Operators

operator usage	result type	description
`bag1->asSet()`	Set	returns a set for that bag (removing duplicate elements)
`bag1->asSequence()`	Sequence	returns a sequence collection for that bag

Table 7.7. OCL Set Operators

operator usage	result type	description
`set1 - set2`	Set	returns a set containing those elements of `set1` which are not contained in `set2`
`set1->symmetricDifference(set2)`	Set	returns a set contained in either of the two sets, but not in both
`set1->asBag()`	Bag	returns a bag for that set
`set1->asSequence()`	Sequence	returns a sequence for that set

Table 7.8. OCL Sequence Operators

operator usage	result type	description
`seq1->first`	Type of first element in sequence	returns first element in sequence `seq1`
`seq1->last`	Type of last element in sequence	returns last element in sequence `seq1`
`seq1->at(i1)`	Type of element at position `i1`	returns `i1`'th element in sequence`seq1`
`seq1->append(obj1)`	Sequence	returns sequence with object appended
`seq1->prepend(obj1)`	Sequence	returns sequence with object prepended
`seq1->asBag`	Bag	returns sequence as bag
`seq1->asSet`	Set	returns sequence as set

2.5.2. Iterating across a collection

The syntax of an OCL iteration expression is

```
collection->iterate(i: IteratorType, accumulator: AccumulatorType = initializat
   | expressionUpdating accumulator)
```

The previous constraint on the number of bookings is insufficient since a booking can be a booking of more than one candidate. We actually have to iterate over the bookings and some up the number of candidates and compare that to the maximum number of candidates. The resultant OCL constraint could be something like this:

```
context Presentation
inv: maxCandidates >= self.bookings->iterate(b: Booking; result: Real = 0
       | result + b.numCandidates)
```

2.5.3. Selecting a specific type of collection

If we want to avoid iterating over duplicate elements we can explicitly select a *Set* as the choice for the collection we use to specify the contraint in:

```
context Presentation
inv: maxCandidates <= self.bookings.asSet()->iterate(b: Booking;
          result: Real = 0  | result + b.numCandidates)
```

2.5.4. Collecting and summing across a collection

The previous example we could actually have formulated in a simpler way using the `collect` method available in OCL for collections. Using the `collect` method, the previous constraint looks like this:

```
context Presentation
inv: maxCandidates >= self.bookings->collect(numCandidates)->sum
```

in the above example the `collect` operator creates a new collection containing the selection (in our case we get a collection of the number of candidates enrolled in the various bookings) and then we apply the `sum` operator to that result collection and compare the result with the maximum number of candidates which can be accommodated in that presentation.

2.5.5. Selecting and rejecting elements from a collection

The OCL provides a `select` method for collections which extracts those elements from a collection which fulfill a supplied criterion. Assume, for example, that you want to enforce the business rule that in every presentation of a course there should be three bookings reserved for single-candidate bookings.

```
context Presentation
 inv: self.bookings->select(numCandidates>1)->collect(numCandidates)->sum
          <= self.maxCandidates-3
```

In OCL collections provide a corresponding `reject` method which is similar to the `select` service except that the matching selections are rejected from the result collection.

2.5.6. Testing an expression across all elements in a collection

OCL supports testing an expression across all elements of a collection. At times one wants to enforce that an expression is true for all the elements in the collection, while in other cases one simply want to cheque that an expression holds for at least one of the elements in the collection.

2.5.6.1. Enforcing an expression across all elements in a collection

The `forAll` method applies an expression to all elements in a collection. It returns `true` if and only if the expression holds for all elements in the collection.

Assume you want to apply the constraint to presentations that all bookings must be bookings for at least one candidate but for no more than the maximum number of candidates for the presentation. This can be done in OCL via

```
context Presentation
  inv: self.bookings->forAll(numCandidates>0 and numCandidates<self.maxCandidat
```

Here we also made use of the logical `and` operator.

3. Pre- and Postcondition in OCL

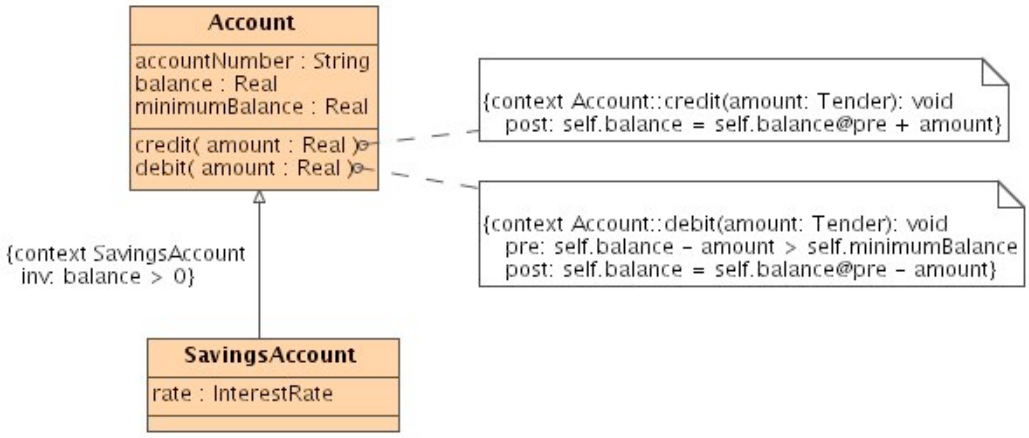
Pre- and postconditions are constraints which are applied to services offered by instances of classes (i.e. to operations). Preconditions are constraints which are checked before a service provider attempts to provide the service. Post-conditions are the constraints whichmust be satisfied upon successful completion of a service.

Pre- and postconditions are specified via the `pre:` and `post:` prefixes. The context must be a service offered by a class.

Example 7.1. Pre- and postconditions of the account services

In Figure 7.6, " Specifying pre- and postconditions for the credit and debit services. " we show the pre and post conditions which apply to the `credit` and `debit` services of accounts.

Figure 7.6. Specifying pre- and postconditions for the credit and debit services.



4. OCL and Testing

The specification of functional and system integrity tests is directly supported by OCL.

4.1. Functional testing

Any object which provides services to other objects is a service provider. It is desirable that the requirements of client objects on their service providers should be encapsulated within a *contract*

which consists of

- the interface through which the services are requested, i.e. a specification of the structure of the service request messages, and
- the pre- and postconditions for each service specifying

  1. the conditions which must hold before the service is available to the client and
  2. the deliverables upon successful completion of the service.

Instances of any class which realizes this contract can be plugged in as service providers.

The *unit test* for the service provider should typically test that if the preconditions as specified in the contract (i.e. on the interface) are satisfied the service is provided as per contract, i.e. that the all the deliverables are provided.

Note

The design-by-contract rules state that post-conditions may never be reduced and pre-conditions may never be increased during specialization.

Since pre-conditions may indeed be reduced we cannot test that the service is refused if the preconditions of the contract (as specified on the interface) are not met -- after all service providers may provide the service more generously. However, if we specify pre-conditions on the individual classes which realize that contract, then the unit test on the class should test that the service is refused if the pre-conditions for that particular service provider are not met.

4.2. System integrity testing

To test the system integrity, one tests the invariant constraints for all system components. As soon as any of these is violated, the system is said to be in an invalid state with an embedded system fault.

5. Exercises

1. Use the *Object Constraint Language* (OCL) to formalize the constraints for the e-commerce retailer you have been modeling.

Chapter 8. URDAD for System Design

Abstract

URDAD, Use-Case, Responsibility Driven Analysis and Design, provides a simple, intuitive design process which can be embedded within iterative, use-case driven software development methodologies like extreme programming or the Rational Unified Process (RUP). URDAD has been formulated in a way which encourages sound design principles. The process directly generates clean layers of granularity, with good responsibility localization across loosely components of minimal structural complexity. The resultant design is architecture and technology neutral. Following OMG's Model Driven Architecture (MDA) this platform independent design is mapped onto some choice of architecture and realization techonologies.

1. Introduction

Good design makes systems more flexible and robust, reduces maintenance cost and increases life expectancy. Yet, generating a "*good design*" is a non-trivial task. This article looks at core attributes of good design and explains how URDAD provides a simple algorithmic design process which leads to a design satisfying many of the requirements of a good design.

1.1. Design versus development processes

Software developers are phased with the problem of having to provide a design and ultimately an implementation which realizes the use case requirements. Software development processes like RUP (the Rational Unified Process) and Extreme Programming provide a higher level process for effectively managing the risk and quality of the development process. They do not, however, provide a design process. One still has to decide how to design and integrate the system components in order to obtain the desired functionality.

URDAD addresses this problem by providing a process for taking a use case through to realization. This design process can be plugged into higher level software development processes.

1.2. Background

URDAD has grown out of Responsibility Driven Design (RDD) methodology pioneered by Rebecca Wirfs-Brock and Brian Wilkerson (see [Wirfs-Brock-Wilkerson-1989], [Wirfs-Brock-Wilkerson-Wiener-1990] and [Wirfs-Brock-McKean-2002]). Like RDD, URDAD focuses during the early stages of the design on identifying and assigning responsibilities. Also, like RDD, URDAD puts a lot of emphasis on client-server contracts. However, unlike RDD, URDAD critically requires that responsibilities should be identified before one identifies the objects which will ultimately host the responsibilities. Furthermore, RDD does not provide a framework which generates the different layers of granularity naturally and cleanly. The ability to look at use case realization at different levels of granularity is a major benefit of URDAD. Finally, URDAD provides a step-for-step algorithm for designing a system across its levels of granularity.

Other methods like the *ICONIX* process from Doug Rosenberg discussed in [Rosenberg-Scott-1999] provide a structured process for evolving the static model from the collaboration requirements, but are not really responsibility driven, nor do they project out clean layers of granularity.

1.3. Design versus Architecture

URDAD is a design process. It does not address architecture. Modern approaches view architecture as orthogonal to design. While design realizes the functional (use case) requirements, architecture addresses non-functional requirements like scaleability, reliability, security, modifiability and so on. These are called the quality attributes of the architecture. It is the architecture which ensures that these qualities are realized across the various use cases of the system. Architecture will, for example, specify whether clustering should be used to achieve availability, whether reliability should be guaranteed with session replication, whether thread, component and resource connection pooling

should be used to improve performance and reduce resource demand, the choice of integration technologies and so on.

Technologies change quite frequently and a change in implementation technology should not require a change in design. URDAD generates a design which is architecture and technology neutral. This approach is aligned with OMG's *Model Driven Architecture* (MDA) -- see [Frankel-2003]. MDA requires that the design phase yields a *Platform Independent Model* (PIM) which ensures that design survives technologies and architectures. MDA then maps the PIM onto the chosen architecture and technologies resulting in the *Platform Specific Model* (PSM).

URDAD thus generates MDA's PIM. The PIM is then mapped onto the chosen architecture and technologies. This may be done manually or using MDA tools.

1.4. Requirements for good design

In order to be able to demonstrate that URDAD is a process which tends to lead to "*good design*", we first have to understand the core qualities of good design. Most of these are accepted design principles:

- **Responsibility localization.** A design with good responsibility localization is often referred to as a design with a high level of cohesion. In such a design each component adheres to the *single responsibility principle*; i.e. each component thus has only a single responsibility at some level of granularity and all its attributes and services are narrowly aligned with its responsibility.

- **Clean layers of granularity.** This very important aspect of good design enables one to work effectively at various levels of granularity. The layers should adhere to the dependency inversion principle, i.e. components in a lower level of granularity should not have any dependency on higher-level components. Also, one should be able to understand a higher-level workflow without having to understand the finer details. At any level of granularity the responsibilities should be well defined and the workflow should be self-contained and comprehensible.

  ## Note

  To illustrate the benefit of being able to understand a system at different levels of granularity, let us have a look at a car. A non-technical person can learn to effectively drive a car. This is only possible because they do not have to understand the details of the lower level functioning of the car. A mechanic in the local service station will have to understand the car at a lower level of granularity, but does not need to understand the finer details of how the gearbox works. Again, it would be difficult (and expensive) to source a mechanic who would be able to understand the functioning of the car at all levels of granularity. Finally a gearbox specialist will understand the system at an even lower level of granularity, without having to understand the higher-level workflows. The various levels of granularity facilitate that different role players can all function effectively without anybody having to understand the entire system.

- **Decoupling.** Decoupling provides a high level of flexibility and improves maintainability. If one component uses another component we effectively have a client-server relationship. Generally clients would not want to lock into a particular service provider. Instead, the client defines the requirements in a contract (in business modeling this would be an SLA). The contract specifies the services which service providers need to provide (the interface), the pre- and post-conditions for those services and the non-functional requirements.

- **Simplicity.** If everything else is equal, then the simpler solution is preferable. Complexity results in increased development costs, risk and maintenance costs. A design which is understandable and conceptually intuitive is preferable above one which is difficult to explain and non-intuitive.

- **Architecture and technology neutral.** The design should remain valuable over a long period. To this end the design should be able to survive technologies, changes in access mechanisms and architectural changes. This is usually achieved by following the guidelines of OMG's *Model Driven Architecture*, (MDA), which suggests that the core design should be technology and ar-

chitecture neutral and that this core design should then be mapped onto one's choice of technologies and architecture.

1.4.1. Benefits of adhering to these design principles

Adhering to the above design principles provides a range of short and long-term benefits to organizations including

- **Understandability.** Understandability is promoted by simplicity, good responsibility localization, intuitive naming, and the ability to view workflows at various levels of granularity.
- **Reusability.** Reusability is really a direct consequence of good responsibility localization together with a component based approach where components realize well defined contracts. Classes which address a particular combination of responsibilities relevant for a particular problem are not generally re-usable. On the other hand, classes whose services address only a single domain of responsibility and whose behaviour is well defined in a contract are generally much more likely to be re-usable.
- **Testability.** This is facilitated through specifying a contract for each component at any level of granularity.
- **Maintainability.** Simplicity, responsibility localization which results in localized maintenance, the ability to effectively work at different levels of granularity, decoupling, testability and reusability all contribute to making a system maintainable.
- **Longevity.** A design which is architecture and technology neutral can survive changes in technologies and architecture. Furthermore, all the design principles which assist maintainability contribute also significantly to the longevity of the design.

1.5. URDAD drivers

URDAD aims to provide a simple design process which leads to good design. To this end it uses the requirements for a good design as direct drivers for the process.

- **Use-Case driven.** Virtually all modern software development processes are iterative, use case driven processes. They deliver value incrementally to users and clients through iterative realization of use cases. Furthermore, use case driven approaches deliver iteratively testable deliverables. Suitable design methodologies must hence similarly be use case driven, realizing the functional requirements provided with the use case requirements.
- **Good responsibility localization by design.** At each level of granularity URDAD starts by identifying the responsibilities at that level of granularity before identifying objects. Each responsibility is then assigned to a separate object. This approach yields good responsibility localization by design.
- **Simplicity: minimized structural complexity.** In URDAD one does not start with the design with the static model (the class diagrams). Instead, after having identified (via the responsibilities) the core components which collaborate to realize the use case at that level of granularity, one first looks at how they collaborate (the dynamics). The static model required to support the dynamics realizing the use case is then projected out from the dynamic model. The only structural features thus generated are those required to realize the use case.
- **Clean layers of granularity.** In URDAD a level of granularity is fixed by the responsibilities. The only components for a particular level of granularity will be those to which the core responsibilities were assigned. URDAD projects out the workflow as well as the static structure at a fixed level of granularity. Finally, URDAD provides a simple mechanism for stepping from the current to the next lower level of granularity.

- **Decoupling via contracts at all levels of granularity.** URDAD requires responsibility identification followed by responsibility allocation to core system components and core external service providers (actors). For each responsibility, irrespective of whether the responsibility is by an internal component or an actor, URDAD requires a contract. Contract are specified along the guidelines provided by *Design by Contract*.

In URDAD re-usability is viewed as a consequence of responsibility localization and contract (interface) based decoupling.

2. URDAD: The Process

URDAD assumes that one is following an iterative software development process where one realizes use cases iteratively. We thus assume that one has selected the use case(s) for the current iteration and that the use case requirements are available.

2.1. Overview of URDAD

URDAD generates MDA's *Platform Independent Model* (PIM) which is then mapped onto the chosen architecture and technologies to yield the *Platform Specific Model* (PSM). It takes use-case based functional requirements through an iterative design process generating the various levels of granularity of the system iteratively.

Figure 8.1. Use-Case/Responsibility Driven Design

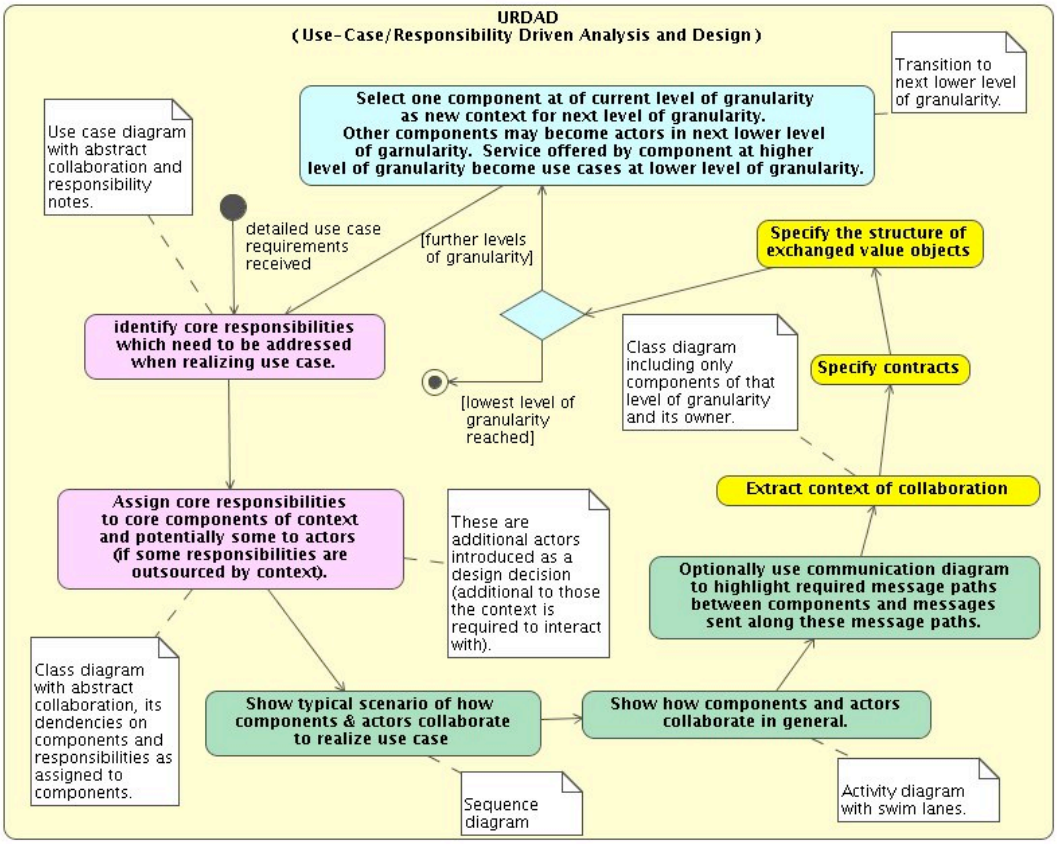


Figure 8.1, “Use-Case/Responsibility Driven Design” provides an overview of URDAD. The core steps of an iteration in URDAD are

1. Identify the core responsibilities which need to be addressed when realizing the use case.

2. Allocate each responsibility to either a component of the current context or an actor.

3. Specify how these components and actors collaborate to realize the use case.

4. Project out the context of the collaboration. This is that subset of the static model which at the current level of granularity is required to realize the use case.

5. Specify for each responsibility the contract they have to realize in the context of the current use case.

6. Specify the structure of exchanged value objects using class diagrams.

7. Traverse to the next lower level of granularity by selecting one of the components from the previous iteration as the new context with the services at the previous level of granularity becoming the use cases of this new, lower level of granularity.

8. Repeat the above steps for the use cases at the next lower level of granularity.

Note

URDAD is a double-iterative process with

1. *use case iterations* ensuring the system is designed iteratively, realizing use case after use case, and

2. *design iterations* taking the design iteratively through lower and lower levels of granularity.

In the following sections these steps will be explained in detail, using the design of a simple mail client as an example.

2.2. Responsibility Identification

The first step of URDAD focuses on identifying the responsibilities which need to be addressed when realizing the use case. It is in many respects the most critical and most difficult step. This step fixes the level of granularity for the current design iteration. To this end it is important that the responsibilities identified are at the same level of granularity (or the same level of abstraction).

Note

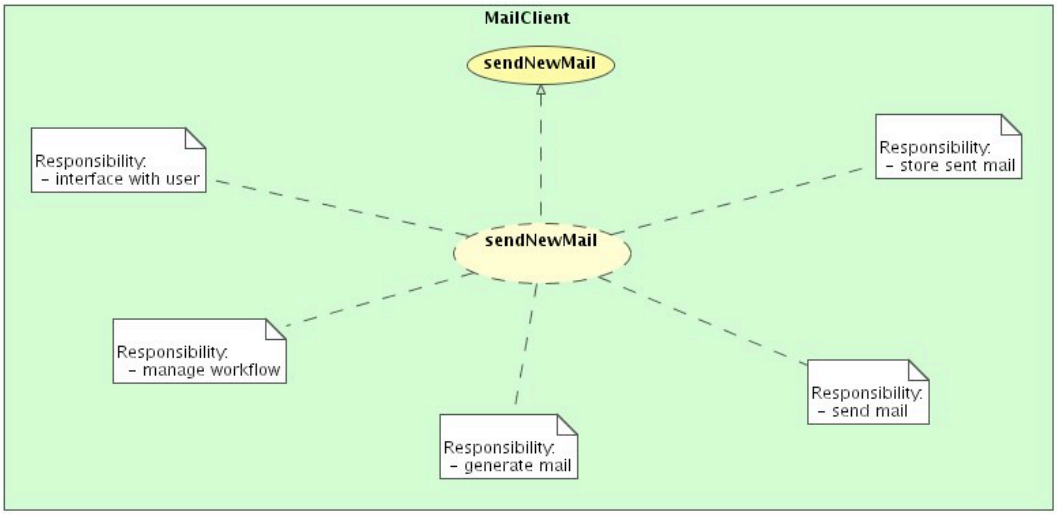
URDAD requires that the *responsibilities should be identified before identifying system components*. As a second step these responsibilities will be assigned to core system components, ensuring good responsibility localization across system components.

