Business Analysis using UML and URDAD

COLLABORATORS

	TITLE :	*REFERENCE* :
	Business Analysis using UML and URDAD	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Fritz Solms	June 9, 2010	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

List of Figures

List of Tables

Chapter 1

Introduction to business analysis

1.1 Introduction

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

1.2 Definition of business analysis

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

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

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

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

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

1.3 Sub-disciplines of business analysis

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

- **Enterprise analysis** Enterprise analysis 'focuses on understanding the needs of the business as a whole, its strategic direction, and identifying initiatives that will allow a business to meet those strategic goals.' This domain includes the maintenance of the vision and mission of the organization and the responsibility of ensuring that the organizational architecture is aligned with this vision and mission, i.e. that it provides a suitable infrastructure within which the vision and mission can be realized.

- **Requirements planning and management** Requirements planning and management ‘ involves planning the requirements development process, determining which requirements are the highest priority for implementation, and managing change.’
- **Requirements elicitation** Requirements elicitation ‘describes techniques for collecting requirements from stakeholders in a project.’
- **Requirements analysis and documentation** Requirements analysis and documentation ‘describes how to develop and specify requirements in enough detail to allow them to be successfully implemented by a project team.’ This domain includes business process design -- the technology neutral business process design provides a complete view onto the functional requirements.
- **Requirements communication** Requirements communication ‘describes techniques for ensuring that stakeholders have a shared understanding of the requirements and how they will be implemented.’

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

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

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

1.4 Responsibilities of business analysis

The responsibilities of the business analysis domain include

- **Assist in formulating a business case** Business analysis usually assists in formulating the business case for a new service or changes to an excisting service.
- **Elicit and documenting stake holder requirements** Business analysis needs to capture the detailed requirements around any service or use case the organization aims to provide to any of its users (e.g. clients, investors, employeees, ...). For each use case it needs to identify the stake holders which have an interest in that use case as well as the detailed requirements they have for that use case.
- **Design and document technology neutral business processes** Business analysis is responsible for designing business processes without concern about the implementation infrastructure and technologies. These business processes would then be mapped by development (including software development) onto the current choice of implementation technologies.

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

- **Assist strategic management to define the organization’s vision and mission** Having an understanding of the stake holder requirements for the organization, business analysis can assist strategic management in defining the vision and mission of the organization.
- **Design and document the architecture of the organization** Business analysis can assist organizational architecture to define a conceptual organizational architecture. This architecture is then mapped by management and the technical team onto an implementation architecture.

1.5 Model driven approach to business process design

In a model driven approach to business process design takes its basis from the *Model Driven Architecture* (MDA) published by the *Object Management Group* (OMG). The core aim of MDA is too separate the business process model from its implementation. The latter includes

- the deployment infrastructure (organizational and systems architecture),
- and the realization technologies.

The role of the architecture of the organization is to provide an infrastructure within which the organization can effectively realize its vision and mission. It must support the core capabilities through which the organization aims to differentiate itself from its competitors. Examples for such capabilities include

- *ingenuity*, the ability to generate innovative solutions,
- *reliability*, the ability to reliably meet stake holder expectations,
- *low cost*, the ability to compete price wise,
- *performance*, the ability to realize service requests fast,
- *flexibility*, the ability to customize the services/products to the customer's specific needs, and
- *scalability*, the ability to meet high demand.

Any business process deployed within this architecture should be realized according to the organization's qualities, i.e. if the architecture of the organization is designed to be able to meet high demand, then any business process deployed within this architecture will also be scalable.

The role of business process design is to design a business process which meets the functional / use case requirements around a business service. This is usually done iteratively designing and implementing one business service after another. If one follows a a model driven approach business process, the business process design results in a technology neutral business process model. In MDA this model is called the *Platform Independent Model* (PIM). This technology neutral business process design is then mapped onto a model which shows how the business process is realized across organizational and systems architecture. The result of this mapping is the *Platform Specific Model* (PSM). Ultimately the PSM is implemented resulting in an implemented business process which is ready for deployment. The artifacts (including the implemented software, trained staff, ...) is called the *Enterprise Deployment Model* in MDA.

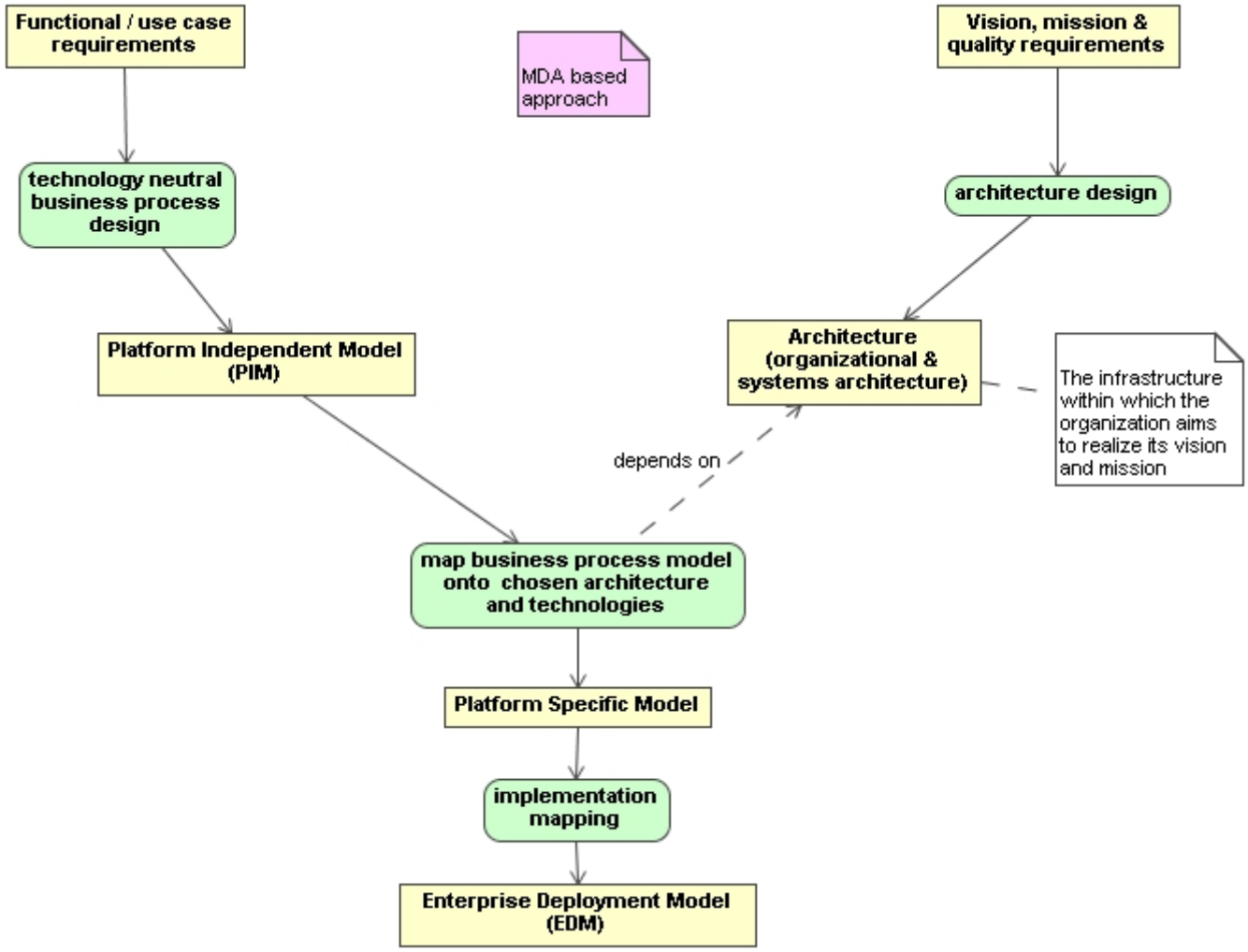

Figure 1.1:

1.5.1 Benefits of a model driven approach

The first core benefit of a model driven approach is an improved separation of concerns. The business process design can be done without knowledge of the implementation architecture. This is usually done iteratively designing and implementing a business process for each service iteratively.

The architecture is designed separately in order to provide a suitable infrastructure within which the organization can deploy its business processes. It must support the vision and mission of the organization.

The core advantage of a model driven approach is that the architectures and technologies can change, without affecting the technology neutral business process design. One will, however, have to redo the mapping of the business process onto the new architectures and technologies. MDA envisages that a large part of this mapping can be automated, reducing the cost associated with changes in the deployment architecture and technologies.

1.5.2 Role players and their responsibilities

Figure **??** shows the envisaged high level activities around organizational architecture and business process design. Below we discuss the role players and their responsibilities within such a model driven approach.

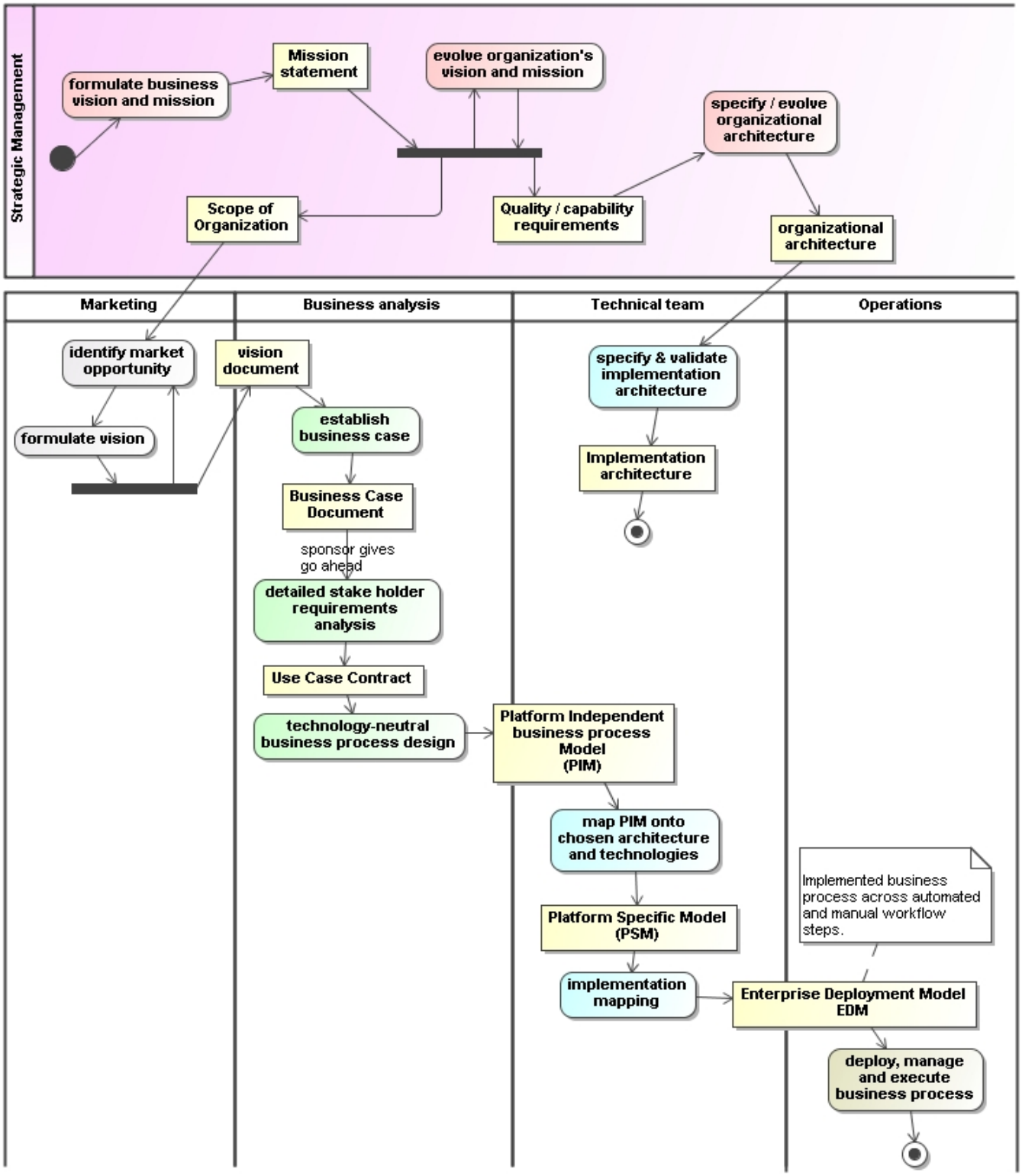

Figure 1.2: Model driven approach to business process design and organizational architecture

1.5.2.1 Strategic management

Strategic management is responsible for defining and evolving the organization's vision and mission. This includes a clear understanding of the scope of the organization's operations and the core capabilities or the organization, i.e. the desired quality of service.

This forms the basis on which strategic management, with assistance from business analysis, can select architectural strategies and define the core infrastructure of the organization. This may be done within the *Capability Driven Architecture* approach which provides a framework for defining an organization's architecture by suitably combining architectural patterns and strategies to realize the organization's capabilities.

1.5.2.2 Marketing

Marketing would typically identify market opportunities and specify a vision for products and or services which exploit these. In this context marketing plays the role of a client proxy which identifies and develops client needs.

However, not all services provided within an organization will necessarily be client services. Other stake holders like investors, regulatory organizations, the receiver of revenue, as well as of course the employees of the organization itself may require further services addressing their needs. Some may even have specific functional requirements around the client services themselves.

The requirements of these different stake holders would thus be provided by role players which interface with the relevant stake holders. For example, employee requirements as well as the legislative requirements around human resourcing would typically originate from human resourcing.

1.5.2.3 Business analysis

The role of business analysis includes requirements specification and business process design.

The business analyst has to understand in depth the requirements of the various stake holders around the services provided by the organization and to formalize these within a use case contract.

The stake holder service will be realized by a business process which fulfills the use case contract. The business analyst could use URDAD to design the business process. The resultant business process should be a *Platform Independent Model* (PIM). This model is independent of the implementation architecture and technologies. The same business process could thus be implemented as either a manual or an automated process. Furthermore, changes in realization technologies would not require changes to the PIM. Only the mapping of the business process onto its implementation will have to be modified.

1.5.2.4 The technical team

The technical team will concern itself on how the architecture and business processes are implemented within the organization.

On the one hand the technical team takes the organizational architecture specification obtained from strategic management and maps it onto an implementation which does effectively realize the organization's capabilities. The architecture will span across organizational and system components. The technical team will typically validate and document the proposed technical architecture. Furthermore, the architecture will be improved over time through better insight and improved technologies.

The second role of the technical team is to map the technology and architecture neutral business process design (the PIM) onto a *Platform Specific Model* (the PSM) which shows the realization of the various work flow steps within the chosen technologies. This may include decision on

- which work flow steps will be realized within the organization and which will be outsources
- whether certain work flow steps should be automated or not, and
- the technologies which should be used for the automated processes.