Consider, as an example, a simple mail client. Assume we want to realize the *send-new-mail* use case. This use case will be realized through the collaboration of certain system components and potentially some actors. In Figure 8.2, “Responsibility identification.” we identify the responsibilities which need to be addressed when realizing the use case. They include

- the workflow control and user interfacing responsibilities for the current context (the mail client as a whole),

- and the functional responsibilities for the mail client including that of generating the e-mail, sending it and storing the sent mail.

Figure 8.2. Responsibility identification.

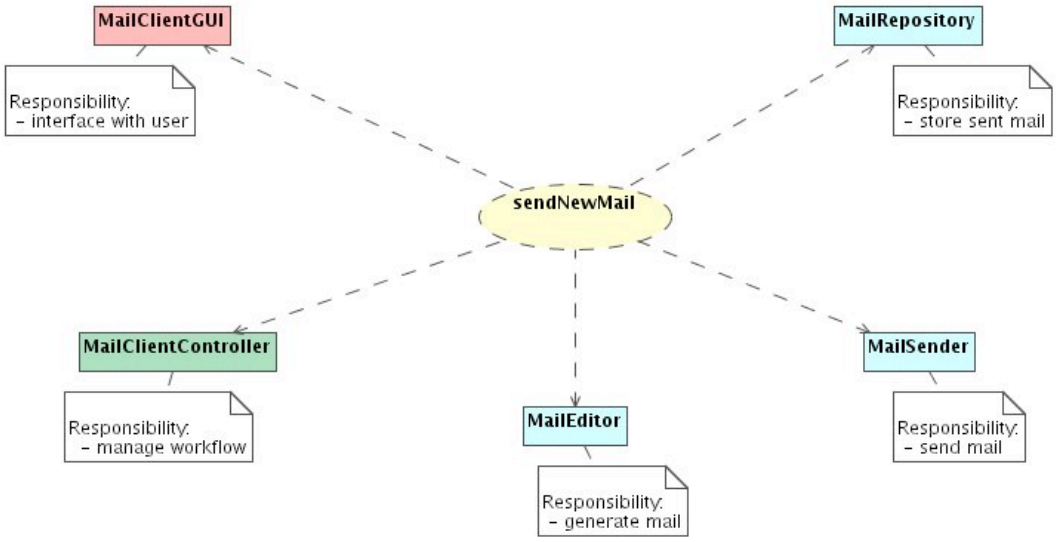


Responsibilities like that of managing addresses or marshaling the message on to the SMTP protocol are not relevant at this highest level of granularity. The responsibility of selecting addresses for an e-mail will be addressed in the context of creating the mail object and that of marshaling the mail onto the SMTP protocol will be addressed in the context of sending the e-mail. Hence both these responsibilities are lower level responsibilities which will be addressed at a lower level of granularity.

2.3. Responsibility Allocation

The second step of URDAD requires that each responsibility is assigned either to a separate system component or to an actor. URDAD thus enforces single responsibility principle by design. The objects thus introduced will all be at the same level of granularity as fixed by the responsibilities identified in the first step.

Figure 8.3. Responsibility allocation.



Revisiting our mail client we could potentially assign the responsibilities as illustrated in Figure 8.3, “Responsibility allocation.”.

2.4. Specifying the collaboration

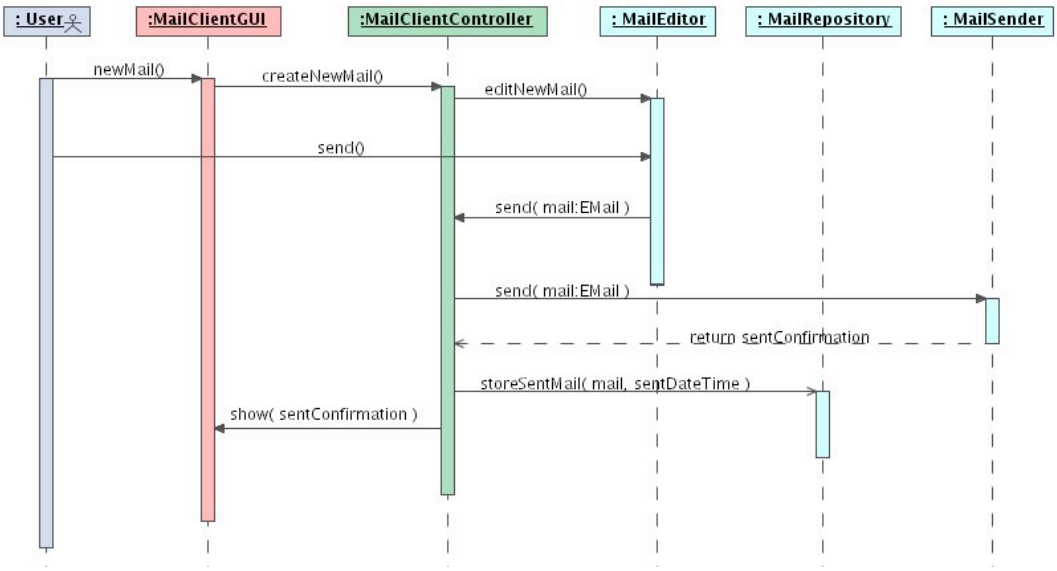
Having

1. identified the responsibilities which need to be addressed when realizing the use case, and
2. assigned these responsibilities to core system components and external service providers (actors),

we now need to look at how these components and actors collaborate to realize the use case.

Usually one first looks at a particular example (scenario) of realizing the use case. To this end one generally starts with a sequence diagram. Figure 8.4, “A scenario of realizing a use case at a specific level of granularity.” shows an example of such a sequence diagram.

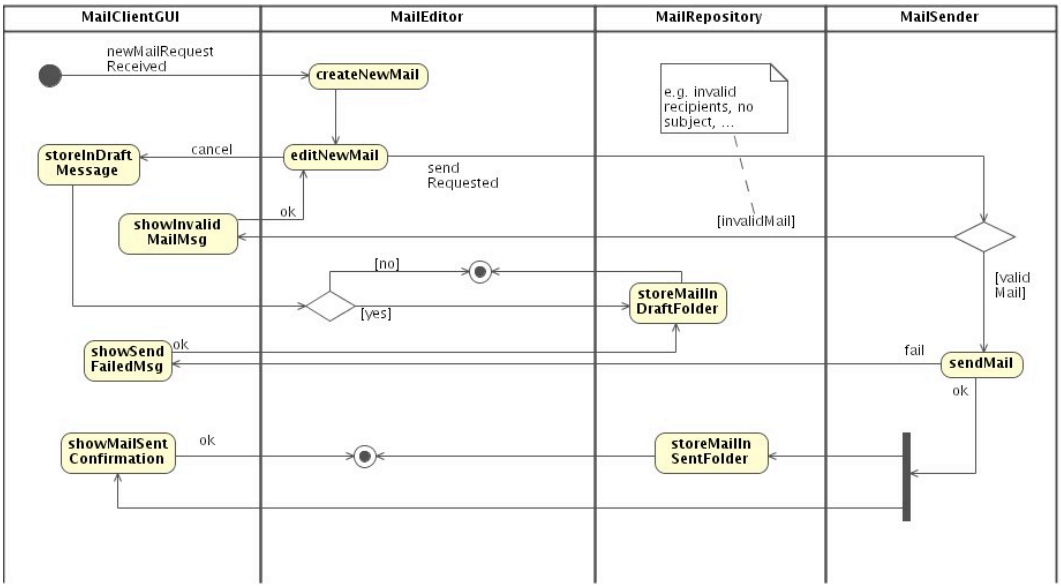
Figure 8.4. A scenario of realizing a use case at a specific level of granularity.



Note that the only objects participating in the collaboration shown in the sequence diagram are those which address the core responsibilities as identified for this level of granularity.

Once one is comfortable with a particular scenario (often a typical success scenario is chosen), one can look at the collaboration in general. This is commonly documented using a UML activity diagram. In Figure 8.5, “The use case collaboration in general.” we show an activity diagram documenting the general *send new mail* collaboration at the current level of granularity as fixed by the initial responsibility identification step.

Figure 8.5. The use case collaboration in general.

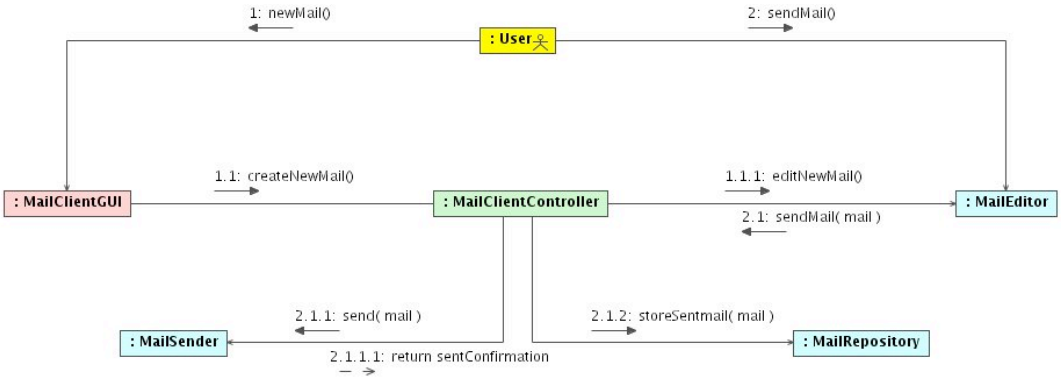


Finally, to simplify the transition to the collaboration context, one can use a UML communication diagram. It highlights the required communication paths as well as the service request messages sent along these paths and may contain other information from the static model. URDAD refrains from adding further structural information at this stage. The communication diagram corresponding to the sequence diagram in Figure 8.4, “A scenario of realizing a use case at a specific level of granularity.” is shown in Figure 8.6, “Communication diagram simplifying transition to collaboration context.”.

Note

Note that since URDAD only feeds message path information into communication diagram, the latter contains essentially the same information as the sequence diagram and may be auto-generated from the latter. This step may be (and often is) omitted.

Figure 8.6. Communication diagram simplifying transition to collaboration context.

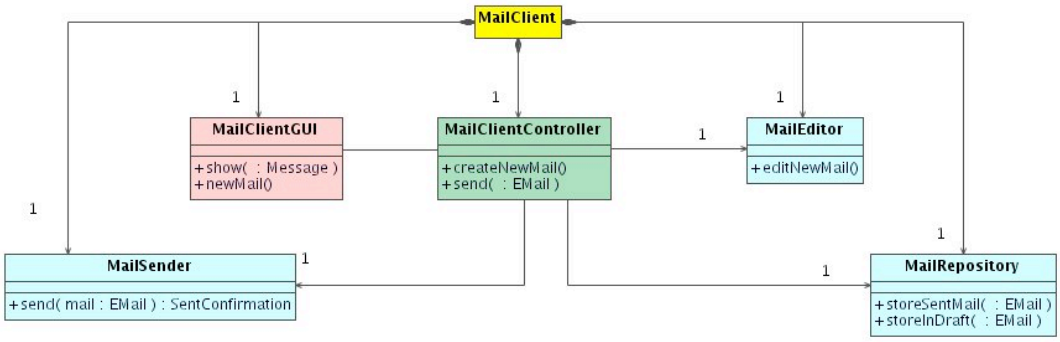


2.5. Projecting out the context of the collaboration

The collaboration shows the services the role players request from each other in the context of realizing the use case as well as the message paths required between them. URDAD now generates the context of the collaboration by projecting out the static structure required at the current level of granularity to realize the use case. This is a very simple step and the resultant class diagram is

shown in Figure 8.7, “The context of the collaboration.”.

Figure 8.7. The context of the collaboration.



The context of the collaboration is thus that subset of the static model which, at the current level of granularity, is required to realize the use case. The objects from that level of granularity are peers and hence the relationships between them will be associations (client-server) and not aggregation or composition relationships (otherwise the objects would not all be at the same level of granularity). However, some of these are components of the higher level context, filling in composition relationships between layers of granularity.

Note that unlike many other design methodologies (see for example [Ben-Abdallah-et.al-2004]) which go from the use case model to the static model (often via an object dictionary), URDAD defines the dynamics realizing the use case first. Only then is the required static structure identified. URDAD thus generates minimal structural complexity, i.e. only those structural features which are actually required to realize the use case.

Note also that the objects and classes generated are all at the same level of granularity. They are all either components of the use case context or actors. We have explicitly refrained from introducing any lower level classes at this stage. Instead URDAD provides a simple approach for going over to the next lower level of granularity.

2.6. Service provider contracts

Like many other modern design approaches, URDAD is also contract centric. For each component at any level of granularity one first specifies the contract. After all the difference between a class and a component is that the latter realizes a contract as specified by an interface with pre- and post conditions, invariance constraints and quality requirements. Contracts facilitate not only pluggability but also testability -- what would you be testing if there was no contract?

Note

Tests should be written for contracts (interfaces), not for classes. Any service provider claiming to realize a specific contract should be tested against the test for that contract.

URDAD requires that for each responsibility there should be a contract against which all service providers (components) which realize that responsibility should be tested. The contract may have functional aspects and non-functional aspects. The functional aspects are defined within the standard design-by-contract framework (see [Meyer-1991] and [Meyer-1992]) by an interface with pre- and post-conditions on the services and, if applicable, invariance constraints on the service provider itself. Contracts are developed from the perspective of the client. They thus resemble the “signed contracts” of Andreas Rausch (see [Rausch-2002]).

The non-functional aspects may include features like scaleability, usability, reliability, security, and so on. These are typically specified in a quality requirements note.

2.6.1. Pre-Conditions

If any of the pre-conditions is not met, the service provider is entitled to refuse the service without breaking the contract. On the other hand, if all preconditions are met, the service provider is obliged to provide the service. Otherwise it is a breach of contract and hence a failure. For example, for the debit service of an account there may be the pre-condition that there must be sufficient funds in the account. If there are insufficient funds the account may refuse the service without breaking the contract.

In software systems service providers use exceptions to notify clients that a requested service is refused due to a pre-condition violation.

2.6.2. Post-Conditions

The post-conditions are the deliverables of the service provider. These include the return value, but may also include service provider state information.

For example, the post-condition of debiting an account may include the requirement that the transaction must have been entered into the account's transaction history.

2.6.3. Invariance constraints

These are symmetry rules around the service provider's state. If at any stage (or at least on transactional boundaries) any of the invariance constraints are not met, then the object and hence the system is in an invalid state.

For example, for an account the invariance constraint could be that the sum of all credits minus the sum of all debits must always yield the current balance. If at any stage (at least on transactional boundaries) this symmetry does not hold, then the account and hence the system is in failure.

2.6.4. Quality requirements

From design by contract we know that the interface together with the pre- and post-conditions and invariance constraints provide a complete functional requirements specification for a service provider. In addition to the functional requirements, service provider may also need to adhere to certain non-functional (quality) requirements like scaleability, reliability or security requirements. Adding these quality requirements to the functional requirements completes the contract.

For our `MailSender` component we could require auditability as a non-functional requirement. The latter may be facilitated through logging all send requests together with their completion status.

2.6.5. Contracts and testing

Tests should be developed for contracts, not for individual service providers. The tests need to test both the functional, as well as the non-functional aspects of a contract.

During functional testing one tests that, if all preconditions are met, the service is provided such that

1. the client obtains the correct return value,
2. all post-conditions are met, and
3. if the invariance constraints were met prior to the service having been requested, that they are still met after the service has been provided.

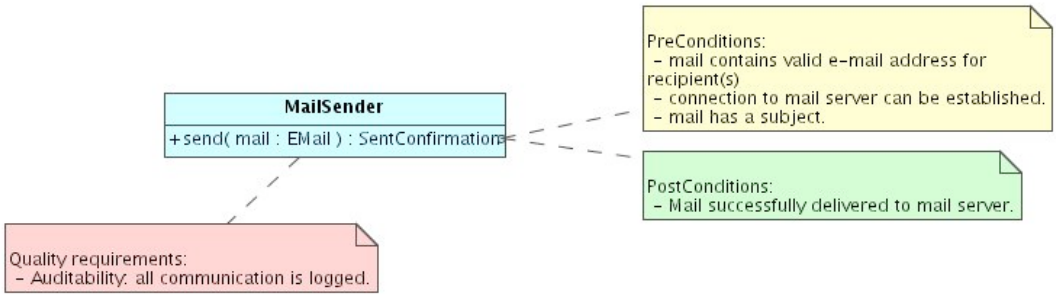
2.6.6. Contracts and the Object Constraint Language

UML enables one to specify the contracts formally in OCL, UML's *Object Constraint Language*. Doing this facilitates the automatic generation functional and system integrity tests.

2.6.7. MailSender contract

Figure 8.8, “Contracts are specified for each responsibility and hence for each service provider.” shows an informal contract for the `MailSender` component. It shows the required services together with the pre- and post-conditions for them as well as the non-functional auditability requirement. The contract could be formalized using the OCL.

Figure 8.8. Contracts are specified for each responsibility and hence for each service provider.



2.7. Value objects

One still needs to specify the structure of any object exchanged between system components or between system components and actors, the so called “value objects”. If we look at our mail client, then we see that an instance of an `EMail` is sent from the `MailClientController` to the `MailSender` and that the latter returns an instance of a `SentConfirmation`. We need to specify the structure of these value objects. This is naturally done using UML class diagrams.

Figure 8.9. Class diagram for the e-mail value object.

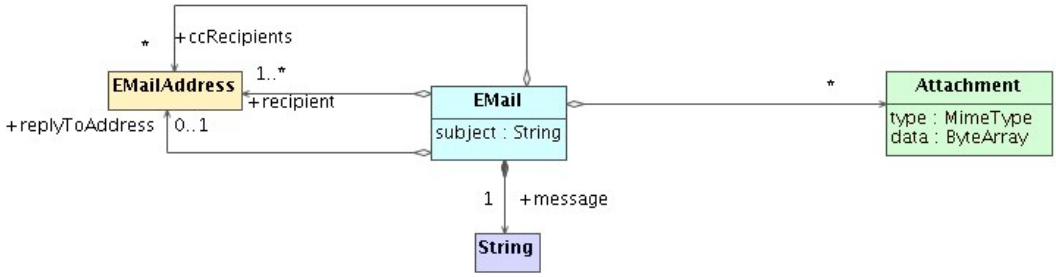


Figure 8.9, “Class diagram for the e-mail value object.” shows a simplified class diagram for the e-mail value object exchanged between the `MailEditor`, `MailClientController` and `MailSender`.

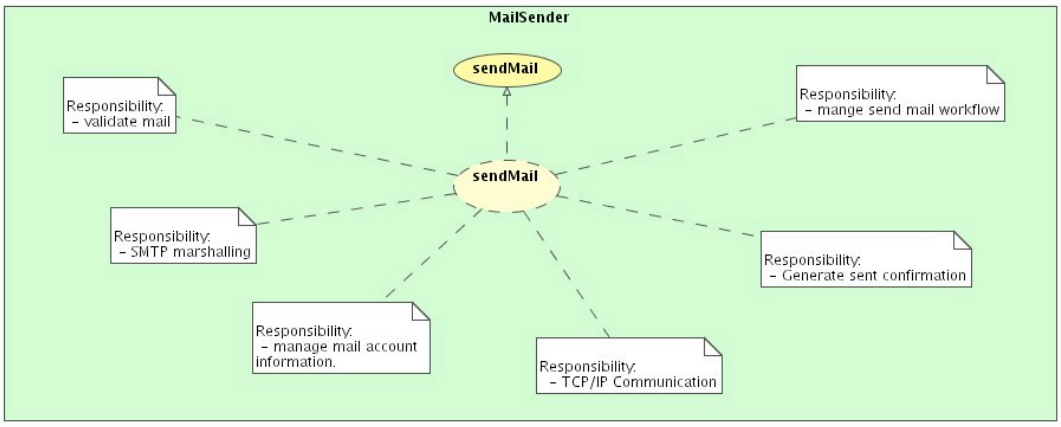
2.8. Transition to the next level of granularity

In order to go over to the next level of granularity, one selects one of the components from the current level of granularity as the context for the next level of granularity. Its services at the current level of granularity will become the use cases of the next level of granularity. Those components which interface with that object at the current level of granularity will become the actors. For example, if we select the `MailSender` as our new context, the corresponding use case diagram would be given by Figure 8.10, “Use case diagram for a component at the next lower level of granularity.”, i.e. the `MailSender` is used by the `MailClientController` to send e-mails.

Figure 8.10. Use case diagram for a component at the next lower level of granularity.

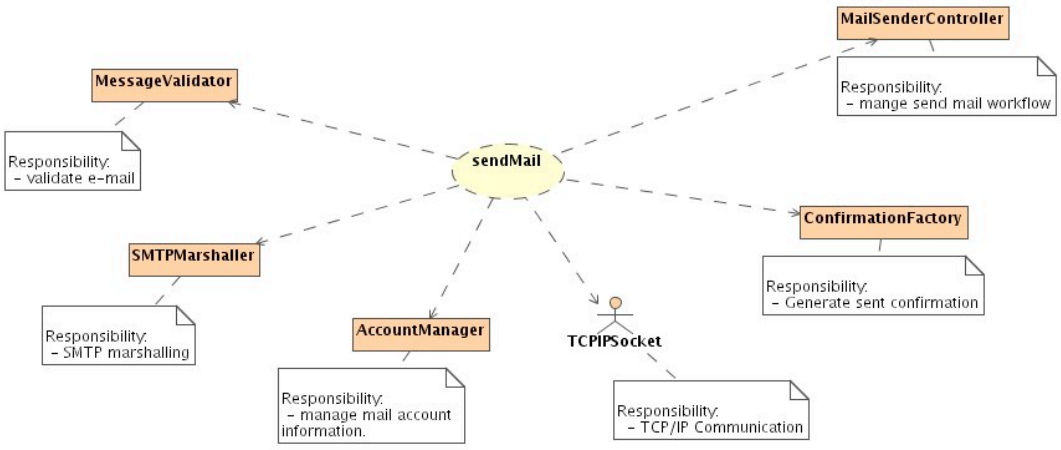
Of course the object may choose to realize the use case in a way which makes use of further actors. This should, however, be responsibility driven. We thus first identify the responsibilities at this new, lower level of granularity and before assigning them to components and/or actors of this lower level object as shown in figure Figure 8.11, “Responsibility identification at next lower level of granularity.”.