Finally the technical team will have to map the implementation model onto an actual implementation and specify how the latter will be deployed within the organization's architecture. The result of this is the *Enterprise Deployment Model* of the MDA.

1.5.2.5 Operations

Operations will have to deploy business processes into production, manage the business processes and ultimately execute them.

1.6 Challenges for technology neutral business process design

Technology neutral business process design is a non trivial exercise. Typical challenges faced include

- managing complexity,
- remaining technology neutral,
- the design should be a "good design",
- business process design requires input from knowledge from a wide range of business domains,
- the design must be understandable by the implementation team,
- business should be able to take ownership of its business processes,

1.6.1 Managing complexity

Business analysis should specify the full business process, not leaving any business decisions to developers. This can easily develop into a very complex business model, making it difficult to understand, to find relevant model elements

1.6.2 The design should be a "good design"

A good design should enforce

- enforce responsibility localization,
- support bidirectional traceability,
- enforce decoupling of service providers, and
- be testable and
- maintainable.
-
-
-

1.7 URDAD as a methodology for technology neutral business process design

Business process design is a non trivial exercise. Typical challenges faced include

- **Managing complexity**
- **Remaining technology neutral**
- **Ensuring responsibility neutrality**
- **Input from knowledge from a wide range of business domains**

Chapter 2

Introduction

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

2.1 What is UML

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

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

2.2 History of UML

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

Over time the semantic consistency of the UML was improved resulting in UML 2.0 which was released towards the end of 2005. UML also acquired support for specifying rules via the Object Constraint Language (OCL). The OCL contribution made by IBM enables UML to specify formal contracts for service providers.

2.3 UML does not provide a design methodology

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

2.4 Applicability of UML

UML is typically used to either

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

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

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

2.5 Relevance of UML for business analysts

Business analysts have used the UML for some time to specify the requirements specifications for systems. UML is also increasingly used to specify business processes and the structure of domain objects in a technology neutral way.

For example, one could specify the information which should be captured with a loan application in UML and then map this data structure onto paper based forms, web forms, XML data structures, system objects and onto one's choice of a database technology.

Similarly, one could use UML to document the business process for processing a home loan application and then map this business process onto either

- a manual process deployed within the organizational architecture or onto an
- automated process deployed onto one's current choice of systems architecture and technologies.

In future one may expect that the UML will also be increasingly used to specify the structure of the organization itself.

2.6 Overview of the UML diagrams

2.6.1 Introduction

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

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

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

2.6.2 Use case diagrams

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

2.6.3 Relevance of use case diagrams for business analysis

In the context of organizational modeling, business analysts would commonly use use case diagrams to

- document the services an organization offers to its clients,
- an overview of the stake holder requirements around these services,
- the scope of the organization's operations,

System requirements specification would preferably be preceded by business process design phase. One would project out the use case contracts for the system. In this context use case diagrams are used to document

- the user roles and the use cases which they will be using,
- an overview of the functional requirements for these use cases,
- the external systems the system may need to interface with as well as
- the scope of the system.

2.6.4 Sequence diagrams

Sequence diagrams were originally developed to provide an intuitive way to show the sequence of messages exchanged in a specific usage scenario. UML has extended the notation to support alternative flows, reuse of interaction patterns and concurrencies. Nevertheless, the core value of sequence diagrams lies in providing a very simple and intuitive way of showing an example of how objects collaborate to realize a specific usage scenario. Furthermore, they provide a natural notation for specifying time and duration constraints on processes.

Sequence diagrams ore commonly used to validate business or system process scenarios. They also show the information or objects provided when a service is requested from a service provider, as well as what service providers return upon completion of a service.

2.6.5 Applicability of sequence diagrams within business analysis

Sequence diagram are used by business analysis to

- document the messages exchanged between the participants of a business process,
- specify the required user work flow for a use case offered by a system.

In either case one can specify time related constraints including when certain actions should be taken as well as duration constraints on work flow steps.

Since sequence diagrams are so simple and intuitive, business analysts often use them to validate defined business process scenarios with business.

2.6.6 Activity diagrams

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

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

2.6.7 Activity diagrams in business analysis

Activity diagrams are the most useful notation for business process design. They document the activities performed by the various role players which collabrate to realize a client service. These role players could include

- organizational components,
- systems used by the organization, and
- external service providers to the organization.

Activity diagrams clearly show the decision points in a business process, show which work flow steps can be executed concurrently, when a business process needs to block until certain activities have been completed, the objects exchanged between the role players which collaborate to realize the business process

Often the choice of objects which collaborate to realize the client service is left to the implementation stage of the business process. In such a case the role players are abstract role players. When the business process is implemented one could assignthe abstract role to a system, a department or even an external service provider.

Note
Activity diagrams provide a more modern and more powerful alternative to flow charts and object flow diagrams.

2.6.8 Interaction overview diagrams

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

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

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

2.6.9 Interaction overview diagrams for business process design

Interaction overview diagrams can be useful for documenting a high level overview of a business process which focuses on the flow across the core interactions.

For example,the processing of a home loan may include

- an interaction with a client to capture the loan requirements,
- an interaction with credit agencies and internal departments to establish the credit worthiness of the applicant,
- an interaction with property agents to establish the value of the property and
- an interaction with legal services providers to generate the legal document for the bond.

The high level flow across these lower level interactions could be captured in an interaction overview diagram.

2.6.10 Class diagrams

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

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

2.6.11 Applicability of class diagrams within business analysis

Class diagrams are primarily used in business analysis for three things:

1. To specify the information structure of data objects which need to be captured, communicated and/or persisted.
2. To specify the core of services contracts for service providers required for a business process.
3. To document organizational structure.

2.6.11.1 Documenting the structure of information entities

A class diagram can be used to specify the information content of domain objects like an order in a technology neutral way. This information structure would be independent of whether the information is to be captured via a paper form, a web form or an XML data structure submitted via a web service. The data structure would also be independent of any database structure.

2.6.11.2 Class diagrams for SLAs

Class diagrams with interfaces and constraints can be used to specify the core of an SLA. The interface would specify the services which must be provided, the information exchanged, and potentially the pre- and post-conditions for each service.

2.6.11.3 Documenting the structural elements of an organization

Class and object diagrams can also be used to document the structural elements of an organization including

- the organizational components and the services they provide,
- the portals through which clients request services from the organization,
- the reporting channels within the organization, and
- other communication channels within the organization and those to external service providers and observers.

2.6.12 Object diagrams

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

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

2.6.13 State charts

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

UML state charts extend Harel state charts by adding

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

2.6.14 State charts within business analysis

Even though state charts could be used to trace the state of domain objects through a business process, they are used only very rarely by business analysts.

2.6.15 Timing diagrams

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

2.6.16 Timing diagrams and business analysis

Timing diagrams are used in business analysis when one would like to show, within a single diagram, both

- the messages exchanged between the role players participating in a process, as well as
- the activities they perform at various stages in the business process.

Amongst other things it can be used to identify when resources are used and when they are idle.

2.6.17 Communication diagrams

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

2.6.18 Applicability of communication diagrams within business analysis

Communication diagrams are used relatively infrequently by business analysts. They are typically not used when documenting the requirements specification for a system. However, they do prove, at times, useful for documenting the messages exchanged between the participants of a business process, clearly showing the message paths required between these participants.

2.6.19 Component diagrams

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

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

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

2.6.20 Component diagrams for organizational modeling

Component diagrams can be used to specify the replacable internal or external service providers used by an organization. The service providers can be internal departments, external service providers or systems used by the organization.