Figure 8.11. Responsibility identification at next lower level of granularity.



Each responsibility is, once again, assigned to either a component of this lower level component (this is where the composition relationships are identified) or an actor. This is shown in Figure 8.12, “Responsibility allocation at next lower level of granularity.”

Figure 8.12. Responsibility allocation at next lower level of granularity.



Note that we have a workflow controller at each level of granularity taking over the responsibility of managing the workflow for the use case at that level of granularity Similarly, if the user interfaces directly with the lower level object (as, for example, the `MailEditor` would), then there would also be the responsibility of interfacing with the user.

URDAD is thus an iterative design process which projects out one level of granularity after another. Typically one will require between 2 and 4 levels of granularity depending on the complexity of the system.

3. Summary and conclusions

URDAD provides a simple algorithm for designing a use case realization. It projects out different levels of granularity. At each level of granularity one starts with responsibility identification followed by responsibility allocation. Once one has established the objects which take care of the responsibilities which need to be addressed when realizing a use case, one specifies how they collaborate. From the collaboration one can project out the collaboration context which is that subset of the static model which, at the current level of granularity, is required to realize the use case. URDAD then provides a simple mechanism for going from the current level of granularity to the next lower level of granularity. This is done by selecting one of the components from the higher level of granularity as new context. Its services at the higher level of granularity become the use cases at the lower level of granularity. Some of the components of the previous level of granularity may become actors at this lower level of granularity. The process continues symmetrically by identifying and assigning responsibilities at the new lower level of granularity to components of the new context and external service providers (actors). The collaboration realizing the lower level use cases project out the static model at this lower level of granularity.

URDAD encourages *"good design"* by generating clean layers of granularity with minimal structural complexity, enforcing responsibility localization and contracts across system components.

Finally, URDAD provides a design process which follows the spirit of OMG's Model Driven Architecture (MDA) by generating a platform independent model (PIM). The PIM is then mapped onto some choice of architecture and technologies resulting in the platform specific model (PSM) which is ultimately mapped onto a realization resulting in the Enterprise Deployment Model (EDM).

Chapter 9. Design Patterns

Design patterns were the first patterns which found widespread use in the software development field. Today it is virtually expected of any software developer to have at least a rudimentary background in design patterns.

1. History of design patterns

It is difficult to pinpoint the origins of the use of patterns in society because, in many ways, the identification of generic solutions has been with us for a very long time. Here we focus on publications which have explicitly and consciously used patterns.

1.1. The classical design patterns book

The book which has pushed design patterns in the public spot-light is the classical design patterns book from Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides [Gamma-Helm-Johnson-Vlissides-1995]. This book discusses 23 design patterns which we call the classical design patterns. Since it is published in 1995 it does not yet use the UML for the OO diagrams and the example implementations are in C++. Nevertheless, this book remains invaluable as a generic, solid introduction to design patterns.

Gamma et. al already emphasized the following aspects

- that they resemble a generic solution to a wide range of problems,
- that one should program to an interface, not an implementation,
- that patterns can assist in designing for change,
- that patterns are related to frameworks in the sense that a framework (for example a user interface framework) is meant to be generic and used in a wide range of problems.

1.2. Patters in architecture

The origins of patterns can, however, be traced to at least 2 decades before the classical design patterns book was published. Christopher Alexander identified patterns in architecture of buildings and design of towns and published his ideas in two books, "*A Pattern Language*" ??? and "*A Timeless Way of Building*" [Alexander-1979]. Christopher Alexander describes a pattern as

> *A generic solution to a given system of forces in the world.*

1.3. Early patterns in software development

But, even design patterns were used prior to the classical design patterns book. For example, the *Model-View-Controller* pattern which was introduced with the first object-oriented programming language, *Smalltalk*, was published in 1988 [Krasner-Pope-1988].

2. Introduction

Design patterns were the first patterns which found widespread use in the software development field. Today it is virtually expected of any software developer to have at least a rudimentary background in design patterns.

Design patterns are *generic, re-usable design solutions*. These simple design components can often

- simplify your design considerably,
- improve system functionality, and
- make systems more flexible and more maintainable.

Many of these design patterns are extremely simple and, in hind sight, you will often be tempted to thing “of course”. However, without having seen some of these simple and obvious solutions, you may have potentially ended up with a solution which is significantly more complex and less flexible.

3. Benefits of patterns

The use of patterns typically

- simplifies the solution to a problem,
- makes the resultant solution more flexible,
- and hence more maintenance friendly.

Having had exposure to a collection of generic patterns enables one to identify patterns in problems and come up with a solution more readily. The solution is often also more elegant and powerful. In hind-sight one often finds that the solution is so obvious. However, not having had exposure to patterns may result in many cases in a more complex and less generic solution.

Once patterns have been used in a design, the design can be explained in a simpler and more compact way -- at least to people who have had exposure to patterns.

4. Discovering patterns

Patterns are not typically designed. In the context of a particular problem one comes up with a design solution. When working on another problem one may face some similar aspects and the correlation may lead to a similar design solution.

Over time the generic core is identified. This core is usually a simple generic component of the specific solutions for the various problems and this can be regarded as a *pattern*.

5. Documenting Patterns

Over the years standards for documenting patterns have become more important. The classical design patterns book by the so-called gang-of-four already uses a documentation template for documenting patterns. Most of the pattern documentation templates are variations of this documentation template and we, too, shall follow this approach.

5.1. Documentation template

We suggest the following template for documenting patterns. It contains the elements suggested by GOF and includes elements used in many pattern repositories (e.g. the Portland, Sun and TheServerSide repositories):

- **Name:** Every patterns should have a name. The following criteria should typically be applied to naming a pattern

    - The name should uniquely identify the pattern within your pattern repository.
    - Standard names should be used when the pattern is documented in public pattern resources.
    - The name should be descriptive.
    - If the pattern is known under different names add a *Also-Known-As* sub-section.

- **Intent:** The type of problem the pattern aims to solve.
- **Solution:** The design which solves the problem. For this one typically uses UML diagrams including:

    - Sequence, activity and collaboration diagrams showing how objects which participate in a pattern collaborate to realize the solution.
    - Class diagrams showing the static structure required by the pattern including

        - attributes and services required from the objects participating in the pattern,
        - message paths between objects including associations, aggregation and composition relationships,
        - specialization relationships and interfaces
        - and other dependencies.

- **Consequences:** The consequences of using the patterns including

    - the benefits provided by the pattern,
    - the trade-offs made when using the pattern, and
    - any issues introduced by using the pattern.

- **Example applications:** Some example problems which have been solved by using the pattern.
- **Implementation Guidelines:** List any issues implementors should be aware of including

    - Potential pitfalls.
    - Special techniques for implementing elements of the pattern.
    - Implementation issues (e.g. for different programming languages or different technologies).

- **Related Patterns:** List any related patterns including

    - Alternatives you should consider.
    - The elements which differentiate this pattern from the alternatives.
    - Any patterns often used in conjunction with this pattern.

6. Pattern repositories

There are many design patterns repositories on the web. Two of the most widely used generic patterns repositories are

- `http://hillside.net` and
- the Portland pattern repository, `http://c2.com/cgi/wiki` .

Other pattern repositories which you may find useful include

- the *Object-Oriented Pattern Digest* at `http://patternDigest.com`
- as well as the pattern repository hosted by *TheServerSide* at `http://www.theserverside.com/patterns`.

Having identified a new pattern,you may want to submit the pattern to one or more of these repositories. Additionally you may want to consider hosting a design pattern repository within your own organization, collecting patterns particularly useful in your domain.

7. The composite pattern

The composite pattern is one of the easiest patterns to understand and has very general applicability starting from your file system to portfolios, document structures, part catalogs and many other structures.

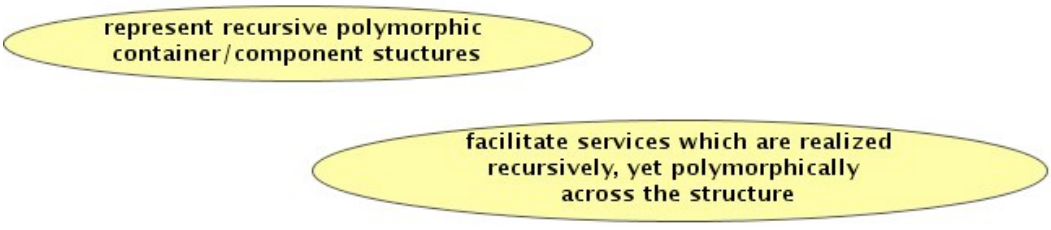
7.1. Intent

Typically there are two core intents when using the compsite pattern:

1. To represent tree structures supporting recursive part-whole relationships.
2. To facilitate services which are realized recursively through the structure.

These can be shown in a generic use-case view for the pattern:

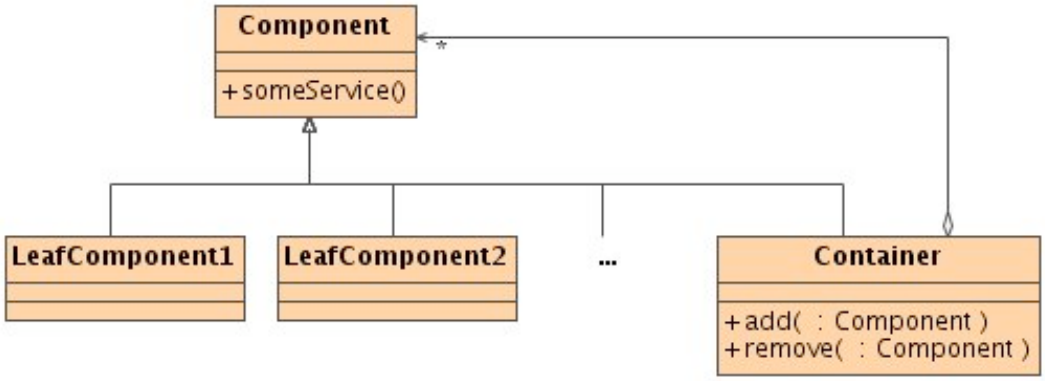
Figure 9.1. Use-case view of the composite pattern



7.2. Structure

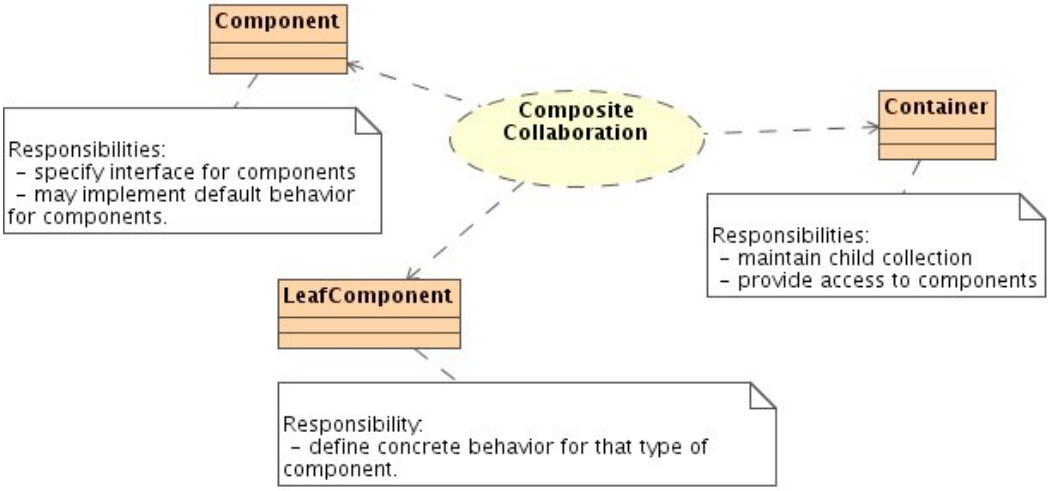
Containers have many components, but is itself a component. Hence containers may may hold as components further containers. The structure is shown in Figure 9.2, “ Structure of the composite pattern. ”.

Figure 9.2. Structure of the composite pattern.



The core objects of the composite pattern are thus the container, the root component and the leaf components. The responsibilities of these objects within the composite pattern are shown in the abstract collaboration depicted in Figure 9.3, “ Responsibilities of the components of the composite pattern. ”.

Figure 9.3. Responsibilities of the components of the composite pattern.



Typycally all components in the recursive structure provide at least one comnmon generic service which usually is realized polymorphically. The benefit of using the composite pattern lies thus not only in providing a recursive polymorphic structure, but also inpotentially providing services which are realized polymorphically across the recursive structure. We shall see examples of this in Section 7.3, “Example applications”.

7.3. Example applications

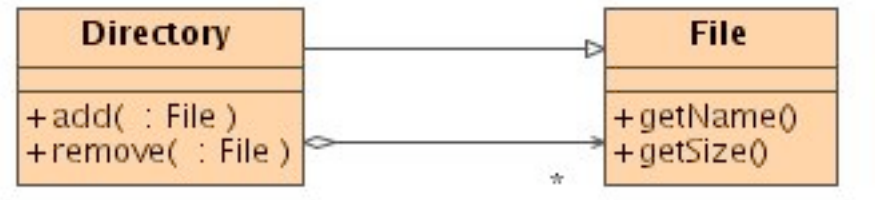
The composite pattern can be applied to a wide spectrum of fundamentally different problem domains.

7.3.1. File system

Most file systems support a recursive container/component stucture where the containers are directories and components are files. A directory contains many files some of which are again directories.

This structure is shown in Figure 9.4, “ A directory has files, but is itself a file. ”.

Figure 9.4. A directory has files, but is itself a file.



7.3.1.1. Design of the Virtual File System

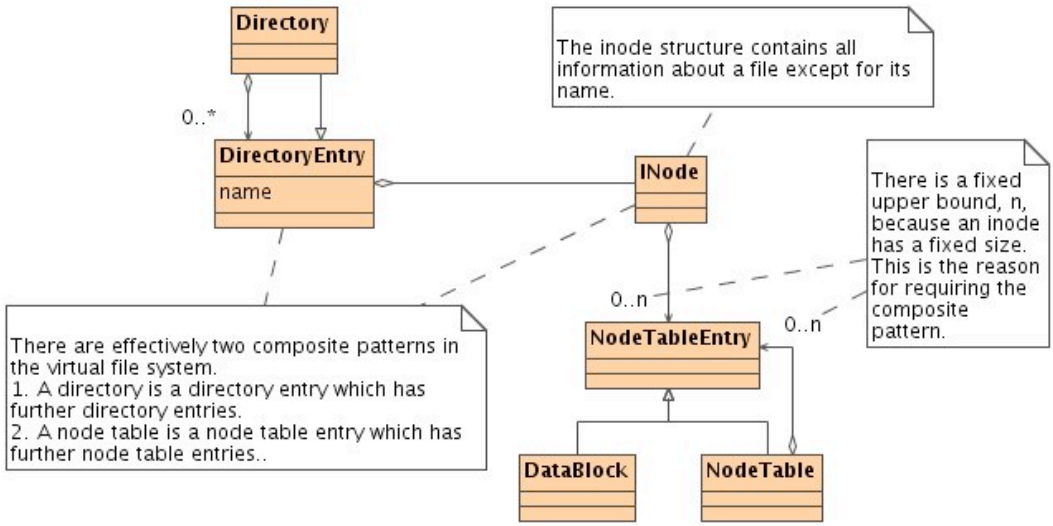
The *virtual file system* of Linux decouples the directory structure from the files themselves, i.e. a directory entry contains a file name and a pointer to an actual file. This design facilitates multiple directory entries for the same file. The design uses two composite patterns,

- one for the directory structure and
- one for the node tables references from the *inode*s.

The design is shown in the UML diagram of Figure 9.5, “The design of the Linux's virtual file system”.

> **Note**
>
> The design makes use of the classical bridge pattern to introduce an abstraction layer which isolates file system users from the concrete realization of the file system.

Figure 9.5. The design of the Linux's virtual file system

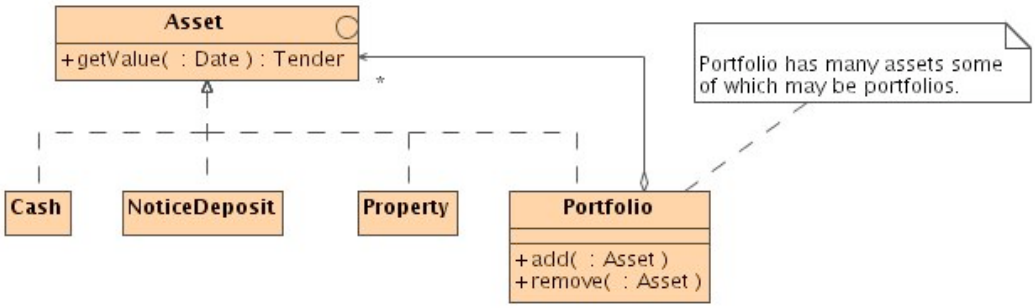


7.3.2. Portfolios and assets

Assets are entities which have value. Portfolios are groupings of assets. But the portfolio itself can

be viewed (and traded) as an asset and hence it is itself an asset. This very natural design for this is a direct application of the composite pattern (see Figure 9.6, “ A portfolio holds asset and can be itself viewed as an asset. ”).

Figure 9.6. A portfolio holds asset and can be itself viewed as an asset.



7.3.2.1. Services realized polymorphically across the recursive structure

The assets/portfolios example provides an ideal platform for demonstrating not only the direct structural benefits of the composite patterns, but also the benefit of being able to easily realize a service polymorphically across the recursive structure.

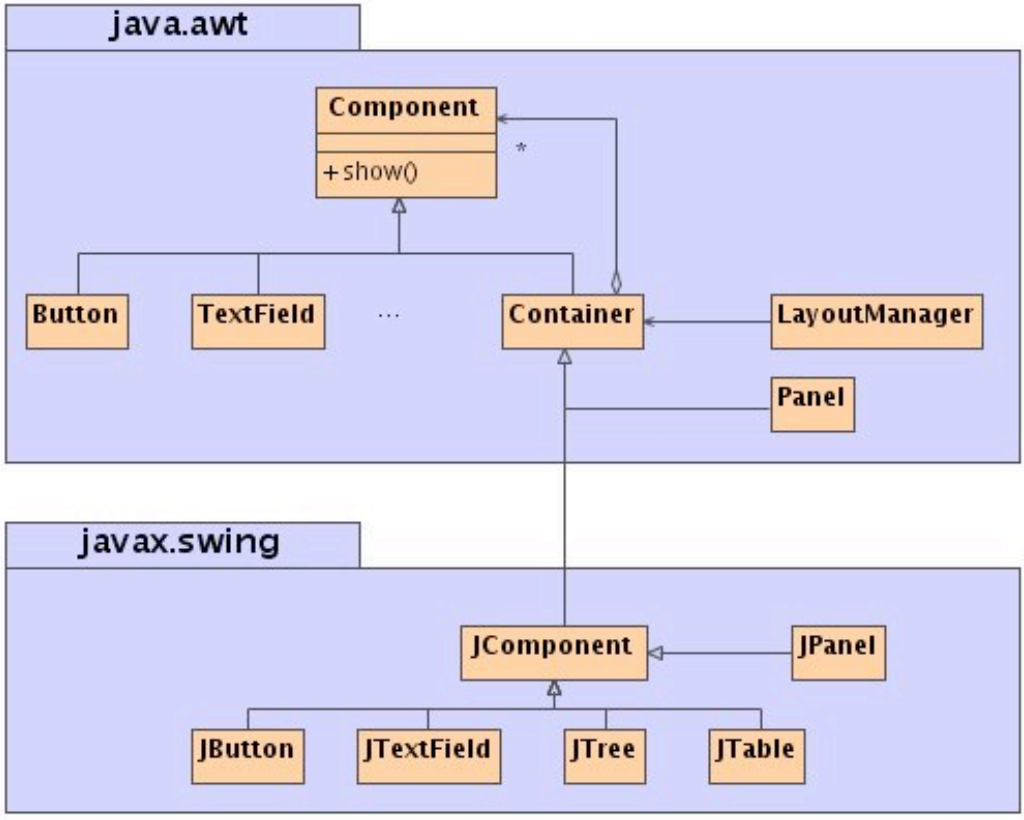
Consider the `getValue` service. Each asset may calculate its value in its own way. The container, i.e. the portfolio, will calculate its value by asking each of its components (i.e. assets) for their value and summing these values up. If any of the assets is a portfolio itself, it will automatically calculate the value by recursing to the next level in the hierarchy.

7.3.3. Java GUI libraries

The composite pattern together with the concept of layout managers provide the foundation for Java's platform and resolution independent user-interface design. The idea is simply that there are containers which hold GUI components. The delegates the responsibility of positioning its components within it to a separate class, a `LayoutManager`.

These layout managers are typically very simple, stacking the components on top or next to each other or within grid positions on the screen. A sophisticated, slick panel design which is platform and resolution independent is then achieved by nesting containers. The result is usually a simpler presentation layer where screens are assembled from re-usable, simple panels. The design of the Java user-interface libraries are shown in Figure 9.7, “ Java's user interface libraries achieve resolution independence partially through the composite pattern. ”.

Figure 9.7. Java's user interface libraries achieve resolution independence partially through the composite pattern.



7.3.4. Document structures

Documents themselves may require recursive structures which are processed recursively. For example, *DocBook*, a public statndard for XML-based documentation which is also th source for the document you are currntly reading, supports a composite pattern for sections. Sections may contain further sections (the sub -sections) which in turn may recursively contain further sections.