The diagram exposes the services contracts the currently selected service providers realize. As service provider can be replaced with another realizing the same contract without the organization having to modify its business processes.

For example, an organization which currently handles its own deliveries may decide to outsource this responsibility to an external service provider. Changing from an internal deliveries department to an external courier should not affect the remainder of the business process.

Similarly, a plumber who changes from a manual invoicing to an invoicing system should not have to change the higher level business processes realizing the plumbing services.

2.6.21 Composite structure diagrams

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

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

2.6.22 Using composite structure diagrams to document the structure of an organization

Composite structure diagrams prove very useful to document the structure of an organization. The diagrams will show

- the core components of the organization,
- the connectors or communication links between the organizational components, and
- the portals through which clients can request services from the organization.

Documenting the strucuture of the entire organization would, of course, be impractical. Instead one would typically use componsite structure diagrams to only show those aspects of the structure which are

- relevant for the collaboration realizing some client service, and
- at a similar level of granularity.

2.6.23 Package diagrams

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

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

Package diagrams can be used to show

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

2.6.24 Deployment diagrams

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

2.6.25 Deployment diagrams for organizational modeling

A deployment diagram for an organization would show

- the nodes onto which organizational components are deployed,
- optionally specify the contract which each component realizes,
- the external service providers the organization makes use of, and

- the communication channels established between these components.

For example, one could use a deployment diagram to show that some bank

- has a head office which is deployed in a particular building in the city centre,
- has a retail outlet which is deployed in some shopping centre (a node which hosts retail outlets),
- uses a specific legal services provider for drafting bond agreements and conveying properties, and that this legal services provider needs to realize the Service Level Aggreement (SLA) or contract which the bank establishes with conveyancers,
- uses a particular courier service as communication link to and from the conveyancer.

2.7 The UML model

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

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

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

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

2.8 UML and MDA

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

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

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

Chapter 3

Use-case diagrams

3.1 Introduction

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

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

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

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

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

3.2 The scope of an organization

The high level scope of an organization's operations is usually defined by its mission statement. For example, the scope of a landscaping services provider may be to design, establish and maintain landscaped gardens. This high level scope of the organization could be documented using a use case diagram.

3.3 Stake holders of an organization

The stake holders of an organization typically include

- *clients* who extract value from the organization's services or products,
- *share holders* who receive a return on their investment,
- *employees* who receive financial and other return on their time investment,
- *other value extractors* like the receiver of revenue, society at large, ...

3.4 Simple use case diagrams

3.4.1 Introduction

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

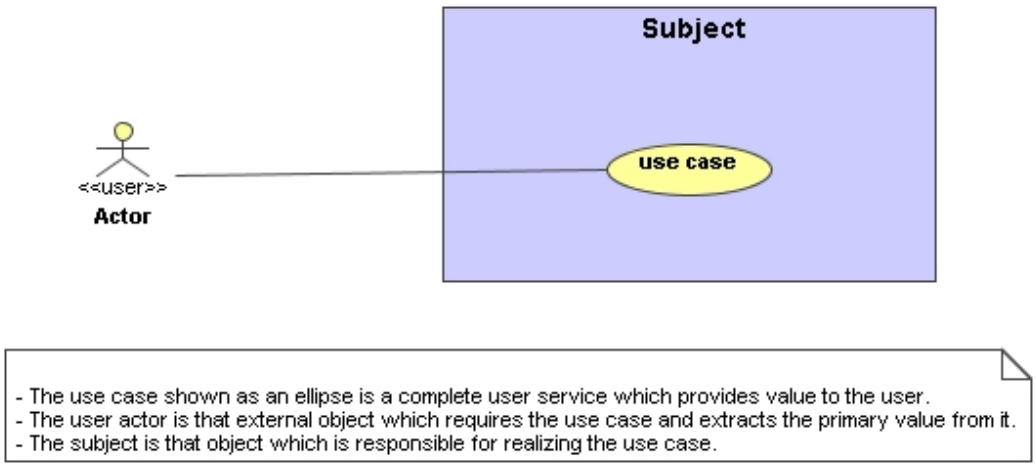

Figure 3.1: Simple use case diagrams

3.4.2 Use cases

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

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

- Display time of day.
- Display current date.
- Measure elapsed time.

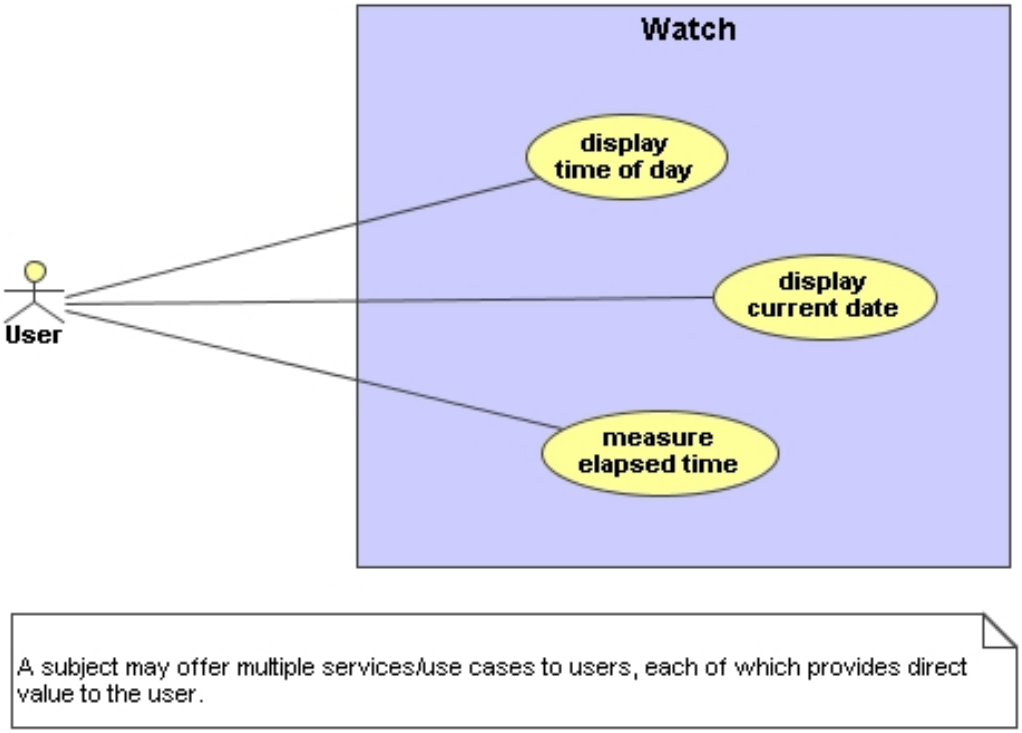

Figure 3.2: A user can access multiple use cases

3.4.3 The subject

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

The subject could be

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

3.4.4 Communication links

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

3.4.5 User roles

A user role is a role played by an external object which makes use of one or more services/use cases offered by the subject. Different user roles may have access to different use cases.

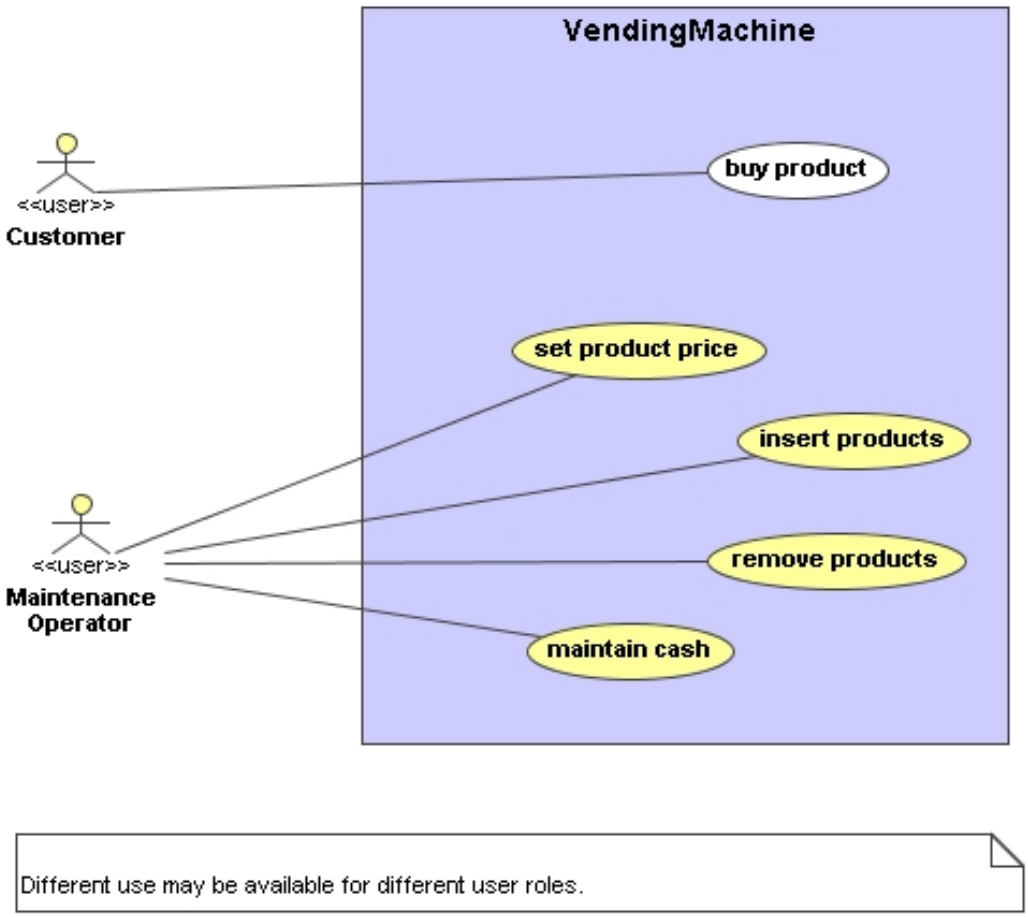

Figure 3.3: A vending machine provides different services to customers and maintenance operators

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

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

3.5 Stereotypes in use case diagrams

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

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

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

A stereotype is shown in UML using french quotation marks or guillemets, `<< ... >>`. Alternatively or additionally a stereotype icon may be assigned to the stereotype. The standard stereotype icon for the `<<control>>` stereotype.

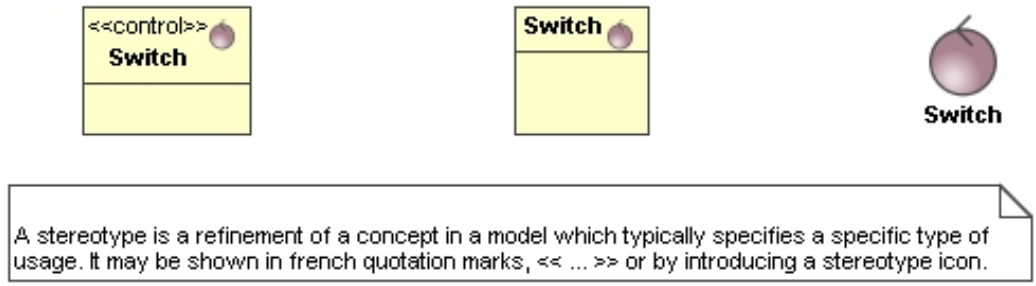

Figure 3.4: A switch which has been stereotyped as a control object

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

3.6 A stereotype for an agency

One could use a stereotype to introduce the concept of an agency to a model. This stereotype could then be applied to a range of classes which are used as agencies.

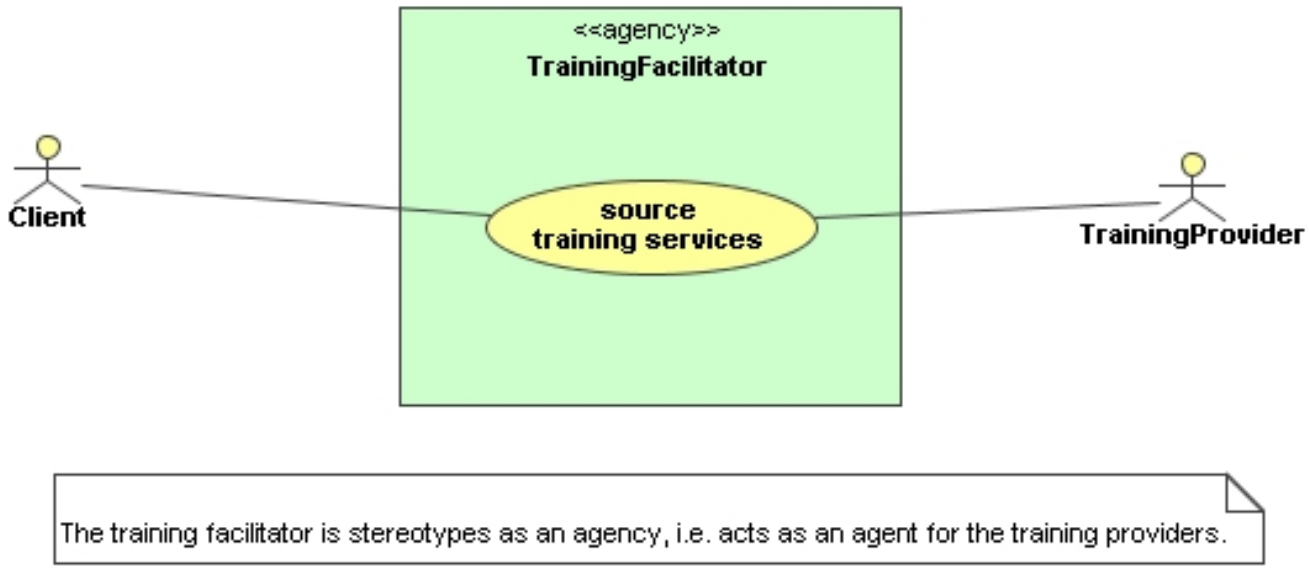

Figure 3.5: A training facilitator is used as an agency for training providers

3.7 Actors

3.7.1 Introduction

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

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

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

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

3.7.2 Actor types

One can distinguish between three core types of actors:

- **Users** The user is the role which makes use of a use case or service offered by the subject. It extracts the primary value from the use case, typically initiates the use case and, at times, may drive it.
- **Service providers** Service providers provide services to the subject, assisting it to realize its services or use cases to its users (e.g. clients or system users). They do not make use of the use case. The services provided by these service providers usually fall outside the scope of the subject.

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

- **Observers** An observer of a use case does not make use of the use case. Nor does it provide lower level services to the subject to assist it to realize the use case for its users. Instead, an observer receives certain deliverables from the subject in the context of it realizing the use case for its user. While users extract the core value from the use case, observers extract auxillary value from it.