7.4. Consequences

- Primitive and composite objects are interchangeable. Client applications may treat both, primitive and composite objects uniformly and often don't know or care whether they are working with a primitive or a complex object.
- You can support a wide range of custom made object, each with its own set of components.
- Services can be realized polymorphically across the recursive containment structure.
- The composite pattern can be realized for all 3 core OO relationships:
  - **Composition.** In this case the components are fully owned by the container and may be accessed only through the container. In this case two containers cannot contain the same component in the sense of being the same object -- the same instance. They can at most contain each their own copy of a component. Composite patterns using a composition relationship between the container and its components are not used very often.
  - **Aggregation.** This is the most common form of the composite pattern. Here the components are part of the state of the container. This implies that if a component's state changes the state of the container changes (e.g. if the state of an asset in the portfolio changes, for ex-

ample, its value, then the state of the portfolio itself changes -- it too changes its value). Here multiple containers can contain the same component.

- **Association.** An association represents a client/server relationship. Thus, if a composite pattern uses an association relationship between the container and its components, then we have a potentially recursive client/server relationship. This structure provides facilities for adding responsibilities to a service on the fly. In that sense it would be similar to the decorator pattern except that

  - lower level service providers are added to the other end of the structure -- the decorated component is the container itself,
  - a whole range of extra deliverables to a service can be added at the same time,
  - clients continue interfacing with the same object, i.e. the client reference along which the service requests are sent need not be modified to point to a decorator but continues to request the service from the original service provider.

  In the case of an associative composite pattern the root component is usually an interface which is implemented by both, leaf components and containers.

7.5. Implementation guidelines

The composite pattern is very easy to implement. The core decision to be made is whether one should use composition, aggregation or association for the containment relationship (where the latter extends the original intent of the composite pattern).

7.5.1. Implementing composition relationships

When implementing a composition relationship you need to enforce *access control*. To this end make certain that the container is the only object which has a reference to any mutible component. This may require cloning the component when it is added to the container as well as when the component is queried from the container.

Composition also implies that the state of the component forms part of the state of the composite object. Hence, if your system supports state change notification, then you will have to implement *state transition propagation*.

For example, if you make use of the JavaBeans state change propagation framework, your container will have to

1. register as `ChangeListener` with the components,
2. generate a `ChangeEvent` for every `ChangeEvent` received from any of its components.

7.5.2. Implementing aggregation relationships

Aggregation does not require access control and hence containers may share the same components. t does, however, still imply that the component state is part of the state of the aggregate object and hence *state transition propagation* must still be implememnted.

7.5.3. Implementing associative composite patterns

Here the container simply keeps a reference to each of its components. When the composite service is requested the container will provide the generic realizations of the services and then forward the service request to each of the components so that they can address the additional responsibilities as-

signed to the service.

7.5.4. Simple implementation of asset/portfolio example

Let us, as an example, look at a very simple Java implementation of the asset/portfolio example.

7.5.4.1. Valuable.java

```
package za.co.stc.finance;

public interface Valuable
{
  public double getValue(java.util.Date date);
}
```

7.5.4.2. Asset.java

```
package za.co.stc.finance;

public abstract class Asset implements Valuable
{
  public Asset(String id) {this.id = id;}

  public String toString() {return id;}

  public String getId() {return id;}

  private String id;
}
```

7.5.4.3. PurchasedAsset.java

```
package za.co.stc.finance;

import java.util.*;

public abstract class PurchasedAsset extends Asset
{
  public PurchasedAsset(String id,
                        double purchasePrice,
                        Date purchaseDate)
  {
    super(id);
    this.purchasePrice = purchasePrice;
    this.purchaseDate = purchaseDate;
  }

  public double ageInYears(Date date)
  {
    double ageInMilliSecs
      = date.getTime() - purchaseDate.getTime();
    return ageInMilliSecs / milliSecsPerYear;
  }

  public double getPurchasePrice() {return purchasePrice;}

  public Date getPurchaseDate() {return purchaseDate;}

  public String toString()
  {
    return super.toString() + " purchased on "
      + purchaseDate + " for R" + purchasePrice;
```

```
  }

  private static final double milliSecsPerYear
    = 1000.0 * 60 * 60 * 24 * 365.25;

  private double purchasePrice;
  private Date purchaseDate;
}
```

7.5.4.4. Vehicle.java

```
package za.co.stc.finance;

import java.util.*;

public class Vehicle extends PurchasedAsset
{
  public Vehicle(String id, double purchasePrice,
                 Date purchaseDate,
                 double writeOffPeriodInYears)
  {
    super(id, purchasePrice, purchaseDate);
    writeOffPeriod = writeOffPeriodInYears;
  }

  public double getValue(Date date)
  {
    double age = ageInYears(date);

    return getPurchasePrice() * (1 - age / writeOffPeriod);
  }
  public String toString()
  {
    return " Vehicle " + super.toString()
      + " written off over " + writeOffPeriod + " years, "
      + " current value = R" + getValue(new Date());
  }

  private double writeOffPeriod;
}
```

7.5.4.5. Property.java

```
package za.co.stc.finance;

public class Property extends PurchasedAsset
{
  public Property(String id, double purchasePrice,
                  java.util.Date purchaseDate)
  {
    super(id, purchasePrice, purchaseDate);
  }

  public double getValue(java.util.Date date)
  {
    return getPurchasePrice();
  }

  public String toString()
  {
    return " Property " + super.toString();
  }
}
```

7.5.4.6. Portfolio.java

```
package za.co.stc.finance;

import java.util.*;

public class Portfolio extends Asset
{
  public Portfolio(String id) {super(id);}

  public void add(Valuable asset)
  {
    assets.add(asset);
  }

  public double getValue(Date date)
  {
    double sum = 0;
    Iterator iter = assets.iterator();
    while (iter.hasNext())
    {
      Asset asset = (Asset)iter.next();
      sum += asset.getValue(date);
    }
    return sum;
  }

  public String toString()
  {
    StringBuffer result = new StringBuffer();
    result.append("Portfolio: " + super.toString());
    result.append(", current value = "
      + getValue(new Date()) + eol);
    Iterator iter = assets.iterator();
    while (iter.hasNext())
      result.append(iter.next()).append(eol);
    return result.toString();
  }

  private Collection assets = new LinkedList();

  private final static String
    eol = System.getProperty("line.separator");
}
```

7.5.4.7. PortfolioTest.java

Now we can add assets to portfolios. Some of these may be portfolios themselves. The portfolios are recursively valuated down to leaf-asset level:

```
package za.co.stc.finance;

import java.util.*;

public class PortfolioTest
{
  public static void main(String[] args)
  {
    new PortfolioTest().run();
  }

  public void run()
  {
    Portfolio p1 = new Portfolio("myStuff");
    p1.add(new Property("myHouse", 180000,
      new GregorianCalendar(1999, 0, 20).getTime()));
```

```
    p1.add(new Vehicle("myCar", 350000,
      new GregorianCalendar(2000, 10, 22).getTime(), 5));

    System.out.println("portfolio (p1) value = "
      + p1.getValue(new Date()));

    System.out.println(p1);

    try
    {
      Thread.currentThread().sleep(10000);
    }
    catch (InterruptedException e) {}

    System.out.println(p1);

    Portfolio p2 = new Portfolio("myOtherStuff");
    p2.add(new Property("myBeachHouse", 120000,
      new GregorianCalendar(2002, 0, 20).getTime()));
    p2.add(new Vehicle("myBeachBuggy", 35000,
      new GregorianCalendar(2001, 10, 22).getTime(), 10));

    p1.add(p2);

    System.out.println("\n\n" + p1);
  }
}
```

7.6. Related patterns

- **The decorator pattern.** The decorator pattern is often used instead of the associative composite pattern. It facilitates the step-for-step decoration of a service with further responsibilities.
- **Chain of responsibilities.** This is another pattern often used when further responsibilities need to be added on the fly.
- **The Visitor pattern.** The visitor pattern is used if further polymorphic services are to be added externally to a class hierarchy.
- **Iterator.** Iterators are commonly used by containers to taverse the component collection.

8. The decorator pattern

The decorator pattern is used widely for GUI libraries, but it is applicability ranges far beyond this. Many applications can be improved by using the decorator instead of subclassing to dynamically add additional responsibilities to a context.

8.1. Intent

- To add responsibilities or tasks to a service at object (instance) level without modifying the class which hosts the original service, i.e. different instances of the same class can thus realize a service in different ways.
- To be able to plug in these extra responsibilities on the fly (at run-time).
- To be able to remove extra responsibilities on the fly.
- To simplify the maintainance of services which may be provided with a wide range of add-ons.

8.2. Structure

The decorator

- *is a* specialization of the decorated component facilitating substitutability.
- *has the decorated component*, i.e. the decorated component is contained facilitating run-time plug-ins.

The structure is shown in Figure 9.8, " Structure of the decorator pattern. ".

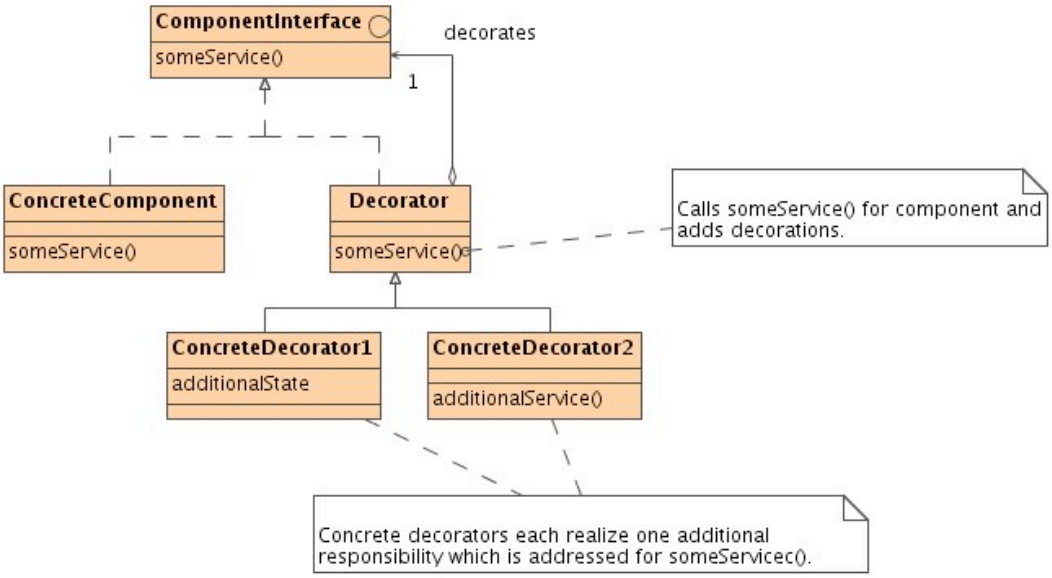
Note

The "is a" (specialization) relationship is used solely for facilitating *substitutability* and not for either

- inheritance or to
- delegate the service request up the specialization hierarchy as is commonly done.

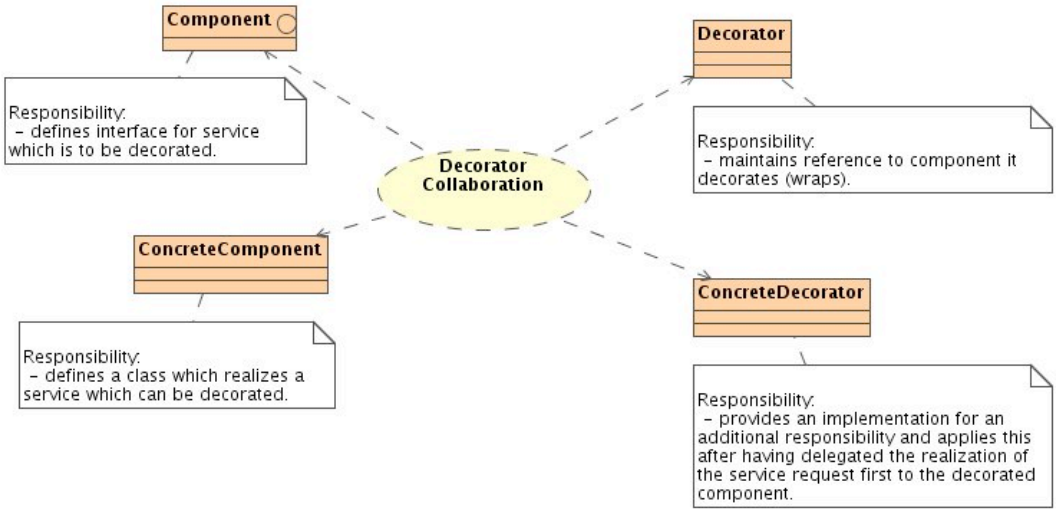
Instead, the service request is passed up the containment hierarchy.

Figure 9.8. Structure of the decorator pattern.



The abstract collaboration shown in Figure 9.9, "The responsibilities of the role players of the decorator pattern" shows the responsibilities of the role players of the decorator pattern.

Figure 9.9. The responsibilities of the role players of the decorator pattern



The structure of the decorator pattern is very similar to that of the composite pattern except that

- the decorator has only one component it decorates and
- the decorator provides the same service as the component it decorates and in its implementation it

1. Calls the corresponding service of the object it decorates using the aggregation relationship as service request path.
2. After the decorated component has realized the service up to its level of responsibility the decorator adds further responsibilities.

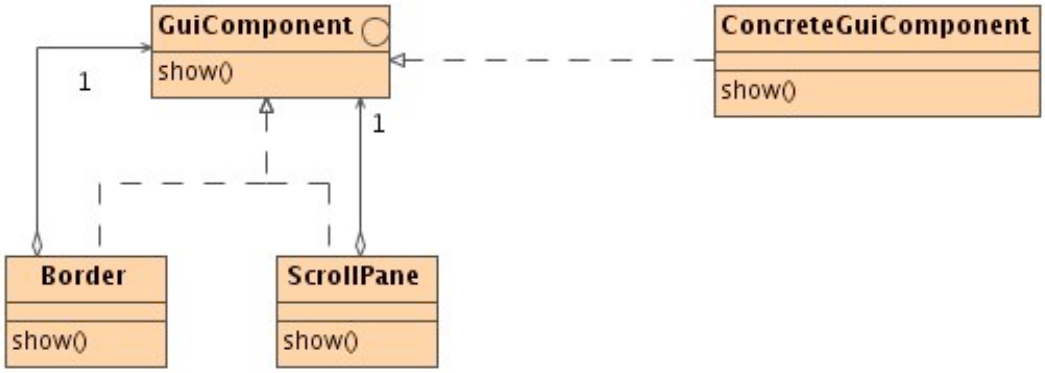
8.3. Example applications

It is common to use the decorator pattern for GUI class libraries where core components, supporting pluggable borders, scroll panes, and other

8.3.1. GUI components

The decorator pattern is used widely in GUI libraries where a GUI component may be decorated by potentially multiple decorators.

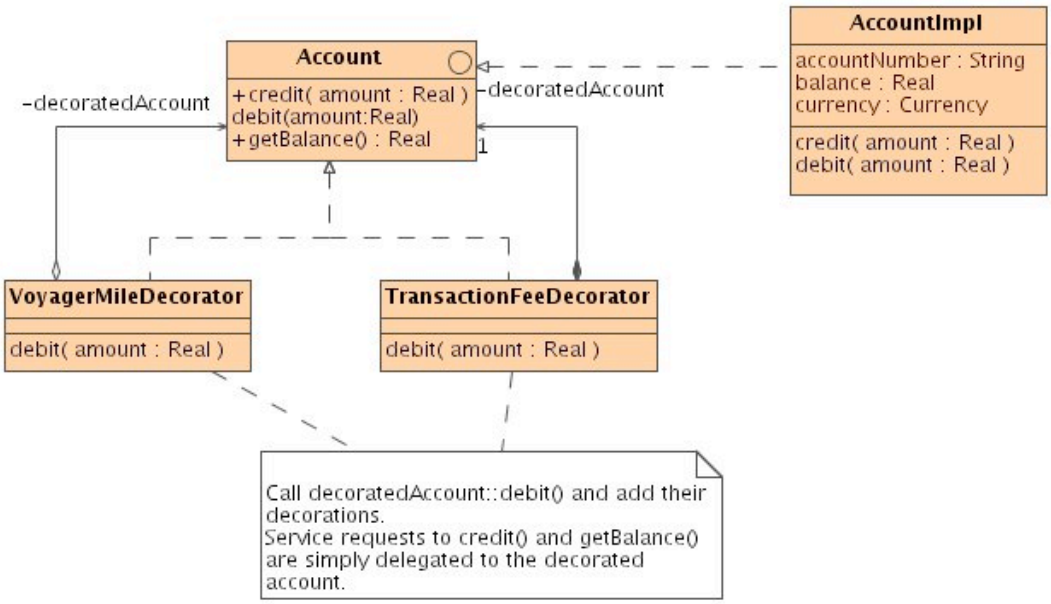
Figure 9.10. Any Gui component can be decorated with a border or a scroll pane.



8.3.2. Account decorators

The decorator pattern is, however, more widely applicable that simply GUI libraries. For example, in Figure 9.11, “ The debit service of an account can be decorated by voyage miles, transaction fees and potentially further decorators. ” we decorate accounts with voyager miles, service fees or any other pluggable functionality.

Figure 9.11. The debit service of an account can be decorated by voyage miles, transaction fees and potentially further decorators.



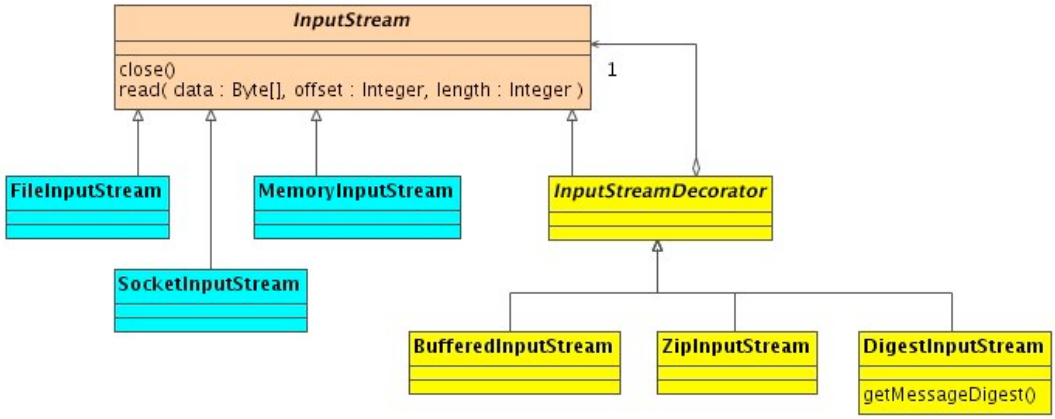
8.3.3. I/O stream decorators

Another very common application of the decorator pattern is that of I/O streams. The decorators add additional responsibilities before the data is written to or read from a stream. These may include

- buffering,
- compression/decompression of data,
- marshaling onto primitive data types or onto objects,
- encryption/decryption,

- accumulation of a hashing function value to verify data integrity and so on.

Figure 9.12. The various concrete input streams can be decorated with a combination of input stream decorators.



Once again, we can plug the decorators one into another. For example, we can attach a

- `BufferedInputStream` to a `FileInputStream` to provide buffering,
- then a `DigestInputStream` which accumulates the message digest (i.e. hashing function value) during reading and provides a mechanism for querying the message digest after the data has been read,
- and finally a `ZipInputStream` onto the `BufferedInputStream` which unzips the contents as it is read.

8.4. Benefits above method overriding

You may ask yourself

> " *Why have the decorator pattern at all if we can simply define a specialized class which decorates the superclass by delegating the realization of a service request to it before adding any additional responsibilities?* "

The core benefits for using the decorator pattern instead of simple method overriding are that

- you can add decorations to an object on the fly and that
- you can add different responsibilities to different instances of the same class.

8.5. Consequences

- **Avoids complex specialization hierarchies.** Using the decorator pattern avoids complex specialization hierarchies with a subclass for any particular combination of responsibilities.
- **Better localization of responsibilities.** Each decorator encapsulates a specific responsibility

which can be added to different components. The responsibility is only defined at that particular place. For example, scrolling support is only defined within a single class in the Java GUI class libraries -- `ScrollPane`. Any object which requires scrolling (e.g. an editor pane, a regular or drop-down list box, or an entire panel) is simply decorated with a `ScrollPane`.

- **Decorating at object-level.** Individual instances of the same class can be decorated with different responsibilities.
- **Adding/removing responsibilities dynamically.** The decorator pattern enables you to add and remove responsibilities for a service on the fly.

8.5.1. Object identity is not preserved

Technically speaking the object which is decorated and the decorated object are two different objects and the *client object needs to change its reference* from the original object to the decorator.

This compromises the identity of the object and can be vulnerable to error if the implementation is not done very carefully. If the service is requested directly from the object and not through the decorator, the original non-decorated service will be provided.

8.6. Implementation guidelines

- Since the specialization link between the decorator and the decorated component is purely for substitutability and not for the sake of inheritance, the *component being decorated should virtually always be an interface* and not an implementation.
- When service which adds or removes a responsibility dynamically will need to update all the references from the decorated object to the decorator. This is potentially complex.

8.7. Related patterns

- **Associative composite pattern.** If the fact that an object changes its identity when an additional responsibility is added becomes an issue -- i.e. the maintenance of the references to the decorated object becomes a excessively complex, you may want to use the associative composite pattern instead. In the latter case the client reference remains unchanged (i.e. the client continues to interface with the same object via the same message path -- the same reference).
- **State pattern.** If the decorations are state dependent and change frequently, you may want to use the state pattern either instead of the decorator pattern or in conjunction with the decorator pattern.
- **Strategy pattern.** Both, the decorator pattern and the associative composite pattern add additional responsibilities to a service on the fly. They do not change the core realization of the service -- both still deliver the original service before adding to it. If the core realization of the service needs to be altered, you may want to consider the strategy pattern.
- **Chain of responsibilities.** This is another pattern often used when further responsibilities need to be added on the fly.
- **The Visitor pattern.** The visitor pattern is used if further polymorphic services are to be added externally to a class hierarchy.