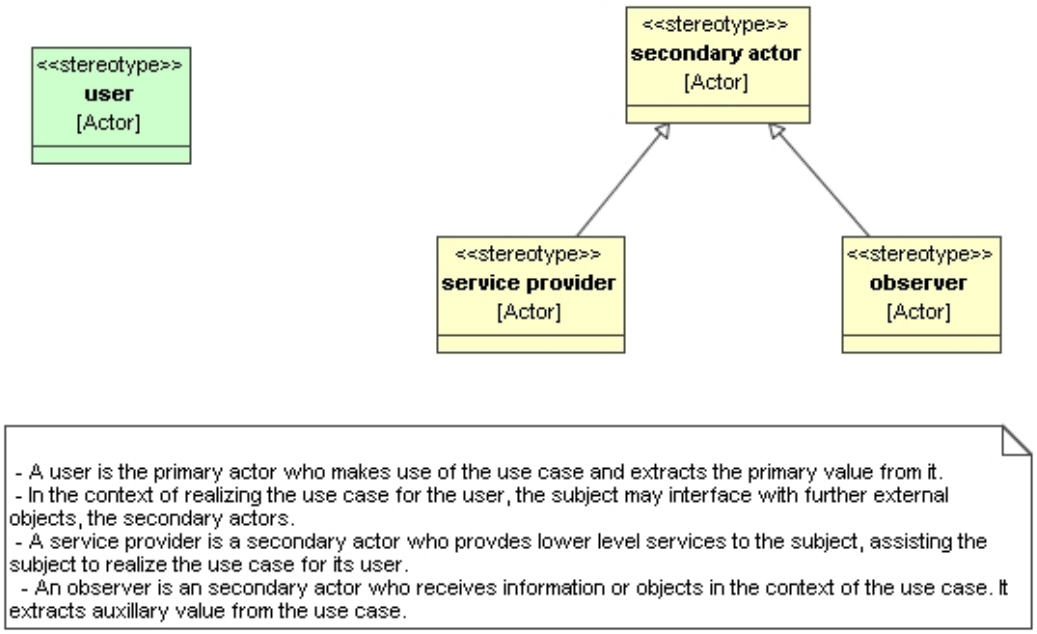

Figure 3.6: Users, service providers and observers as the core actor types

Either of these roles, user, service provider or observer, can be played by external organizations, systems or people.

3.8 Typical actors of an organization

Typical actors of an organization include clients, service providers to the organization, investors, and other external entities which extract value from the organization like tax collection agencies.

3.9 Example of actor types in organizational modeling

The standard actor types, users, service providers and observers are just as relevant in organizational modeling. Consider, for example, the present course service offered by a training institution shown in Figure **??**

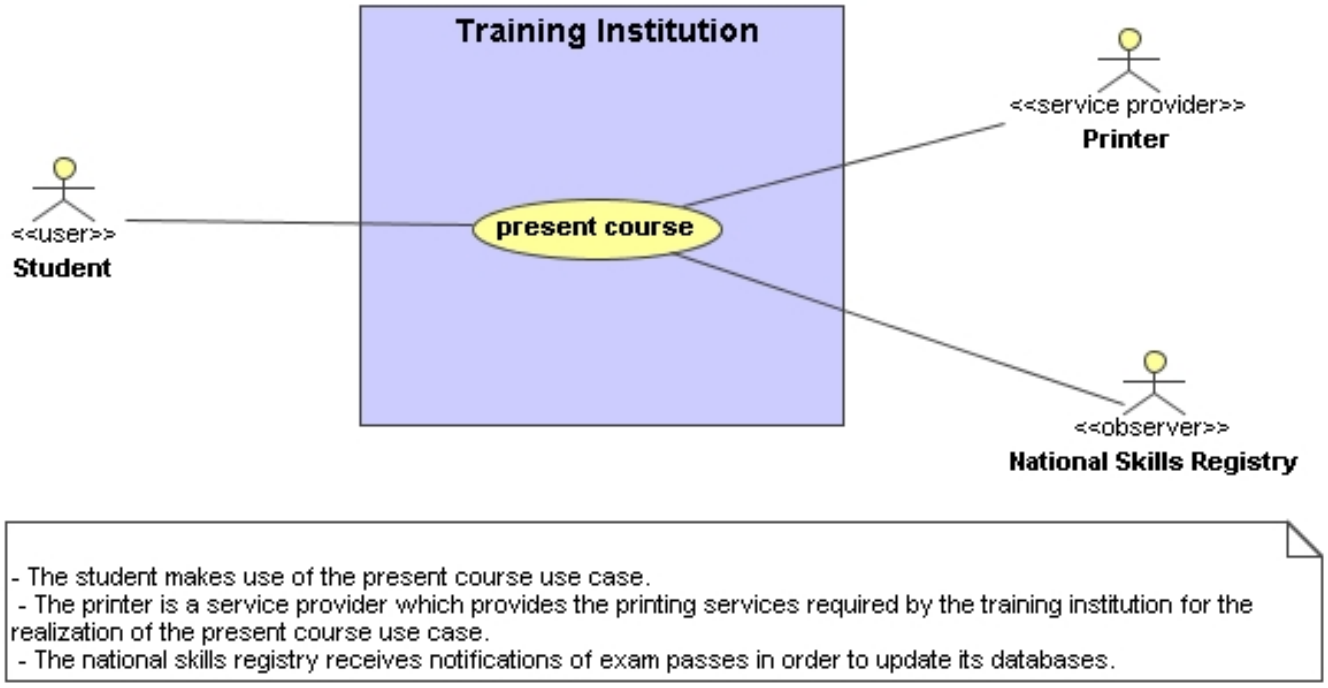

Figure 3.7: Users, service providers and observers of a training institution

The user of the *present course* use case is a student, i.e. some or other object which, in the context of this use case, playes the role of a student. The student obtains the primary value from the use case.

The printing of course material is outside the scope of the organization. The training institution out-sources this responsibility to external organizations which play the role of a printer. The printer provides a lower level service, that of printing course notes, to the training institution, assisting the training institution to realize its higher level use case, that of presenting a course, for its students. The printer plays the role of a service provider to the training institution.

When a student passes a course, the national skills registry is informed. This enables the national skills registry to maintain the information about the skills available within the country. The national skills registry thus obtains some auxillary value from the training institution presenting the course to its students, without either making use of the present course use case and without providing any services to the training institution. It plays the role of an observer.

3.10 Functional requirements

3.10.1 Introduction

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

3.10.2 Mandatory functional requirements via the include relationship

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

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

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

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

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

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

3.10.3 Conditional functional requirements via the extend relationship

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

Consider, for example, the following functional requirements around the *withdraw cash* use case:

1. If the client requests to select the types of notes, provide him the facility to do so.
2. Print a transaction confirmation if the printer is working.