9. The visitor pattern

At times one has a compiled class library and one wants to add some polymorphic functionality to the classes without modifying the classes themselves. The reason for this may be that

- **Source code is not available.** The source code may not be available and hence one may not be able to modify the classes to add the desired functionality.
- **Maintain ownership status of classes.** At other times the source may be available but the team who is responsible for maintaining the source is unwilling to modify it and you do not want to maintain your own branch of the source.
- **Protect the responsibility focus (cohesion level) of the classes.** The classes may be used across the organization and one has not got the primary responsibility of the classes. The development team may be hesitant to add the functionality because they feel that functionality is only useful to a specific user group and they fear that their classes will bloat over time and loose their cohesion -- that they will acquire over time a wide range of responsibilities which are not aligned with the primary responsibility of the classes.

In such cases the Visitor pattern may be useful. It enables one to add polymorphic functionality to a class hierarchy without modifying or sub-classing the classes. This is quite an ambitious task and the visitor pattern provides a novel solution to this problem.

9.1. Intent

To add a polymorphic service to a class hierarchy without

- sub-classing the classes in the hierarchy and
- without modifying any of the classes in the class hierarchy.

9.2. Solution

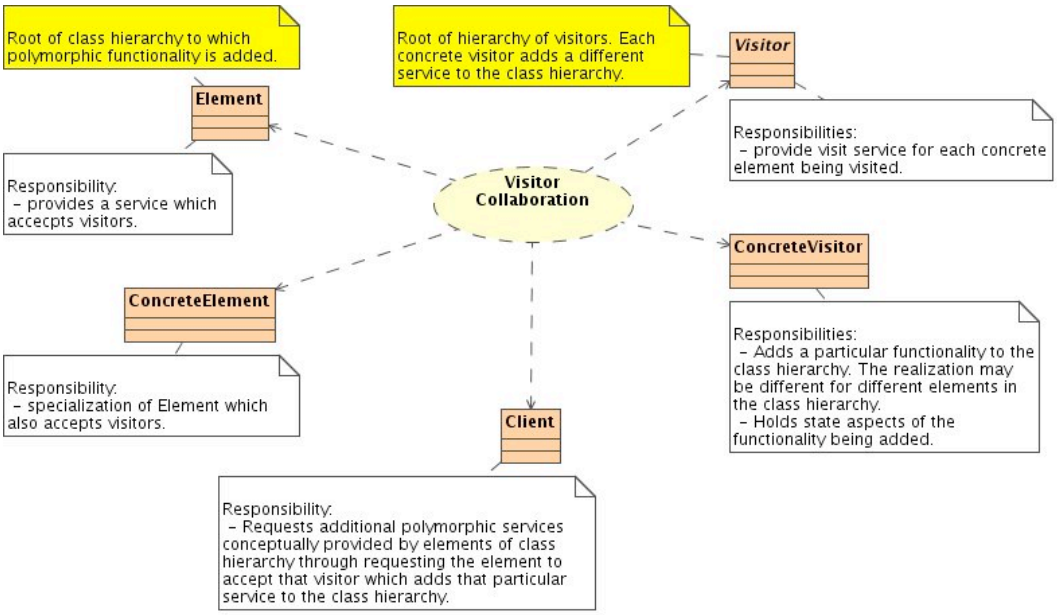
The solution has 3 aspects:

1. The responsibility distribution across the elements of the visitor pattern.
2. The static structure required by the visitor pattern.
3. The dynamics (i.e. the algorithm) used by the visitor pattern.

9.2.1. Responsibility Allocation

The abstract collaboration shown in Section 9.2.1, “Responsibility Allocation” shows the responsibilities of the different role players of the visitor pattern.

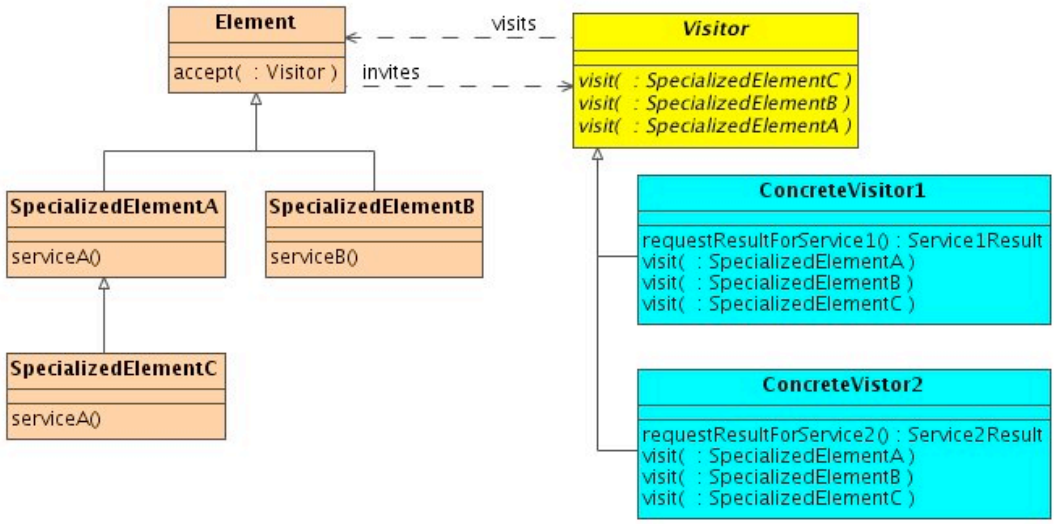
Figure 9.13. Responsibility allocation across the members of the visitor pattern



9.2.2. Structure

The structure of the visitor pattern is relatively straight forward. We have a class hierarchy which accepts visitors. Different specializations of a visitor add different polymorphic services to the class hierarchy. The structure of the visitor pattern is shown in Figure 9.14, “Structure of the visitor pattern”.

Figure 9.14. Structure of the visitor pattern



9.2.3. Dynamics

The dynamics of the visitor pattern may look a little convoluted at first:

1. Clients request a service added to a class hierarchy via a visitor from an element added to that class hierarchy by asking the element to accept the vistor.

2. The element then invites the visitor to visit it, providing the visitor with its address -- how else

could the visitor come and visit?

3. The visitor then makes one or more call-backs to that element, requesting any information it may need from the element to realize its service.

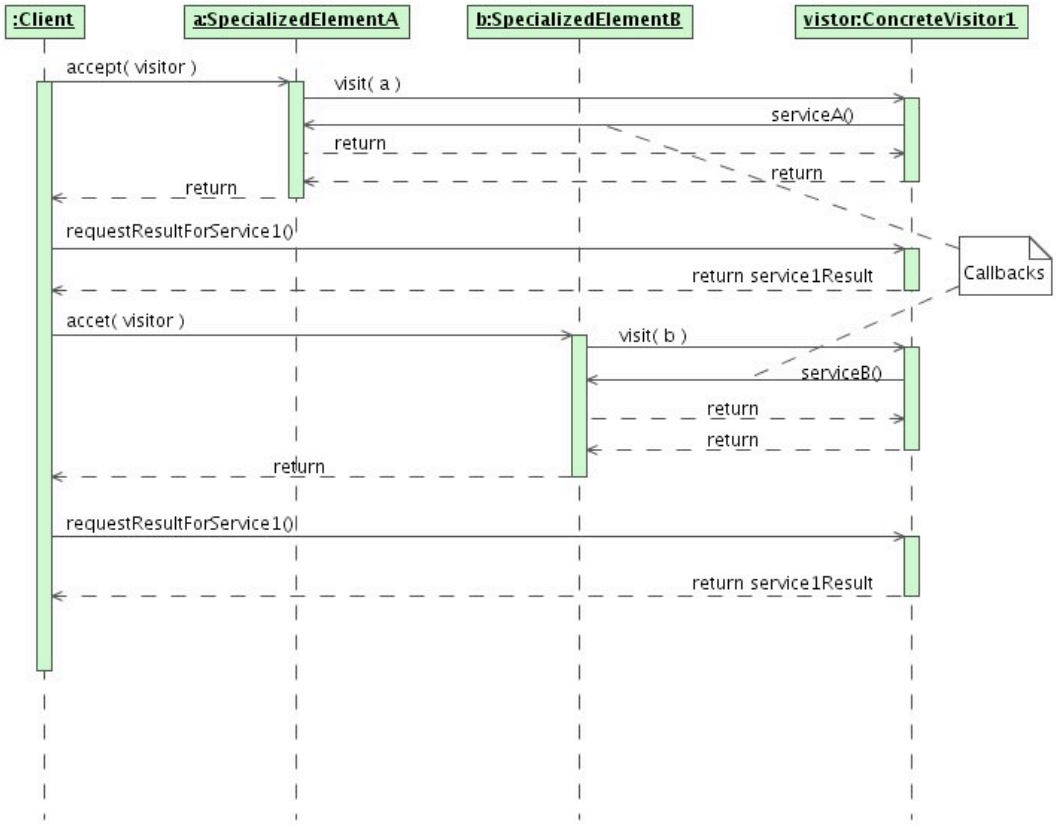
4. Upon completion of the service, the visitor returns control to the element which, in turn, returns control back to the client.

5. At this stage the client may request the deliverables of the service request directly from the visitor.

Note

Alternatively the visitor may accumulate a result across multiple service requests and the result may be queried at the end.

Figure 9.15. Dynamics of the visitor pattern



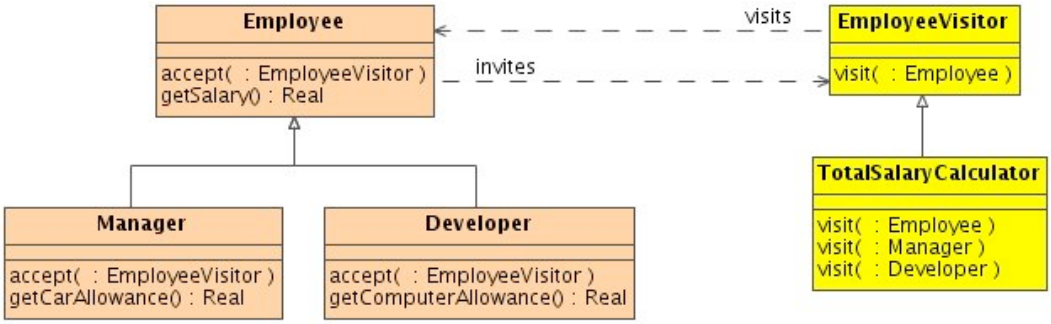
9.3. Example applications

The visitor pattern may become a lot clearer by looking at a simple example application.

9.3.1. Total salary calculator

Consider the example where you have a polymorphic collection of employees. All employees earn a basic salary, but different types of employees receive different types of perks. For example, in Figure 9.16, “The structure used for a visitor adding a getTotalSalary() service to employees.” we have managers which receive a car allowance and developers which receive a computer allowance.

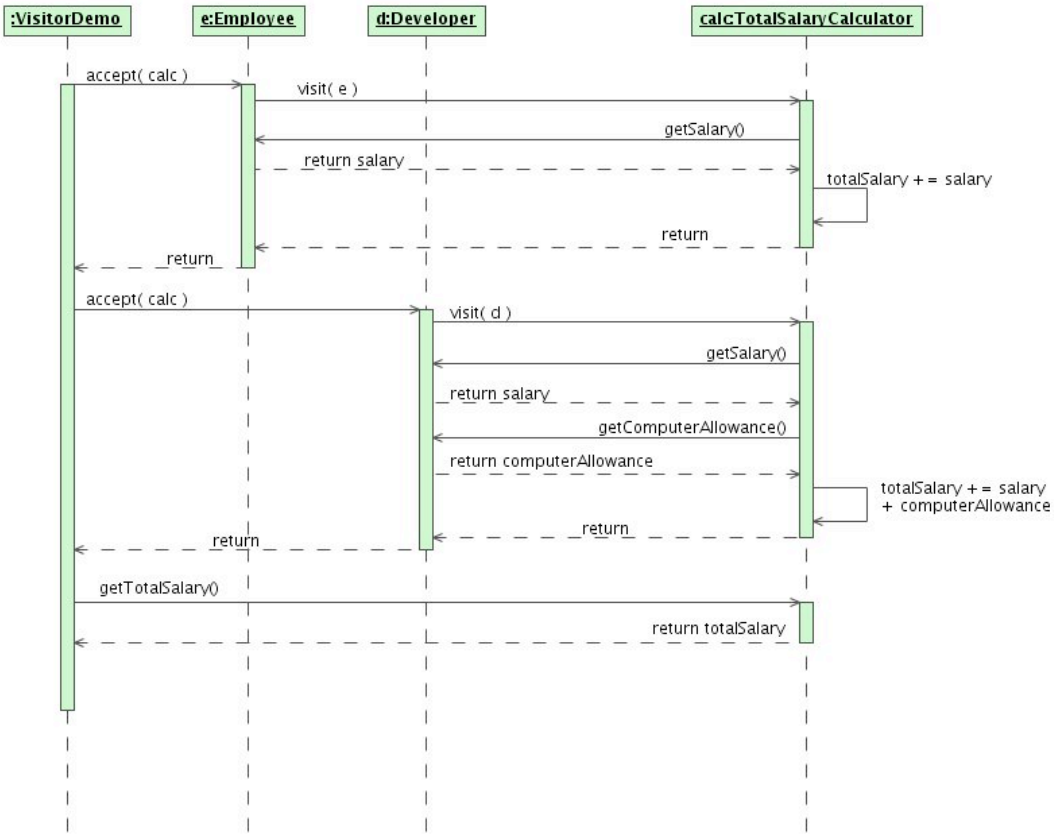
Figure 9.16. The structure used for a visitor adding a getTotalSalary() service to employees.



The `TotalSalaryCalculator` adds a polymorphic `getTotalSalary()` method to the class hierarchy which executes different algorithms for different types of employees.

The dynamics of the pattern is illustrated in the sequence diagram shown in Figure 9.17, “The dynamics of the TotalSalaryCalculator.”.

Figure 9.17. The dynamics of the TotalSalaryCalculator.



9.4. Consequences

- The Visitor pattern enables one to *add polymorphic functionality* to classes without modifying or sub-classing them.
- It can help *preserve the focus of classes* encapsulating functionality which is not part of the core responsibility within separate visitors.
- You can define as *many visitors* as you like, each only adding services within a narrow responsibility focus. In this way one ensures that the design elements maintain a high cohesion level.
- Although our Visitor example only operates within a class hierarchy, the Visitor pattern can also be used at *add functionality to classes across separate class hierarchies*.
- The visitor provides a mechanism to *aggregate information across service requests*.
- *Adding new concrete elements results in a lot of maintenance* across the entire Visitor hierarchy. You should only consider applying the visitor pattern to mature class hierarchies where modifications and additions are not expected.
- Visitors *can only add functionality which uses the public interface of the elements*. The temptation to sacrifice encapsulation may become overwhelming.
- The visitor pattern *introduces cyclic dependencies* which makes it difficult to maintain (see the *acyclic visitor pattern*).
- There is no clean way to add visitors which are meant to only visit a subset of the element hierarchy.

9.5. Implementation guidelines

One of the core issues which need to be addressed when implementing the visitor pattern is that in most current object-oriented programming languages *methods are resolved polymorphically on message recipients, not on message parameters*. This poses a problem because we need polymorphism on message arguments in the visitor pattern on the invitation request, `accept(:Visitor)`.

To support this we need to

1. Override the `accept(:Visitor)` method in each element of the class hierarchy to which the polymorphic service is added with the identical code:

   ```
   visitor.visit(this);
   ```

   The reason why this addresses part of the problem is that the `this` pointer has a different type for each member of the class hierarchy.

2. The interface must provide a separate `visit` method for each element being visited.

Figure 9.18. VisitorTest.java

```
class Employee
{
  public Employee(String name, double salary)
  {
    this.name = name;
    this.salary = salary;
  }
  public double getSalary()
  {
```

```
    return salary;
  }
  public void accept(EmployeeVisitor visitor)
  {
    visitor.visit(this);
  }
  public String toString()
  {
    return name + " :  " + salary;
  }
  private String name;
  private double salary;
}
//---------------------------------------------------------------

class Manager extends Employee
{
  public Manager(String name, double salary, double carAllowance)
  {
    super(name, salary);
    this.carAllowance = carAllowance;
  }
  public double getCarAllowance()
  {
    return carAllowance;
  }

  public void accept(EmployeeVisitor visitor)
  {
    visitor.visit(this);
  }

  public String toString()
  {
    return super.toString() + " (" + carAllowance + ")";
  }
  private double carAllowance;
}
//---------------------------------------------------------------

class Developer extends Employee
{
  public Developer(String name, double salary, double computerAllowance)
  {
    super(name, salary);
    this.computerAllowance = computerAllowance;
  }
  public double getComputerAllowance()
  {
    return computerAllowance;
  }

  public void accept(EmployeeVisitor visitor)
  {
    visitor.visit(this);
  }

  public String toString()
  {
    return super.toString() + " (" + computerAllowance + ")";
  }
  private double computerAllowance;
}
//---------------------------------------------------------------

interface EmployeeVisitor
{
  public void visit(Employee o);
```

```
  public void visit(Manager m);

  public void visit(Developer m);
}
//-------------------------------------------------------------

class TotalSalaryCalculator implements EmployeeVisitor
{
  public void visit(Employee employee)
  {
    totalSalary += employee.getSalary();
  }
  public void visit(Manager manager)
  {
    totalSalary += manager.getSalary();
    totalSalary += manager.getCarAllowance();
  }
  public void visit(Developer developer)
  {
    totalSalary += developer.getSalary();
    totalSalary += developer.getComputerAllowance();
  }

  public double getTotalSalary() {return totalSalary;}

  private double totalSalary = 0;
}
//-------------------------------------------------------------

public class VisitorTest
{
  public void run()
  {
    Employee[] employees = new Employee[6];
    employees[0] = new Employee("Peter", 100000);
    employees[1] = new Manager("Tandi",  100000, 32000);
    employees[2] = new Employee("Jannie", 100000);
    employees[3] = new Manager("Ahmed", 100000, 27000);
    employees[4] = new Developer("Li", 100000, 6000);
    employees[5] = new Employee("Jill", 100000);

    TotalSalaryCalculator totalSalaryCalculator
      = new TotalSalaryCalculator();

    for (int i=0; i<employees.length; ++i)
      employees[i].accept(totalSalaryCalculator);

    System.out.println("totalSalary = " + totalSalaryCalculator.getTotalSalary(
  }

  public static void main(String[] args)
  {
    new VisitorTest().run();
  }
}
```

9.6. Related patterns

- **Decorator pattern.** The visitor pattern enables you to add a polymorphic service externally to a class hierarchy. The decorator pattern enables you to add responsibilities externally for an existing service of a class hierarchy.
- **State pattern.** The state pattern enables you to externally specify state-specific ways of realizing an existing service.

- **Exposed state pattern.** While the visitor pattern enables you to add a polymorphic service without changing any of the classes in the class hierarchy, the exposed state pattern enables you to add state specific services to a context without changing the context itself.

- **Acyclic visitor pattern.** The acyclic visitor pattern addresses 3 of the core issues which may be raised against the classical visitor pattern:

    - The visitor pattern *is not maintenance friendly* since all visitors must be modified every time a new class is added to the class hierarchy to which a polymorphic functionality is added.

    - There are circular dependencies between the elements of the class hierarchy and the visitors. The consequence of this is that the visitor pattern has *considerable compilation overheads*, requiring that all classes are recompiled when any of the classes are modified.

    - The classical visitor pattern does not support *partial visitations*, i.e. that a visitor adds a service to only part of the class hierarchy.

Chapter 10. Aspect Oriented Development

1. Aspect-Oriented Development

Aspect oriented development is being used more widely as the benefits of this paradigm become more appreciated.

1.1. Apect-oriented development and non-functional requirements

Aspect oriented development provides a suitable framework for addressing those aspects of the non-functional requirements which are not addressed by infrastructure (i.e. architecture). The logic addressing these non-functional requirements typically needs to applied across use cases.

For example, the logic applying logging, security or profiling may be the same across use cases. One may wish to log all entries and exits from all services and one may wish to validate authorization on entry nto any service. Such aspects are commonly addressed through aspect-oriented development.

1.2. Core concepts of aspect oriented development

Aspect oriented development introduces a number of new concepts. One needs to understand these as well as the associated terminology in order to to understand aspect oriented development.

1.2.1. Cross cutting concern

A cross cutting concern is an aspect of the requirements which is relevant across use cases/services. Examples include logging requirements, authorization, authentication, ...

1.2.2. Advice

An advice is a workflow step which is applied to an existing model. This could be the logic for the logging or authorization code.

1.2.3. Point cut

The point or more often collection of points) where the cross cuttng concern is to be applied. This could be things like

- all methods for all classes,
- all `doGet` methods for all classes whose name ends with `Servlet`.
-

1.2.4. Aspect

The combination of a point cut and an advice.

1.2.5. Weaving

Weaving is used to refer to the process of inserting the aspect logic into the core application logic.

1.3. Aspect versus Object oriented development

Aspect oriented development is not an alternative to object-oriented modeling and development. In-

stead it complements object oriented development by being able to effectively address cross-cutting concerns.

The conceptual solution as well as the implementation realizing the use case (functional) requirements would typically be designed using object-oriented concepts. However, those non-functional requirements which are not addressed by architecture (infrastructure), could typically be addressed by designing aspects into the solution and weaving the resultant code into the object-oriented system.

1.4. Frameworks for aspect-oriented programming

There are a number of frameworks which support aspect oriented programming. The most well kown is the first AOP framework, *AspectJ* for Java. A range of further AOP frameworks are available for the Java based technologies including *AspectWertkz*, *JBoss AOP* and *Spring AOP*. The Microsoft.Net world provides support for aspect-oriented software development via frameworks like Aspect.Net and AOP.Net.

1.4.1. Aspects via annotations

It should be noted that some frameworks like the original AspectJ apply additions to the actual programming language. Such frameworks require special compilers. Most other frameworks address aspects using the natual extension mechanism,annotations, provided by many modern programming languages including C# and Java. extension mechanism, annotations. These frameworks do not require changes to the underlying programming language.

Note

AspectJ and AspectWerkz are collaboraborating on a new common implementation for a Java-based AOP framework.

1.4.2. Real-time weaving

Some frameworks like ApectWerkz support real-time weaving. In this case aspects are woven into compiled classes as they are loaded by custom class loaders. Alternatively aspects can be applied

- at source code level by a pre-compilation step,
- at binary level by a second compilation step.

1.5. A simple example

1.6. Benefits of aspect-oriented development

Using aspect oriented development introduces a number of benefits for the project as a whole.

1.6.1. Improved responsibility location

Instead of having non-functional logic defined across the system, the logic is localized within an aspect which is woven into the core functional logic.

1.6.2. Core functional logic not polluted by non-functional logic

The core code which addresses the functional (use-case) requirements is not polluted by the logic realizing aspects of the non-functional requirements. hese are usually not directly related to the functional requirements.

1.6.3. Clean responsibility distribution across architecture, functional design and aspects

The introduction of aspects complements architecture and object-oriented design by adressing the one aspect which is not well covered by the other two. The result is that

- **Architecture.** addresses the non-functional requirements which can be addressed through infrastructure,
- **Functional (Object-Oriented) design.** realizes the use case requirements, and
- **Aspects.** address the non-functional requirements which require logic.

Chapter 11. Software Development Processes

1. Introduction

In this chapter we look at general software development methodologies including

- general best practices,
- ideas from the *Rational Unified Process (RUP)*
- and ideas from *Extreme Programming*.

2. What do we want from a development process

A process should assist in the following aspects:

- To establish system requirements and document them in a form comprehensible to, the client, end-users and developers.
- To reduce the risk associated with a project.
- To make a good estimate of the time and resources required for the various units of work.
- To price the system or pieces of work in such a way that it is neither overpriced (otherwise competitors will obtain the contract) nor underpriced (otherwise one would potentially make a loss).
- To judiciously assign responsibilities and tasks within the development organization.
- To be able to understand at any stage how far one is in the development process and to obtain an estimate of the time and resources required to complete the project.

3. Basic Process Elements

The basic elements which need to be addressed by any process include:

- Business Modeling
- Requirements Analysis and Specification
- Architectural Design
- Construction
- Introducing Coding Standards
- Integration
- Testing
- Deployment

4. Best Practices

A number of people have looked at a wide range of projects in order to try and identify those aspects of a development process which seem to differentiate the more successful projects from the less successful ones. The result of these studies is a collection of *best practices*.

These best practices resemble general guidelines which should increase the success rate of your projects. They include

1. Management of System Requirements.
2. Iterative development.
3. Visual modeling.
4. Component-based architectures.
5. A process for verifying software quality.
6. A framework for change control for requirements and designs and implementations.

4.1. Managing System Requirements

If you have been involved in software development for any length of time it is likely that you were at some or other stage involved in building a system which was quite different to what the client wanted. The risk of building the wrong system is perhaps the largest risk your development team may be exposed to.

A good process for eliciting, documenting, verifying and managing system requirements is thus essential for successful projects. A study done in 1989 \cite{Finkelstein89}. has shown that well above 50% of system errors were due to errors in the system requirements, i.e. either that

- they were wrong,
- they were incomplete,
- or that they were mis-understod.

Furthermore, because an error at the requirements is much more expensive to fix than an implementation error it was found that more than 80% of the cost of errors were due to errors in the system requirements while the cost of implementation errors was found to be less than 1% (see Table 11.1, “Source of errors in software systems and the cost incurred in order to correct these errors.”).

Things have perhaps improved a little bit but even recent studies suggest that more that around 50% of errors arise from the system requirements specification.

Table 11.1. Source of errors in software systems and the cost incurred in order to correct these errors.

Development Stage	Source of error	Effort (cost) to correct
Requirements	56%	82%
Analysis/Design	27%	13%
Implementation	7%	1%

Development Stage	Source of error	Effort (cost) to correct
Other	10%	4%

System requirements must be elicited, verified, organized and documented in a form which is accessible to the client, users of the system and other domain experts as well as to developers. These system requirements are really business requirements and should be treated as such. UML diagrams are very useful for documenting system requirements, especially the use-case, sequence and deployment diagrams.

The process of eliciting, documenting and managing system requirements is so important that a separate course is offered focusing solely on these aspects of the software development process.

4.2. Iterative Development

If one takes a large software project sequentially through the phases of analysis, design implementation and testing (the traditional waterfall process) one is usually exposed to a lot of risk. It is difficult to monitor how far one is in the development process and it is not yet clear how and at what cost the remaining difficult problems will be resolved.

The modern approach is a more iterative approach where demonstrable prototypes are released frequently in order to obtain client and user feedback. Often the incremental development revolves around use cases, i.e. one or more use cases are implemented or refined during each increment. This approach facilitates an incremental understanding of the problem through successive refinements as well as an incremental development of the final product.

The high-risk elements are tackled early so that the project's risk profile is reduced rapidly. Furthermore, the regular release of incremental executable prototypes facilitates more accurate progress monitoring and helps to ensure that team members remain focused on producing results.

4.3. Component-Based Architectures

A component based approach divides the system into subsystems, each with their own clearly defined responsibilities and interfaces. These components may be developed by different teams whose skills profile makes them the natural choice for the respective components. Alternatively the components may be implemented iteratively.

After a system has been decomposed into components, the system is usually a lot easier to understand and the resultant system is generally more flexible. Furthermore, components facilitate effective software reuse.

Successful component architectures and middlewares include CORBA, SOAP and Enterprise Java Beans.

4.4. Visual Modeling

Visual modeling is used to understand and specify the system requirements, structure and dynamics in a way which is easy to understand and to communicate. The industry standard for object-oriented modeling languages is UML.

4.5. Software Verification Process

It is difficult to know whether the product is correct without tests. Furthermore, it is difficult to know how far in the development process and how close to completion a project is if there are no tests in place.

Generally, any element containing logic should be testable and a test should be developed for it. It is often a good idea to develop the test before developing the actual element. When developing an element, one often tends to focus on implementation issues, while the focus when developing a test is

usually on how the element is used.

In a complex system it is usually extremely difficult to understand the implications of a modification. A testing framework can facilitate automatic testing of all system elements and it helps to monitor any problems which are introduced by system modifications.

4.6. Change Control

Change to released system elements van be very risky due to the potential problems they may introduce into other aspects of the system. At the same time t is important to keep systems dynamic and to allow for rapid implementations of bug fixes, feature improvements and new functionality.

The answer to this dilemma is the use of a change control system which allows for roll-back together with a testing framework (see previous section). In addition, a change control system provides access control to designs and source code elements. It is advisable to introduce a rule that no elements may be released into the system for general use prior to a complete test being available.

Change control systems may provide to developers secure work spaces which are isolated from changes made in other work spaces. The latter can be a blessing as well as a risk. The result can easily be a diverging collection of work spaces which are over time difficult to integrate.

5. Ideas from the Rational Unified Process

The *Rational Unified Process (RUP)* has entrenched itself as one of the most widely used processes. Though seen by many still as a relatively heavy-weight process, it has become considerably more flexible and more light-weight over the years.

5.1. History of RUP

Rational acquired the Objectory process when Ivar Jacobson joined Rational in 1996. Rational enhanced the Objectory process with some of their ideas and with elements from other tool vendors which Rational acquired. In December 1998 Rational released the initial version of the Rational Unified process (version 5.0).

5.2. RUP as an Iterative Process

The Unified Rational Process (RUP) defines an iterative process where each iteration has 4 phases. At the end of each iteration there should be an executable product which may be a prototype, a subset of the complete vision.

5.3. 6 Core Workflows

During each of these phases the RUP defines 6 core process work flows

1. Business modeling
2. Requirements modeling
3. Analysis and Design
4. Implementation
5. Testing
6. Deployment

The amount of time spent on each of these work flows is typically different from phase to phase. During the inception phase a most of the time is spent on understanding the business and, to a lesser

extend on establishing the system requirements.

During the elaboration phase a lot of time is still spent on the business model, but the main focus is on developing the requirements Specification and on Analysis and Design. The first prototypes are being developed during this stage and time is spent on testing (verifying) the system requirements.

As we go through the phases we obtain a better understanding of both the business and the requirements and hence less time is spent on these work flows. During the construction phase the main focus is of course on implementation. The system requirements are refined further and the analysis and design is completed during this phase. A lot of time is spent on testing.

Finally, during the transition phase the main focus is on deployment. The system requirements are refined further to thrash out any loose ends. The deployment typically requires some further implementation work and a lot of testing is done during this stage.

5.3.1. Business modeling

During business modeling the vision and scope of the business is captured. The business use-cases, workflows and business objects are identified and new products and workflows are prioritized.

5.3.2. Requirements modeling

The requirements modeling covers the content of this course including

- Mapping of the business use cases and objects onto system use cases and objects.
- The development of a vision document.
- Scoping of the requirements.
- Verifying requirements.
- Development of the requirements Specification including functional and non-functional requirements.

5.3.3. Analysis and Design

Here the high-level architecture and design is developed and verified. The architecture and design is verified to ensure that

- The architecture can accommodate the requirements, especially the high risk elements.
- The architecture and design results in a maintainable system which can accommodate changing requirements.
- The architecture and design can grow with increasing number of concurrent users and higher transaction volumes.

5.3.4. Implementation

The implementation workflow covers

- The lower-level design of the system.
- The actual construction of the system in terms of components, subsystems and classes.
- The development of low-level unit tests.

- The correction of any errors spotted during unit testing.

5.3.5. Testing

In addition to unit testing done for individual objects the testing workflow tests

- that the objects interact correctly,
- that the integration of higher level components and subsystems results in a stable system,
- that the requirements have been implemented correctly,
- and that any failures are fed beck to the development team and that the system is not deployed with any defects.

5.3.6. Deployment

The deployment workflow delivers the product to the end users. It includes the following activities:

- Producing external releases of the software.
- Packaging the software.
- Distributing the software.
- Installing the software.
- Providing training and assistance to users.
- Beta testing.
- Migration of existing software or data.
- initiating the process of formal acceptance.

5.4. 3 Supporting Workflows

In addition to the 6 core work flows, the RUP defines 3 supporting work flows Project Management is responsible for allocating resources, managing risk and as a provider of status information about the project. Configuration and Change Management controls change access to designs and code and solves the problem of concurrent updates and resultant information loss, change notification and the management of multiple simultaneous versions. The Environment Work flow is responsible for providing the required environment for developers including information, training, software development, deployment and testing tools as well as documentation tools.

Let us now have a more detailed look at the 4 phases, their deliverables and their milestones.

5.5. The Inception Phase

During the inception phase one tries to understands the business motivation for the system. It is important to understand the scope of the project early in the process in order to get a feeling for the resources required for the project.

A good place to start is to identify all the actors, i.e. the external objects which interact with the system. One should understand what they want from the system and how they are going to interact with the system. The result is an initial collection of use cases with accompanying sequence or collaboration diagrams.

5.5.1. The deliverables of the inception phase

The deliverables typically include

- A *vision document* documenting the initial understanding of the core system requirements, the key features and the project and system constraints.
- An *initial business case* including the business context, the success criteria from a business perspective (e.g. increased productivity, increased sales, …).
- A *business model*. This may be necessary in order to facilitate an improved process for eliciting and specifying system requirements.
- An *initial use-case model* which contains the key use cases defining the high-level scope of the system.
- An *initial risk assessment*.
- A *project plan* including a the phases and iterations.
- An initial *project glossary* explaining, to the different role players, the terms used in the business and in the process artifacts.

After the inception phase the stakeholder should have reached agreement on project scope, cost estimates and schedule estimates. The project may be reconsidered or cancelled if consensus is not reached.

5.5.2. Evaluation criteria for the inception phase

The evaluation criteria for this phase are:

- Is there agreement among stake holders on the project's scope and cost/schedule estimates?
- Credibility of the cost and schedule estimates and risks of the development process.
- The depth and breadth of the architectural prototype.
- Actual expenditures for this phase versus the planned expenditure.

5.6. The Elaboration Phase

During the elaboration phase one deepens ones understanding of the problem domain and establishes an architectural foundation for the system, taking into account the use cases and the system constraints (e.g. architectural or performance constraints). A detailed project plan should be developed and the high-risk problems should be solved within the context of a demonstrable prototype. The stake holders should be convinced that the proposed architecture and project plan can be used to develop the complete system.

5.6.1. Deliverables of the elaboration phase

The deliverables of the inception phase typically include

- A *refined business case*.
- A *virtually complete use case model* containing typically 80% of the use cases with detailed descriptions of the use cases.

- The *requirements specification* should be refined including the functional as well as the non-functional requirements and constraints.
- A description of the *proposed system architecture*.
- A prototype demonstrating the *architecture*.
- A *revised risk list*.
- A *project plan* showing the evaluation criteria for each iteration. The project plan should cover the construction plan in some detail and the credibility of the estimates should be demonstrated.
- A preliminary *user manual* helps keeping the focus on the user requirements.

5.6.2. The evaluation criteria for the elaboration phase

The evaluation criteria for the elaboration phase are:

- Is the vision of the product stable?
- Is the architecture stable?
- Is there an executable prototype which demonstrates how the major risk elements have been resolved?
- Is the plan for the construction phase sufficiently detailed and accurate and are the estimates credible?
- Do the actual expenditures for this phase compare well with the planned expenditure?

5.7. The Construction Phase

The construction phase is a manufacturing phase in which the product is designed and implemented. The emphasis is on managing resources and controlling operations to optimize cost and quality.

The construction phase is broken down into several iterations focusing initially on determining the core architecture and then on designing and developing the components delivering the various use cases iteratively.

Separate modules are often developed by separate development teams. The work flows of these teams have to be coordinated and synchronized.

Each component should be documented and tested before it is signed off. A testing framework can be used to integrate the tests for the individual components.

5.7.1. Deliverables of the construction phase

The deliverables of the construction phase include

- The *software product*. It should be demonstrated that the product is stable and mature enough to be deployed on the client site.
- A *testing framework* for the product.
- The *user manual*.
- A *description of the current release*.

5.7.2. Evaluation criteria for the construction phase

The evaluation for the construction phase are

- Is the product release stable and mature enough for deployment?
- Are all stake holders ready for project transition into the user community.
- Are the actual expenditures versus the planned expenditures still acceptable?

5.8. Transition

During the transition phase the product is deployed on the client side and opened to the user community. Once the product is used problems feedback about problems or minor changes to the requirements Specification are the norm. The reasonable requests should be taken up and the requirements Specification and product should be revised to support these. Incomplete features should also be added and tested during this phase.

During the transition phase the system is subjected to the following:

- The user testing phase is usually called “beta testing”. This may require some user training. The system is tested in its real environment and against user expectations.
- If some or all of the use cases were previously delivered by some legacy system (manual or computerized), the new system usually runs for some time in simulation mode next to the legacy system. Any difference in behavior and results should be understood.
- Users and maintainers and potentially marketing staff are trained to fulfill their roles in the context of the new system. The client should potentially achieve self-supportability.
- If applicable, the product is rolled out to the marketing, distribution and sales teams and maintenance teams should be trained.

5.8.1. Evaluation criteria for the transition phase

The evaluation for the transition phase are

- Is the user satisfied with the product.
- Are the actual expenditures versus the planned expenditures still acceptable?

5.9. Feasibility Evaluations

The Rational Unified Process requires that at the end of each phase the feasibility of the project has to be re-evaluated. In this way the Rational Unified Process tries to ensure that investment into doomed projects is limited. This is particularly relevant in the light that more than 80% of large-scale, mission-critical projects are known to fail

5.10. RUP Life-Cycle Artifacts

RUP has evolved into a process which is not document driven. The main artifact should be, at all times, the software product itself. RUP requires that documentation should be minimized to those elements which add real value to the project. It makes recommendations for certain requirements, management and technical artifacts.

5.10.1. Requirements Artifacts

RUP is not envisaged to be requirements driven. The requirements themselves evolve throughout the software development process, but they are documented as A Business Case Document describing

- How the project is going to benefit the business.
- The financial context.
- The contract.
- The projected return on investment.

A Vision Document discussing the key requirements and limitations of the proposed system. This document is expected to evolve only very slowly during the development cycle. Requirements Specification Documents which are largely developed during the elaboration phase but which are refined further during the construction phase where the requirements have to be known in all its gory details.

5.10.2. Management Artifacts

RUP requires the following documents from project management:

- An *Organizational Policy Document* which documents
  - The generic process (which RUP expects to be RUP).
  - The way it has been adapted to the project.
- A Development Plan containing
  - The plan for the outermost iterations.
  - More detailed descriptions of the current iteration and the upcoming iteration.
- An Evaluation Criteria Document containing
  - The high-level requirements specifications.
  - The acceptance criteria.
  - The current technical objectives.
  - Iteration goals.
- A Release Description documenting the scope of the current release.
- A Deployment Document containing information useful for
  - Transition (deployment).
  - Training

    - Installation
    - Sales

- Status Assessment Documents containing the status of the project at the document date including

    - A progress section specifying work which has been completed.
    - Any issues regarding the current staffing situation
    - Expenditures incurred thus far and expenditures which are expected to be incurred in the current and next iterations.
    - A Results section specifying work which has been completed.
    - The Critical Risks identified at this stage.
    - Action items specifying what actions have to be taken to complete the next tasks or to correct certain problems encountered.
    - Post mortems of problems encountered.

5.10.3. Technical Artifacts

The following documents should be come from the technical team (architects, designers and developers):

- A *User's Manual* which is developed early in the life cycle to shift the emphasis more towards user requirements.
- *Software Documentation* which should preferable be in the form of self-documenting source code supported by a CASE tool which maintain the source code together with the design.
- A Software Architecture document describing

    - The overall structure of the software.
    - Its decomposition into sub-systems and high-level components.
    - The way in which the major processes are supported.
    - Critical interface definitions.
    - The Reasoning behind key design decisions.

6. Lessons from Extreme Programming

In 1996 Kent beck pioneered a software development discipline called *eXtreme Programming*. It is a light-weight methodology which has found more and more support among the software community, even though it was initially frowned upon in many quarters due to some of the unconventional ideas spread by its followers.

6.1. A Light-Weight Alternative

Software processes traditionally tend to be rather heavy often introducing a lot of bureaucracy and generating a lot of formal documentation. Lately several lightweight processes have started to gain a larger and larger following. These include the Crystal family of methods, Adaptive Software Development and eXtreme Programming (XP).

Of these XP has gained the largest following. In addition to having established itself as a lightweight methodology which is followed by many project teams world-wide, it has significantly influenced other, more established methodologies like the Rational Unified Process.

6.2. A Process for Continually Changing Requirements

XP is based on the recognition that requirements are continually changing throughout the development process as well as through the remaining life of the application.

XP was specifcally designed to be an effective process for continually changing requirements.

6.3. The 4 basic values of XP

XP is based on 4 core values which drive the entire process and its practices:

- Communication
- Simplicity
- Feedback
- Courage

6.3.1. Communication

Constant communication with the client is facilitated in XP by requiring that the customer should always be on-site. Communication between developers is driven partially by introducing the concept of pair programming. The customer decides what will be built and in what order.

6.3.2. Simplicity

Constant refactoring of the design and code is introduced in XP to ensure that simplicity is continuously maximized. Furthermore, XP requires a minimal set of non-code artifacts (status reports, formal requirements specifications, grant charts, …).

6.3.3. Feedback

Feedback is achieved in XP through

- Releasing increments to the customer in very short intervals.
- Introducing unit tests for each code unit.
- Running acceptance tests for each release.

6.3.4. Courage

Courage is about

- Being honest about what you can and cannot do.
- Doing the correct thing which you know works, even if it is not good for your popularity.

6.4. Business versus Development Cycles

As business changed relatively slowly, business cycles (the period over which the business requirements change significantly to warrant new versions of systems) used to be quite long – typically more 2 or more years.

With the rapid development in technology, improved telecommunication and more aggressive marketing the business cycle has decreased to less than a year, at times even to less than 6 months.

The development cycle must, of course, be shorter than the business cycle. You must not only match the business needs with your development cycle, but you must also leave room for testing, deployment and corrections.

Consequently the development cycle for each release must be 3 months or less. XP advocates a development cycle of only a few weeks with each development cycle delivering a fully functional, shippable increment of the software product.

6.5. The XP Phases

Extreme programming identifies the following life cycle phases:

- Exploration
- Planning
- Iterations to first release
- Productionizing
- Death

The initial exploration phase which leads into the first release is going to scope the project. This phase includes a feasibility study where one evaluates whether it is worth-while continuing with the project.

6.6. Iterative Development

XP is, like most modern processes, modeled around incremental releases which, in XP, define the beat of the project.

For each release XP goes through the following high-level phases: Exploration: During this phase the requirements for the iteration are explored and the customer selects the features which he/she would like to see in the next iteration. Commitment: The development team assesses the scope of the iteration, the required resources, the complexity of the features to be developed in that iteration and the feasibility of completing it in the allocated framework. Finally there will be a commitment to a clearly defined iteration with a list of the deliverable system features. Steering: This phase is about developing the features specified for the iteration. The progress is continuously monitored via testing and integration reports. If it becomes apparent that making the delivery date becomes a problem, the scope is reduced by removing features completely from the iteration. The argument is that a non-complete feature is, from the customers point of view, a non-existing feature.

Even though each iteration adds only one or a few features to the system, it represents a complete cycle. All activities typically present in a software development process are present in each iteration:

- Specification
- Design
- Implementation
- Testing
- Integration
- Deployment
- Documentation
- Training

but each cycle covers only a the restricted scope of one or a few features and is completed in a few weeks (typical XP recommendation is a two-week cycle per iteration).

6.6.1. Guidelines for writing stories

A story in XP is really a system feature or unit of functionality required by the client. XP introduces the following guidelines for stories (feature requirements):

- Each story must represent something of value to the customer.
- Stories must be understandable to the customer.
- Stories should be short.
- Stories must be implementable within one iteration.
- Stories must be testable.