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

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

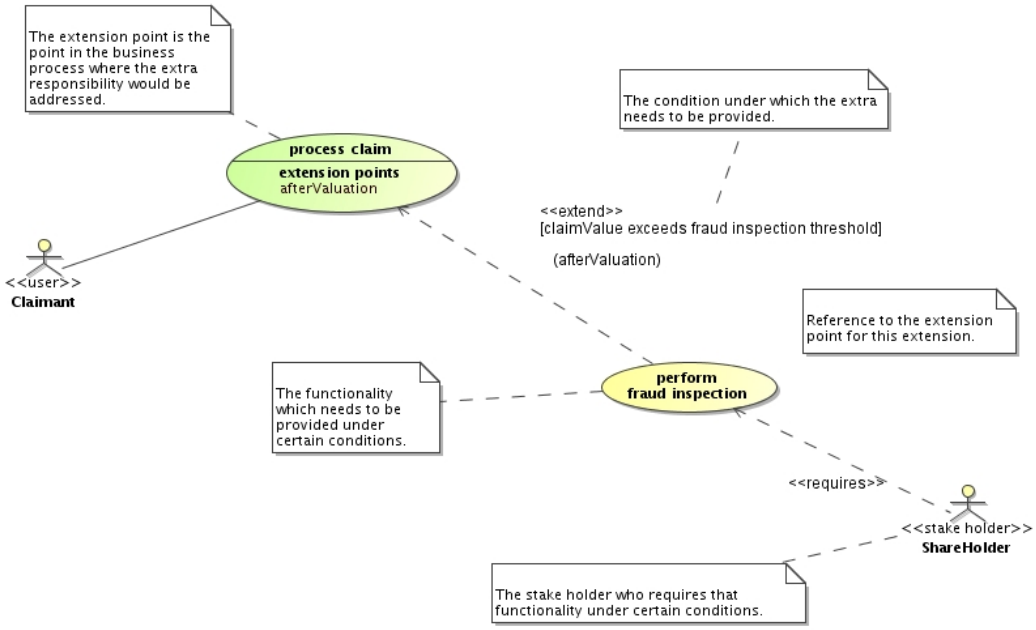

Figure 3.8: Extend relationships for conditional functional requirements

3.10.3.1 Specifying the conditional

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

3.10.3.2 Specifying extension points

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

1. **User confirmation** The point in the work flow where the user confirms the transaction
2. **Bank confirmation** The point in the work flow where the bank confirms the transaction.

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

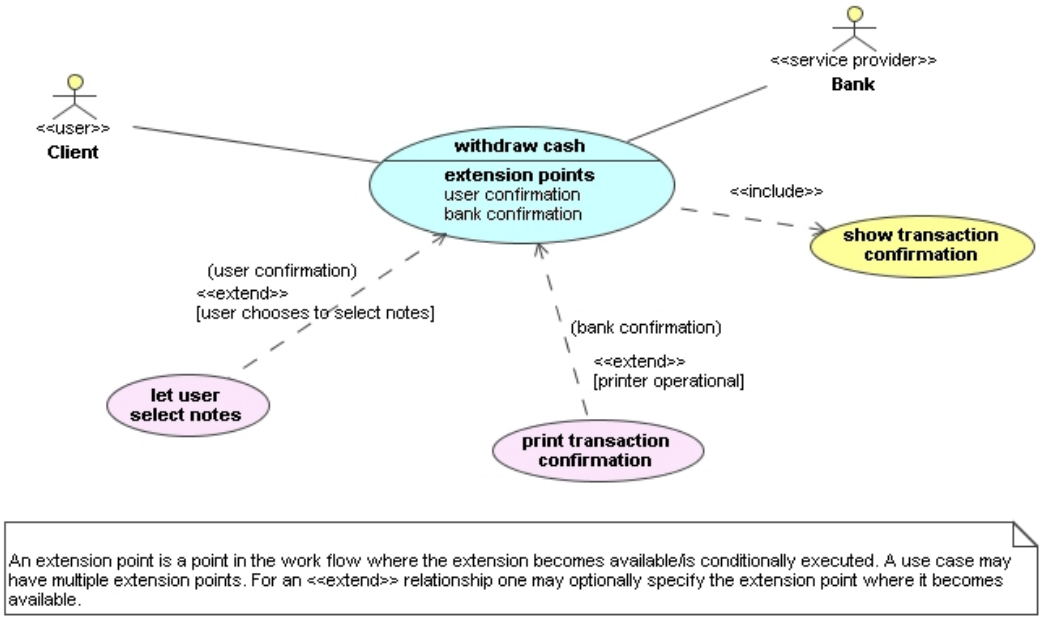

Figure 3.9: Extension points

3.10.4 Identifying functional requirements

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

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

3.11 Mandatory functional requirements for a business process

An example of a mandatory functional requirement for a business process is that of having to verify the policy status when processing the claim. The policy holder should not be able to receive the claim value if the policy status has not been verified.

When specifying mandatory functional requirements for a use case, it is advisable to specify the stake holder who requires the functionality. This can be done using a `<<requires>>` dependency pointing from the stake holder to the functional requirement.

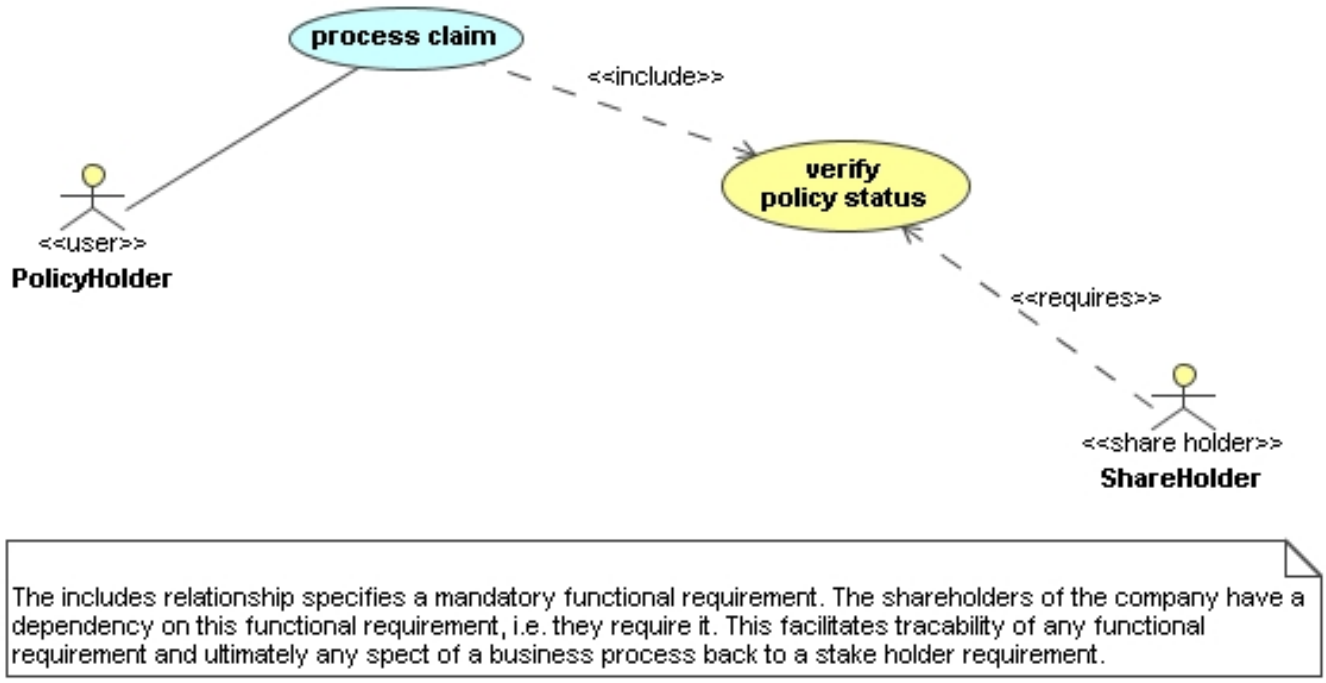

Figure 3.10: Mandatory functional requirement for a use case and the stake holder who requires it

3.12 Conditional functional requirements for a business process

An example of a conditional functional requirement for a business process is that of having to perform a fraud investigation for all insurance claims exceeding some specified fraud investigation threshold. This would typically be required by the share holders of the insurance company.