6.6.2. Story should describe a business value

A story should be a system feature requirement which is of direct value to the customer. Some examples are

- A specific user task which is a direct element of a business work flow.
- A specific feature which enhances a business work flow in some way.
- A Specification of a constraint which should be satisfied like a performance constraint or a constraint specifying that certain system features should only be available under certain conditions.

Stories should be written by the customer and the development team should be able to estimate how long it will take and what resources it will require to deliver a story.

6.6.3. Stories must be understandable to the customer.

Textual descriptions should be in a language accessible to the customer. XP really requires that natural language should be used. I feel that UML can be seen as a graphical representation of natural language constructs and that UML diagrams can significantly clarify a system feature requirement.

6.6.4. Stories should be short.

XP requires that stories should be short and should only include the a small amount of detail. The consequence of this is that the feature requirements represented by the stories are inherently vague.

In the context of XP this is acceptable because it is assumed from the outset that the requirements

6.6.5. Stories must be implementable within one iteration

If stories are complex and large so that they cannot be implemented within a few weeks (typically 2 or 3), then they should be broken up into shorter stories.

6.6.6. Stories must be testable.

The client should be able to provide you with ways in which he/she is going to test the implemented story.

6.7. XP Activities

Extreme programming followers see themselves as engaging nearly exclusively in the following activities

- Listening
- Designing
- Coding
- Testing

6.8. The 12 XP practices

The 4 core values of XP are put into practice via 12 XP practices:

1. Planning
2. On-site customer
3. Simple design
4. Small releases
5. Metaphor
6. Testing
7. Refactoring
8. Pair programming
9. Collective ownership
10. Continuous integration
11. 40 hour week
12. Coding standards

6.8.1. Planning

During the planning phase one determines the features of the next release from from a combination of prioritized stories and technical estimates.

XP identifies 3 critical steps to iteration planning: Customers present user stories and the development team tries to understand the story. Team brainstorms engineering tasks. Designers identify the tasks required to implement the story. If their are multiple design solutions, the simplest is chosen. The customer partakes in this process, ensuring that the story is understood correctly and that the user requirements are met. Programmers estimate the work and sign up. Programmers or pairs of programmers sign up to take primarily responsibility for stories and make estimates for the time required to complete the individual tasks. Note that developers select and estimate their own tasks. Consequently developers will feel more commitment to the work and to the schedule.

XP recommends that an effort should be made to have programmers take responsibility for complete stories and that tasks within a story are not assigned to different programmers.

If the story is large the principal holds in slightly modified form. A small team of programmers takes on the story and team members are, in the context of a particular iteration, not involved in any other stories.

The reason behind this is that if developers are involved in multiple stories, then one developer could be working on one story, having it virtually complete bar the task(s) assigned to another developer who is involved virtually completing another story. The end result of this could be that one has a large number of virtually complete stories – and nothing to show to the customer.

6.8.2. On-site customer

Current statistics still show that about 80% of the cost of errors is due to errors in the system requirements. XP tries to address this by having the customer commit a user/domain expert full-time to the development process.

Through this XP envisages that the developer and customer work on the project.

The main responsibility of the on-site customer are

- To help define the system.
- To write tests.
- To answer questions.

XP is of the view that formal requirements specifications are very expensive to obtain and that very often they do not communicate the requirements very well.

Instead, XP claims that it is much more effective to discuss the requirements with the client verbally as the project continues.

6.8.3. Simple design

Since the stories for each iterations are short, the design for them can usually be very simple. XP suggests going through a rapid design phase using UML sketches where the design should, in most cases require less than half an hour. If, at any stage, it becomes apparent that a design can be simplified, this simplification should be done on the spot (see refactoring below).

I feel that the design should not be rushed too much. Sure, one can reach design paralysis when trying to achieve the perfect design but a good design can simplify the implementation and maintenance significantly.

If you do follow the XP approach, then I suggest you give all developers a basic foundation in design patterns. Design patterns are generic, reusable design elements which can be applied to a wide range of problems and, particularly in the context of rapid design one should have a box full of design patterns handy.

In addition to giving your team members a basic background in general design patterns, you should also include some background on patterns addressing the added complexity and performance con-

siderations of distributed systems.

When team members discover a design pattern themselves it should be inserted into a patterns repository maintained by the team and one should consider introducing now and then seminars or informal discussions on design patterns.

6.8.4. Small releases

The outermost XP cycle is the release. Small and frequent releases provide early benefit to the customer and early feedback to the developers.

If you have complex systems which cannot be used in practice before they are complete, one can often develop the new system in such a way that it integrates with the existing system and that it slowly but surely takes over more and more responsibility from the older system.

For example, one could replace the user-interface layer with a modern GUI. This itself could provide a direct benefit to the client in terms of increased productivity or decreased training costs for new staff members.

Similarly, one could replace those functionalities in the existing system which are not working correctly or which need to accommodate new business rules.

6.8.5. Pair programming

Pair programming is perhaps that aspect of XP which has caused the biggest stir. The idea is that at any time two developers sit on one machine.

The developer sitting behind the keyboard is the driver and the person sitting next to him/her is called the partner. The partner continuously checks the code and tries to put it in the requirements context. The roles can be reversed at any time between the pair members.

XP claims that pair programming has the following benefits:

- Better quality code.
- Higher productivity.
- Lower project risk.
- Better adherence to standards and recommended practices.
- Makes programming more fun.
- General increase in skills levels.

6.8.5.1. Better quality code

The probability of developing a good implementation is certainly higher with two heads and the probability of not spotting a logical or technical error is decreased with two persons thinking about an implementation.

6.8.5.2. Higher productivity

The increase in productivity arises from

- Less mistakes being made.
- Solution to problems are found faster with two persons discussing a problem.

6.8.5.3. Lower project risk

The decrease in project risk originates from

- The increase in design and code quality.
- The fact that at least two persons are intimately familiar with any piece of code reduces the exposure to single individuals.

6.8.5.4. Better adherence to standards and recommended practices

Generally it is found that persons working in pairs are less likely to ignore coding standards or development guidelines.

6.8.5.5. Makes programming more fun

Working in pairs is often experienced as more fun. People enjoy solving problems together and the odd shared joke makes life a lot more enjoyable.

6.8.5.6. General increase in the project team's expertise level

Particularly if pair members are exchanged now and then, each project team member may ultimately learn from every other member, thereby enhancing the general skills level of the project team. It may be beneficial to, at times, team off more experienced developers with younger colleagues. The more experienced colleague

- Potentially has a deeper and wider understanding of the business.
- Often has had more exposure with ideas which do not work.
- Has exposure to design solutions which have been applied successfully in the past.

The younger college, on the other hand

- May be more up to date with the latest technology developments and latest academic trends.
- Is not predisposed by a long history of development and may come up with new, original ideas.

6.8.6. Testing

The first guideline recommended by XP is WRITE THE TEST FIRST. There are very many reasons why it is beneficial to write the test before writing the component you are testing:

Test represents low-level, precise requirements Specification. If the developer has a complete test, he/she has complete requirements Specification. More focused development process. Having complete requirements Specification removes any ambiguities about what the developer should develop. The development process is thus a streamlined process guided by the testable requirements Specification. Test not biased by implementation. If the developer writes the test after the code has been developed, the test may be strongly biased by what has been developed. The developer will simply test what they have just implemented. This may not be the actual test required and the resulting component could pass the test but fail the customer requirements, Test less likely to be neglected. After the component has been completed developers may get the comfortable feeling of having virtually completed the task. The test is a minor final plutocratic overhead which may not get the attention which it should.

Who should write the test? XP requires that the tests should be conceptually devised by the on-site customer. The developer then puts the tests into code, clarifying any uncertainties as he/she does

that.

So, what should be tested? XP's guideline is simply: test everything that can break. There are some extremely simple things which really cannot break. Those do not require testing. Everything else does.

What if there are so many scenarios that writing an exhaustive test would take forever? Typically this is an indication that the component which you are testing is too complex and that it should be broken up into simpler sub-components which require simpler tests.

What should be done if a component fails in real life but passes its tests? Obviously the component breaks and the tests do not test everything that can break. Developers should update the test first, then the design and code.

6.8.7. Continuous integration

A process in which all the system components are first built and, once this has been completed, goes through an integration phase where the system is assembled from these components may end up in deep trouble. Integration problems can be severe and very costly to fix.

XP suggests that the system should be rebuilt and reintegrated continuously, i.e. every time a new implementation task has been completed. This may be several times a day.

If rebuilding the system takes excessively long, its structure should be re investigated and enhanced so that the system can be rebuilt in a few minutes.

6.8.8. Collective ownership

XP believes in collective ownership of code,i.e. that anyone may change any piece of code at any time (assuming, of course, that nobody else is working at the same time on that piece of code – this should be guaranteed by using a simple code repository with facilities for booking code into and out of the repository).

The potential problem with this approach is that the person who makes the modification may perhaps misunderstand the design and code he/she is modifying. XP addresses this by requiring

- The existence of unit tests for every unit of code.
- The requirement that designs and code should always be as simple as possible.
- Very active communication between all team members.

6.8.9. Refactoring

Refactoring is about improving the design or the implementation of completed code without changing the behavior of the code. Often the resultant design and code is much more compact and much less complex than the original design and code.

This makes it difficult to sell to project managers because time is invested without immediate payoff. Particularly if critical deadlines are looming, refactoring is not very popular.

Still, even in the case where critical deadlines have to be met, refactoring can actually speed up the remaining development.

It is virtually insane to refactor production code if there are no tests in place. In fact, if there are no tests in place, project managers may have to introduce the unpopular rule that nothing in production is modified until the tests have been written.

One potential danger of refactoring is that as one digs further and further into the design and code, one fends more and more problems and modifies more and more, slowly digging ones own grave.

To avoid this one should always refactor a small step at a time, retesting after every design/code change.

6.8.10. 40 hour week

XP believes that the quality of designs, code and tests decreases exponentially as people get more tired and more de-motivated. Hence long hours can cost the project more in lost time by ways of bug fixes, redevelopment and failures due to non-thorough tests, than what the extra hours of work add to the project.

6.8.11. Coding standards

Since XP believes in collective ownership, the code developed by one developer should be accessible by every other developer in the team. To this end the team should introduce a common coding standard. This coding standard should

- Encourage simple, clearly understandable code.
- Introduce a uniform code-look-and-feel across the project.
- Help junior developers avoid mistakes.

The standards should cover the regular code as well as the tests.

6.8.12. Metaphor

XP suggests that it is beneficial if one can find a metaphor for a system. The metaphor represents a conceptual equivalent though not a literal equivalent.

6.9. The 4 variables

XP identifies the 4 variables of a software process as

- Cost
- Quality
- Time
- Scope

Changing any one of these variables will result in a change in at least one of the others. You can lock one or two of these variables and still change the others, but locking three of the variables results in the fourth variable be fixed too.

The relationship between the variables is, however, non-linear and at times not predictable. Doubling the cost (by, for example, hiring more people) does typically not half the development time and as you add more people the benefit curve will start #attening off.

6.9.1. Increasing the project cost

Keeping quality and scope fixed, one can reduce the time required for a project by increasing the cost.

The cost may be increased by Increasing the skills resources: Adding more people to the project will reduce the time required for the project, but doubling the number of people on the project will typically not half the time. With more people the communication overheads increase. Furthermore, the initial effect of adding more people may even be to slow down the rate of development, because ini-

tially they may not be productive and furthermore will require time resources from the existing staff to get familiarized with the project. Thus, adding more people to a project which is behind its deadline may shift the delivery date out even further. Paying for longer hours: This can be useful for solving the odd crisis, but generally the morale sinks and tired people are less productive and make more mistakes. Investing in more powerful software tools: The right tools can make a significant contribution to the project. However, they require training time and often the tools do not live up to their promises. Investing in more powerful hardware tools: If a project looses significant amounts of time due to the time required to recompile the increments or due to running testing frameworks it is most probably wise to spend some money on hardware. Furthermore, relatively modest amounts spend on faster workstations with more memory and bigger screens for these workstations can increase productivity significantly by increasing the enthusiasm of team members.

6.9.2. Trading off scope for time

Scope creep is notorious for putting projects further and further behind schedule. When adding scope it is very difficult to estimate the effect this will have on time. XP tries to address this by enforcing short iterations with a delivery at the end of each iteration. At the end of each iteration the effect on time of adding scope is evaluated, and, if necessary, scope is sacrificed to still remain on schedule.

In general XP suggests that you hardly ever slip deadlines or delivery dates for iterations. Instead you should rather sacrifice scope and still make your delivery.

6.9.3. Quality

XP differentiates between external, perceived quality and internal quality.

6.9.3.1. External quality

The external, perceived quality includes elements like the look-and-feel of user interfaces. Increasing external quality can have a significant effect on the other three variables, scope, time and cost. If the client wants certain improvements in the external quality, try and make them choose between this and some functional requirements and hopefully you will get all the functional requirements in place before optimizing the externally perceived quality.

6.9.3.2. Internal quality

Unlike external quality, internal quality typically decreases project cost. The reasons is simply that if you invest in a good quality design and in testing at every level, you going to spend much less on debugging and developing fixes at a later stage.

6.10. Monitoring Project Status and Quality

One of the big benefits of XP is that it should be simpler to monitor the status of the project as well as the quality of work delivered. This is largely due to

- the process being formulated around short term deliverables verified directly by the customer in a production environment.
- Tests being available at all levels and being run every time anything is changed.

At any stage one has thus a clear view of what has been developed,tested and accepted by the customer and the team is never in the process of doing long stretches of work without acceptance testing.

The rate at which stories are being delivered and the proportion of acceptance tests passed give a clear indication of productivity and quality.

7. The Agile Manifesto

There is a growing number of so-called OO gurus who canvas for a more light-weight approach. Many of these light-weight methodologies have been developed in-house and have not been publicized widely. Others, including the following, are well known:

- Extreme programming
- Crystal methodologies
- Adaptive Software Development
- Feature-Driven Development
- Dynamic System Development

Some of the proponents of these light methodologies including

- Kent Beck
- Martin Fowler
- Alistair Cockburn
- Robert C. Martin
- Ron Jeffries

and 12 others met in February 2001 where they produced a document which they called the Agile Manifesto. It reads as follows:

> We have come to value
>
> - Individuals and interactions over processes and tools.
> - Working software over comprehensive documentation.
> - Customer collaboration over contract negotiation.
> - Responding to change over following a plan.
>
> That is, while we value the items on the right, we value the items on the left more.
>
> We follow the following principles:
>
> - Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
> - Welcome changing requirements, even in late development. Agile processes harness change for the customer's competitive advantage.
> - Deliver working software frequently, from a couple of weeks to a couple of months, with preference to the shorter timescale.
> - Business people and developers work together daily throughout the project.
> - Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- Working software is the primary measure of success.
- Agile processes promote sustainable development. The sponsors, developers and users should be able to maintain a constant pace indefinitely.
- Continuous attention to technical excellence and good design enhances agility.
- Simplicity – the art of maximizing the amount of work not done – is essential.
- The best architectures, requirements and designs emerge from self-organizing teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly.

8. The Capability Maturity Model for Software

The Capability Maturity Model (CMM) was developed over a number of years and was published in 1991 by Mark Paulk and Charles Weber with the current CMM version 1.1 being published in 1993.

It grew out of the realization that software projects continued to have a very high failure rate and that the productivity and quality gains promised by many new methodologies and technologies often did not materialize. Software projects continued to be often excessively late and over budget.

CMM claims that this is caused mainly by the organizations inability to manage the software process. According to CMM, the benefts of the improved tools are often lost due to undisciplined and chaotic project management.

CMM aims to help organizations to

- Evaluate their process maturity.
- Help gain control of processes for developing and maintaining software.
- Help evolve toward a culture of sound software engineering and management.

8.1. Characteristics of Immature Software Organizations

CMM identifes a number of features common among immature software organizations:

Ad-hoc software processes are which are generally improvised by developers and project management. Software processes are not enforced. At times immature software organizations have specifed software processes but these are not rigorously followed or enforced. Reactionary management and development. Managers are usually focused on solving immediate crises, stepping from one crises to the next. Non-adherence to schedules & budgets because of the non-structured development process and because of an inability to provide accurate estimates. Lack of quality prediction because of not having an objective process for measuring quality. Quality degradation under hard deadlines due to software quality reviews and testing being hastened or even skipped. Inability to effectively improve processes because of being unable to objectively evaluate benefts of the improvements.

8.2. Characteristics of Mature Software Organizations

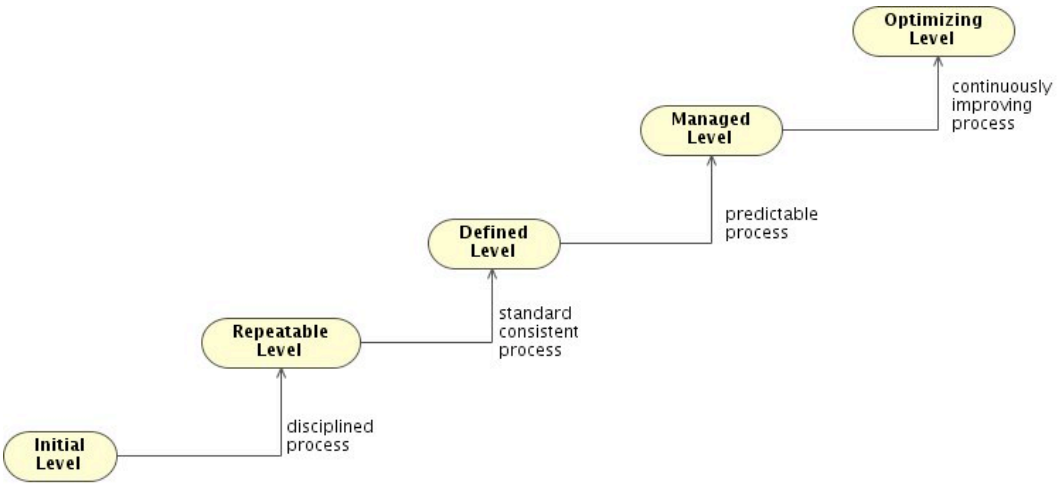
On the other hand, CMM claims that the following characteristics indicate a mature software organ-

ization: Effective management of software development and maintenance processes. Software processes are generally adhered to due to the processes being usable and tailored for the companies requirements. The benefts of the process are effectively and accurately communicated to existing staff and new employees which tend to adhere to them naturally because they understand the benefts of following the prescribed process. Clear Role defnitions where the roles in the project context as well as in the wider organizational context are accurately defined. Accurate, objective quality monitoring of both the software product as well as the process which produced the software. Ability to provide realistic schedules and budgets largely based on historical performance. Ability to effectively evolve processes through the introduction of pilot projects which are subjected to productivity and quality evaluations as well as a solid cost-benefit analysis.

8.3. CMM Maturity Levels

The Capability Maturity Model advocates continuous process improvement based on small evolutionary steps. To provide a course-grained measure the maturity of an organization's software processes CMM introduces the 5 maturity levels shown in Figure 11.1, “The 5 CMM levels of Software Process Maturity.” .

Figure 11.1. The 5 CMM levels of Software Process Maturity.



An organization would work itself sequentially through these levels. Skipping of levels is counter productive because each level provides the foundation for the next level.

8.3.1. What is a maturity level?

The maturity levels represent not only the maturity level of an organization's software processes, but also provide a measure of the capability of these processes.

Each maturity level is viewed as a well-defined evolutionary plateau toward achieving a mature software process. Each maturity level comprises as set of process goals that, when satisfed, stabilize an important component of the software process.

8.3.2. Level 1: The Initial Level

The successes achieved by level 1 organizations are attributed by the CMM to having

- an exceptional manager and/or
- a seasoned, effective software team,

i.e. to the heroic achievements of certain individuals and CMM claims that these successes cannot be repeated unless the same competent individuals are assigned to future software projects.

Level 1 capability is thus attributed to individuals and not to the organization. Problems typically experienced by level 1 organizations include

- Schedules and budgets are regularly overrun.
- Development dynamics deteriorates into crisis management where the dynamics carries the development team from one crisis to the nest.
- During these criseses the process is being abandoned resulting in the production of poor quality designs and code.

8.3.3. Level 2: The Repeatable Level

The characteristics of organizations having reached the repeatable level are identifed by the CMM as

- Project management policies are introduced and implemented on a per project basis.
- The process introduced and enforced for the project is
  - documented,
  - trained,
  - measured and
  - able to improve.
- Planning and managing new projects is based on experience with similar projects resulting in realistic schedule and cost estimates.
- Integrity control is applied to software requirements and the work products developed to satisfy them.
- Installation of basic software management controls including the ability to track
  - schedules,
  - costs and
  - functionalities.

  i.e. throughout the development process the schedule and cost estimates are baselined – the estimates are updated with the latest information and communicated to the customer.
- Introduction and enforcement of software project standards.

The result is that the planning and tracking of software projects is stable and that early success can be repeated without relying on the heroic effort of certain individuals.

Note that for level 2 organizations the processes may differ across projects and that there need not be any linking between the policies guiding different projects.

8.3.4. Level 3: The Defined Level

At the defined level the organization standardizes its software development processes.

CMM characterizes a level-3 compliant organization by

- The standard process for developing and maintaining software across the organization is documented including
  - software engineering processes and
  - management processes

  and these processes are integrated into a coherent whole. This enables management to obtain objective status information across all projects.
- A group, the software engineering process group, takes ownership of the organization's process activities.
- An organization-wide training program ensures that all role players are aware of their roles and are able to fulfll their duties.
- For each project the organization's standard process is tailored to fulfll the specifc project requirements. The resultant tailored process is called the defined software process. The defined software process includes
  - readiness criteria,
  - process input defnitions,
  - standards and procedures for performing the work,
  - verifcation mechanisms,
  - outputs and
  - completion criteria.

8.3.5. Level 4: The Managed Level

The software process capability of organizations which have reached the managed level is quantifable and predictable. The process is continuously measured and stable and corrective action can be taken when process limits are violated.

The CMM requires that level 4 organizations

- Set quantitative goals for both
  - Software products
  - Software development and maintenance processes.
- Software processes are instrumented with well-defined and consistent measurement processes for productivity and quality.
- Management continuously measures productivity and quality across all defined software projects and maintains an organization-wide software process database for this information.

- Projects obtain control of their processes by narrowing the variation in process performance. This enables management to distinguish between standard process variations (random noise) and systematic problems which should be addressed.
- The risks involved moving up the learning curve of a new application domain are known and carefully managed.

8.3.6. Level 5: The Optimizing Level

Organizations which have reached the optimized level are focused on continuous organization-wide process improvements.

- The organization has means to identify strengths and weaknesses of a process pro actively enabling it to introduce measures which could possibly prevent defects.
- If defects occur, they are analyzed for their causes. The process is modifed to prevent reoccurrence of defects and the lessons learnt from that project is distributed to other projects.
- It can perform cost-beneft analysis of new technologies and proposed process changes.
- One of the focuses of level-5 organizations is to minimize waste, i.e. the amount of work which has to be redone.
- Technology and process improvements are planned and managed as ordinary business activities.

Note that levels 4 and 5 are conceptually based on statistical techniques.

8.4. How is maturity expected to influence performance?

CMM expects that more mature organization are in a better position to predict a project's ability to meet its goals.

Projects in level-1 organizations experience wide variations in achieving the targets for

- cost,
- schedule,
- functionality and
- quality.

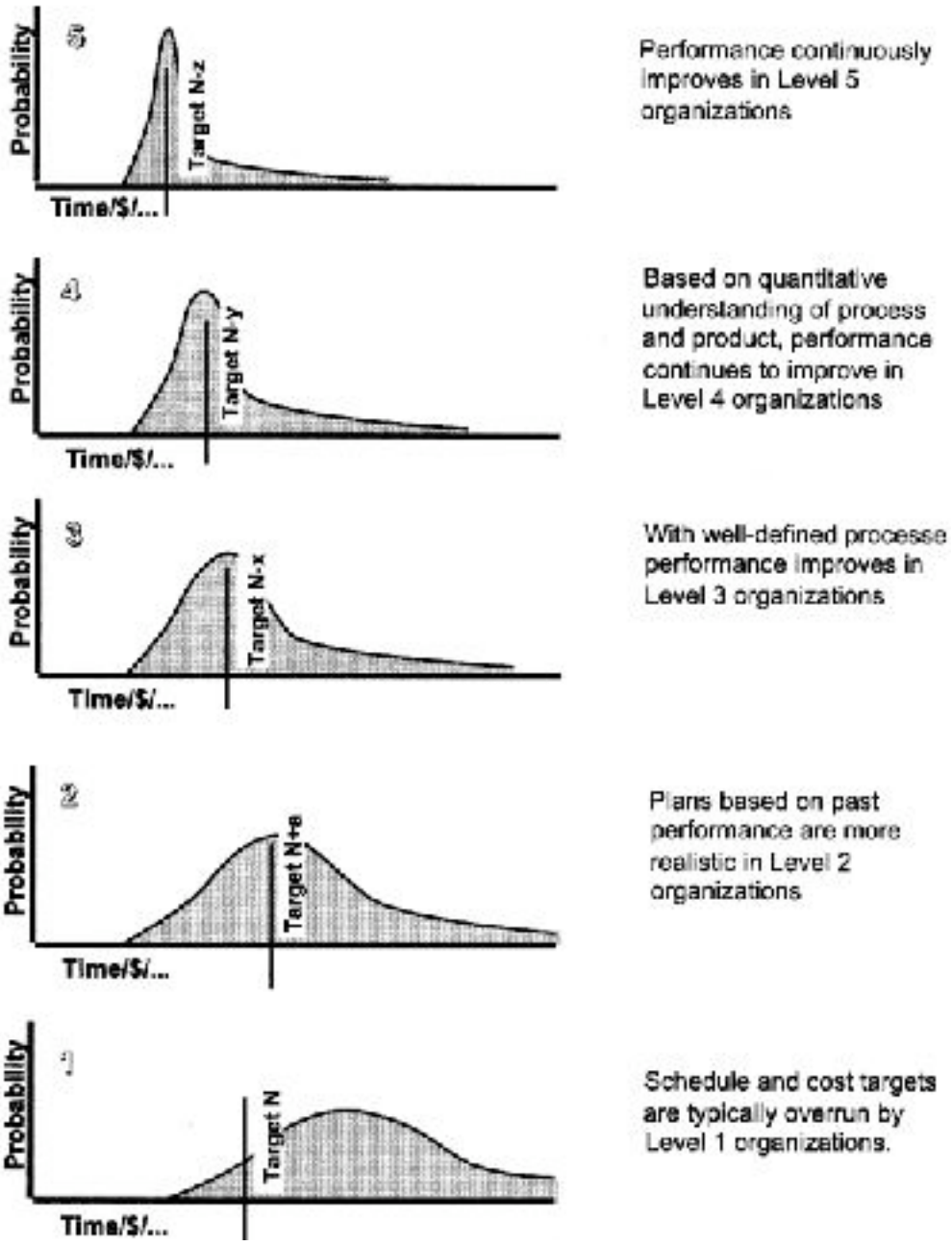
As the maturity increases the organization is expected to experience

- A decrease in difference between targets and actual achievements.
- A decrease in the variability of the actual results around targets.
- A general increase in productivity and quality.

The expected tendencies are shown in Figure 11.2, “Process capability and performance predictions taken from The capability Maturity Model for Software published by Paulk, Curtis, Chrissis and Weber.”.

Figure 11.2. Process capability and performance predictions taken from The

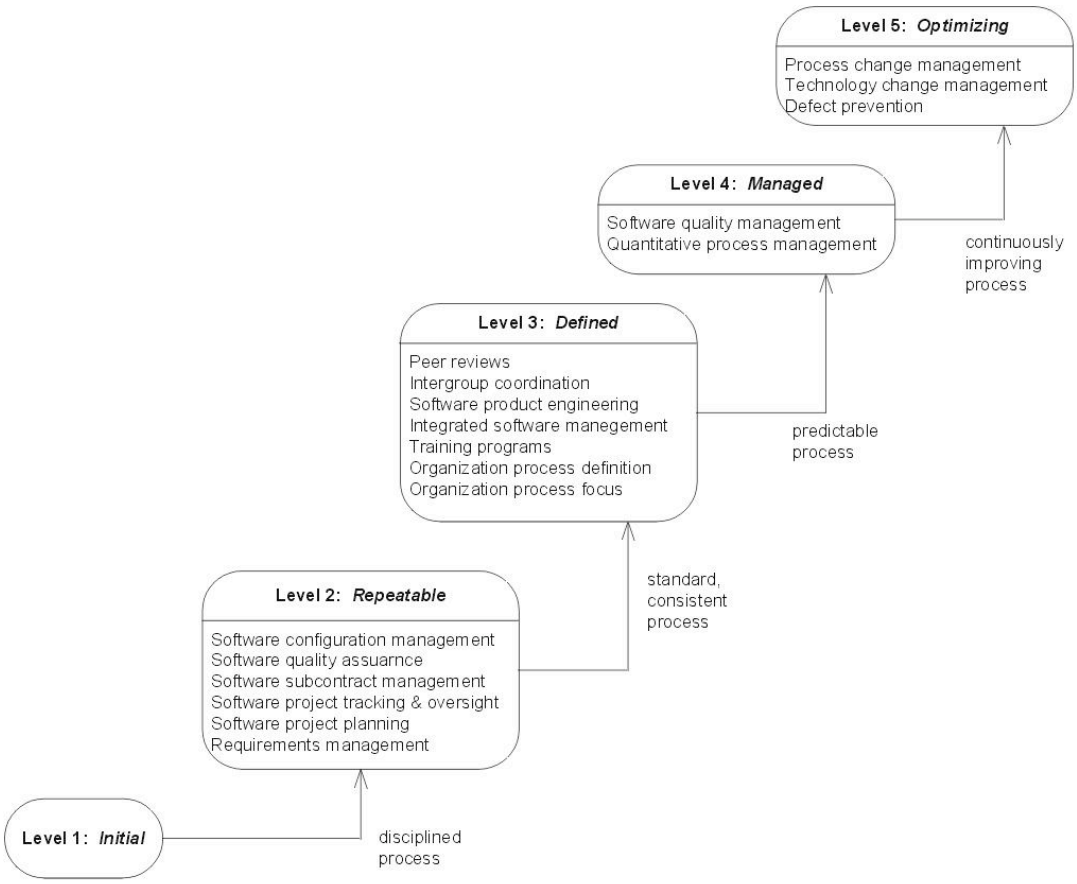
capability Maturity Model for Software published by Paulk, Curtis, Chrissis and Weber.



8.5. Key process areas of the CMM

The process areas for the individual CMM levels identify the issues that must be addressed to achieve the applicable maturity level. Figure 11.3, “Key process areas for the different CMM levels.” illustrates the key process areas for the different CMM levels.

Figure 11.3. Key process areas for the different CMM levels.



8.5.1. Key process areas for level 2 maturity

The level 2 process areas focus on basic project management controls. They include:

Requirements Management which aims to establish a process for

- requirements elicitation,
- requirements verifcation,
- requirements specifcation and
- requirements change management.

Software Project Planning which aims to establish reasonable plans for the software engineering and the managing of the software project. Software Project Tracking and Oversight is about making projects monitorable in order that management can take effective action when the software project's performance deviates signifcantly from the plans. Software Subcontract Management aims to establish a process to evaluate, select and manage subcontractors effectively. Software Quality Assurance aims to provide management with appropriate visibility into the software products being developed and process used to develop them. Software Confguration Management focuses on establishing and maintaining all products of the software project throughout the project's life cycle.

8.5.2. Key process areas for level 3 maturity

The key process areas address issues which help the organization to establish an infrastructure which institutionalizes effective software engineering and management processes across all projects. They include:

Organization Process Focus establish the organizational responsibility for software process activities enabling the organization to improve its overall software process capability. Organization Process Defnition includes

- Developing and maintaining software process assets that improve process performance across projects.
- These software process assets should provide a basis for obtaining meaningful information for quantitative process management.
- The software process assets provide a stable foundation that can be institutionalized via training and common information and guideline repositories.

Training Programs are introduced to develop the skills and knowledge of individuals so that they can perform their roles effectively and efficiently. Integrated Software Management is applied on a per-project basis. It aims to integrate the software engineering and management aspects into a coherent, defined software process that is tailored from the organization's standard software process. The tailoring is based on

- the business environment and
- the technical needs

of the project. Software Product Engineering describes the technical activities of the software project like

- Requirements analysis, specifcation and management.
- Architectural design process.
- System design.
- Implementation.
- Testing.

It aims to establish an effective, well-defined engineering process which is applied consistently to integrate all software engineering activities to produce correct and consistent software products effectively and efficiently. Intergroup Coordination aims to establish an infrastructure enabling different software engineering groups to interact and collaborate effectively. Peer Reviews aims to decrease the defect level of software early and efficiently. In addition the understanding of individual products of the software process is distributed. Typical methods used include inspections and structured walkthroughs.

8.5.3. Key process areas for level 4 maturity

The key process areas for level 4 address issues which introduce quantitative measurability of the software process and its products. They include Quantitative Process Management aims to the process performance of a measurably stable process quantitatively. Causes of systematic variations from targets are analyzed and corrective measures are introduced. Software Quality Measurement aims to

- Develop a quantitative understanding of the quality of the project's software products.
- Achieve specifc quality goals.

8.5.4. Key process areas for level 5 maturity

The key process areas of level 5 address issues which help the organization to establish the infrastructure to continually improve the software process. They include: Defect Prevention aims to

- Identify the causes of defects before they occur and prevent the defects from occurring.
- Initiate a process of introducing changes to the defined software process.

Technology Change Management focuses on performing innovation and introducing innovations efficiently in order to remain competitive in a changing environment. It is responsible for

- Identifying benefcial new technologies including tools, methods and processes.
- Transfer benefcial managers in a structured, well managed way into the organization.

Process Change Management is responsible for continually improving the software processes used in the organization in order to

- improve software quality and
- increase productivity.

8.6. Practices of High-Maturity Organizations

Many companies have introduced the CMM into their organization and in February 200 a study that 60 companies (44 in the US, 14 in India, 1 in Australia and 1 in Israel) have reached CMM level 4 and 37 companies (27 in the US and 10 in India) have reached CMM level 5.

Most high maturity organizations build real-time applications, particularly for the defense industry. Nevertheless, the list includes general software development companies, and companies from the aviation, fnancial, telecommunications sectors.

Below we list some general management practices followed by most high-maturity organizations:

- The software process improvement programs are aligned with Total Quality management (TQM) initiatives at the organizational or enterprise level.
- They typically have achieved ISO 9001 certifcation.
- They use other standard management models for issues which fall outside the scope of the Software CMM like
    - ISO 12207 for software life cycle processes.
    - People CMM.
    - Software Acquisition CMM.
- Use cost models such as COCOMO and Price-S.
- Perform systematic risk management and continuously maintain a top-10 risk list.
- Use the Project Evaluation and Review technique (PERT).
- Employ chief architects and chief engineers.

Bibliography

[Alexander-Ishikawa-Silverstein-Jacobson-1977b] Christopher Alexander, Sara Ishikawa, Murray Silverstein, and Max Jacobson. *A Pattern Language*. Oxford University Press. 1977.

[Alexander-1979] Christopher Alexander. *The Timeless Way of Building*. Oxford University Press. 1979.

[Anderson-2003] Kenneth M. Anderson. *Object Design: Roles, Responsibilities and Collaborations*. Addison-Wesley/Pearsons Education. 2003.

[Ashmore-Henson-Chancellor-Nelson-2004] Perryn Ashmore, Joel Henson, Jeff Chancellor, and Mark Nelson. “Is you enterprise architecture all it can be?”. Business Process Trends. June 2004.

[Bass-Clements-Kazman-2003] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. second edition. Addison Wesley. 2003.

[Beane-Giddings-Silverman-1984] J. Beane, N. Giddings, and J. Silverman. “Quantifying Software Designs”. Proceedings of the Seventh International Conference on Software Engineering. 314-322. March 1984.

[Beck-Cleal-1999] Kent Beck and Dave Cleal. “Optional Scope Contracts”. 1999.

[Beck-Fowler-2000] Kent Beck and Martin Fowler. *Extreme Programming Explained*. Addison Wesley. 0201710919. 2000.

[Ben-Abdallah-et.al-2004] H. Ben-Abdallah, N. Bouassida, F. Gargouri, and A. Ben-Hamadou. “A UML Based Framework Design Method”. Journal of Object Technology. 3. 8. 2004.

[Booch-1994] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin Cummings. 1994.

[Booch-Rumbaugh-Jacobson-1999] Grady Booch, Jim Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language Reference Guide*. Addison-Wesley. 1999.

[Brickley-Smith-Zimmermann-2001] James Brickley, Clifford W. Smith, and Jerold Zimmermann. *Managerial Economics and Organizational Architecture* . McGraw-Hill. 2001.

[Buschmann-Meunier-Rohnert-Sommerlad-Stal-1996] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture*. A system of patterns. John Wiley & Sons. 1996.

[Cantor-1998] Murray R. Cantor. *Object-Oriented Software Management*. Wiley. 1998.

[Carlson-2001] D. Carlson. *Modeling XML Applications with UML*. Addison-Wesley. 2001.

[Clements-Kazman-Klein-2002] Paul Clements, Rick Katzman, and Mark Klein. *Evaluating Software Architectures*. Methods and case studies. Addison-Wesley. 2002.

[Coad-Yourdon-1991a] P. Coad and E. Yourdon. *Object Oriented Analysis*. Yourdon. 1991.

[Coad-Yourdon-1991b] P. Coad and E. Yourdon. *Object Oriented Design*. Yourdon. 1991.

[Cockburn-2001] Alister Cockburn. *Writing Effective Use Cases*. Addison-Wesley. 2001.

[Coleman-Arnold-Bodoff-1994] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremes. *Object-Oriented Development*. The Fusion Method. rentice-Hall. 1994.

[Duell-1997] *Object Magazine*. Michael Duell. “ Non-software examples of software design patterns ”. 54. July 1997.

[Eriksson-Penker-2000] Hans-Erik Eriksson and Magnus Penker. *Business Modeling with UML*. Business Patterns at Work. Wiley. 2000.

[Ethiraj-Levinthal-2004] Sendil K. Ethiraj and Daniel Levinthal. "Modularity and innovation in complex systems". Management Science. 159-173. 50. 2004.

[Fowler-2000] Martin Fowler. *UML Distilled*. second. Addison-Wesley. 2000.

[Frankel-2003] David S. Frankel. *Model Driven Architecture: Applying MDA to enterprise computing*. John Wiley & Sons. 2003.

[Galbraith-Downy-Kates-2001] Jay Galbraith, Diane Downy, and Amy Kates. *Designing Dynamic Organizations: A Hands-On Guide for Leaders at All Levels* . American Management Association. November 2001.

[Gamma-Helm-Johnson-Vlissides-1995] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley. 1995.

[Harel-1988] *CACM*. 31. 5. May 1998. D. Harel. " On Visual Formalism ". 26-49.

[Jacobson-Christerson-Johnson-Overgaard-1992] Ivar Jacobson, M. Christerson, P. Johnson, and G. Overgaard. *Object-Oriented Software Engineering*. Addison-Wesley. 1992.

[Jacobson-Booch-Rumbaugh-1999] Ivar Jacobson, Grady Booch, and Jim Rumbaugh. *The Unified Software Development Process*. Addison Wesley. 1999.

[Jeffries-Anderson-Hendrickson-2000] Ron Jeffries, Ann Anderson, and Chat Hendrickson. *Extreme Programming Installed*. Addison-Wesley. 0201710919. 2000.

[Jacobson-Ericsson-Jacobson-1995] Ivar Jacobson, Maria Ericsson, and Agnetta Jacobson. *The Object Advantage*. Business Process reengineering with Object Technology. Addison-Wesley Longman. 1995.

[Kao-ReferenceArchitectures] James Kao. *Reference Architectures*. A foundation for robust J2EE systems. The Middleware Company.

[Katzman-1994] Rick Katzman. *Toward Deriving Software Architectures from Quality Attributes*. 1994.

[Krasner-Pope-1988] *Journal of Object-Oriented Programming*. 1. 3. Glenn E. Krasner and Stephen T. Pope. " A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80 ". 26-49.

[King-1995] W.R. King. "Creating a strategic capabilities architectures". Information Systems Management. 67-69. 12. 1. 1995.

[Kruchten-2000] Philip Kruchten. *The Rational Unified Process*. An Introduction. second. Addison Wesley. 2000.

[Leavitt-2004] Harold J. Leavitt. *Why Hierarchies Are Here to Stay and How to Manage Them More Effectively*. Harvard Business School Press. November 2004.

[Leffingwell-Widrig-2000] Dean Leffingwell and Don Widrig. *Managing Software Requirements*. Addison-Wesley. 2000.

[Marshall-2000] Chris Marshall. *Enterprise Modeling with UML*. Designing successful software through business analysis.

[Meyer-1991] Bertrand Meyer. *Design by Contract*. Prentice Hall. 1991.

[Meyer-1992] *Computer (IEEE)*. 25. 10. October 1992. Bertrand Meyer. " Applying Design by Contract ". 40-51.

[Mindful-Business-And-Leadership] Mindful Business & Leadership. *Governing Ideas of Any Or-*

ganization or Individual.

[Moncrieff-Smallwood-1997] James Moncrieff and Janet Smallwood. *Ideas for the New Millenium*. FT mastering. Pearson Proffesional. 1997.

[Nadler-Gerstein-Shaw-1992] David A. Nadler, Marc C. Gerstein, and Robert B. Shaw. *Organizational Architecture: Designs for Changing Organizations* . Jossey-Bass. May 1992.

[Nickols-1997] Fred Nickols. *Measurement-Based Analysis*. Hooking what you do to the bottom line. The Distance Consulting Company. 1997.

[OMG-1995] The Object Management Group. *BOMSIG White Paper*. OMG. 1995.

[Pande-Neumann-Cavanaugh-2000] P.S. Pande, R.P. Neumann, and R.R. Cavanaugh. *The Six Sigma Way*. How GE, Motorola, and other top companies are honing their performance. McGraw-Hill. 2000.

[Ponnambalam-1997] K. Ponnambalam. "Characterization and Selection of Good Object-Oriented Design". OOPSLA. 1997.

[Rausch-2002] Andreas Rausch. "Design by Contract + ComponentWare = Design by Signed Contract". Journal of Object Technology. 1. 3. 2002.

[Rosenberg-Scott-1999] Doug Rosenberg and Kendall Scott. *Use Case Driven Object Modeling with UML: A Practical Approach*. Addison-Wesley Professional. 1999.

[Rosenberg-Scott-2000] Doug Rosenberg and Kendall Scott. "Driving design with use cases". Software Development. 2000.

[Rumbaugh-Blaha-Premerlani-1991] Jim Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and F. Lorenzen. *Object-Oriented Modeling and Design*. Prentice-Hall. 1991.

[Schonberger-1982] Richard J. Schonberger. *Japanese Manufacturing Techniques*. Nine hidden lessons in simplicity. Free Press. 1982.

[Shaw-Garlan-1996] Mary Shaw and David Garlan. *Software Architecture*. Perspectives on an emerging discipline. Prentice Hall. 1996.

[Unhelkar-2003] Bhuvan Unhelkar. *Process Quality Assurance for UML-Based Projects*. Addison-Wesley. 2003.

[Wirfs-Brock-McKean-2002] Rebecca Wirfs-Brock and Alan McKean. *Object Design: Roles, Responsibilities and Collaborations*. Addison Wesley Professional. 2002.

[Wirfs-Brock-Wilkerson-1989] Rebecca Wirfs-Brock and Brian Wilkerson. "Object-Oriented Design: A Responsibility-Driven Approach". OOPSLA '89 Proceedings. 71-75. October 1-6, 1989.

[Wirfs-Brock-Wilkerson-Wiener-1990] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software*. Prentice Hall. 1990.