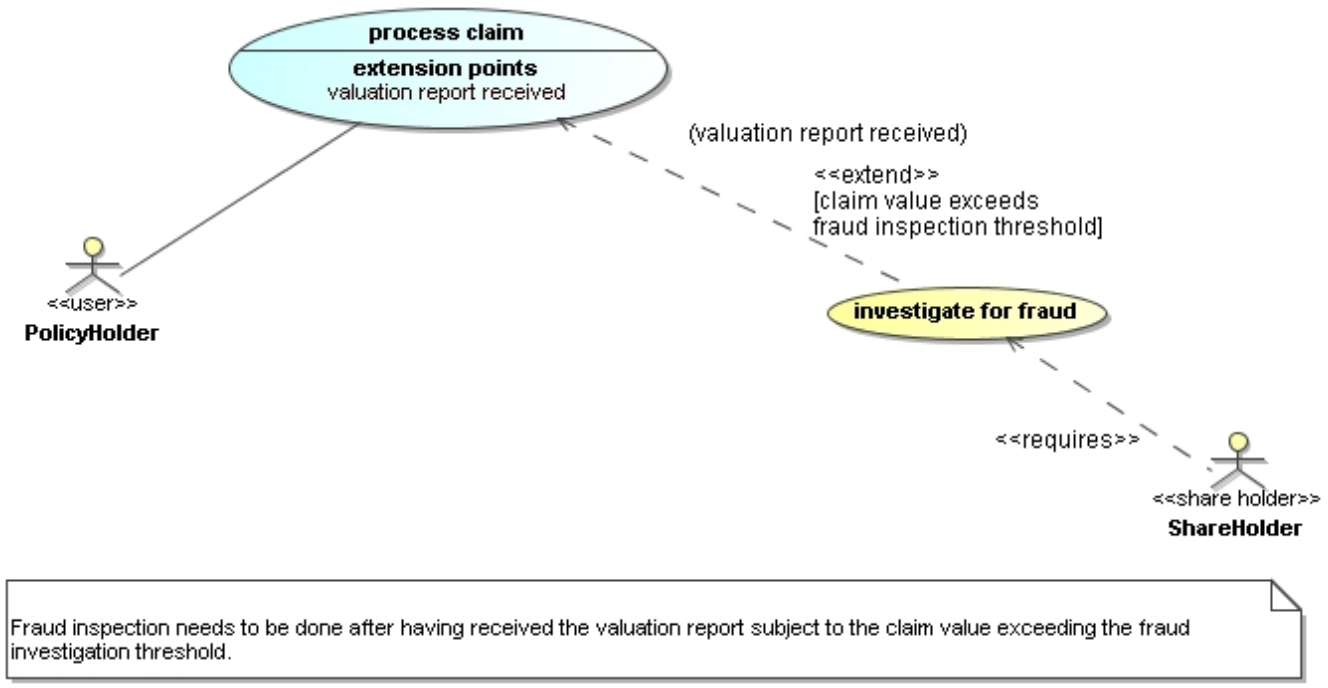

Figure 3.11: Conditional functional requirement for a use case and the stake holder who requires it

3.13 Use-case abstraction

3.13.1 Introduction

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

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

3.13.2 Scoping

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

To specify a use case abstraction or generalization relationship in UML, one draws a solid line with a triangular head (a generalization relationship) pointing from the more concrete (specialized) use case to the more abstract or generic use case.

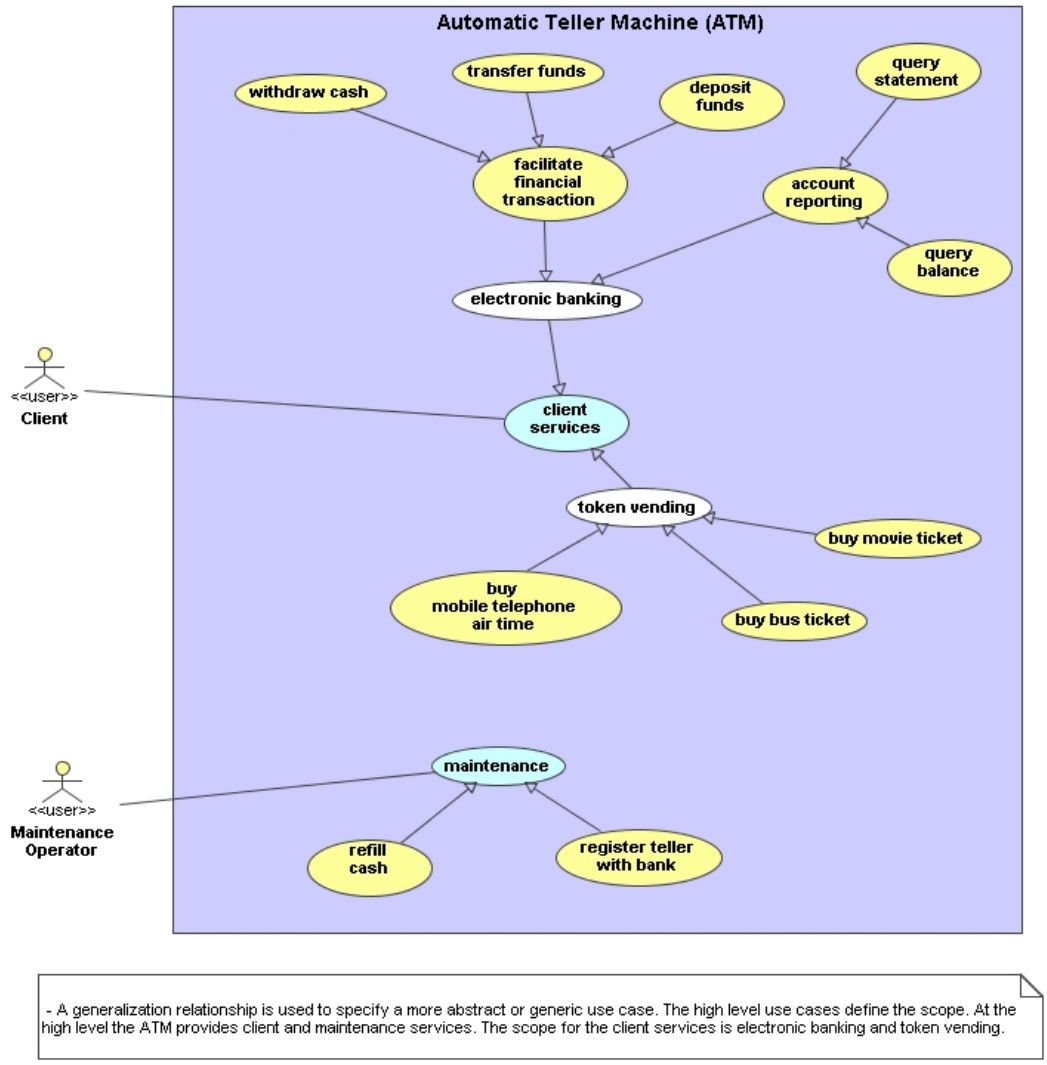

Figure 3.12: Using use case abstraction to define the scope of an ATM

Figure **??** shows that the high level use cases are the client and maintenance services. The scope of the client services is defined by the next level use cases, *electronic banking* and *token vending*.

A request for a new client service which is not a specialization of either electronic banking or token vending would be declared out of scope (alternatively one could revisit the scope of the ATM). An example would be that of vending a product (like chocolates). Such a service could not be seen as a specialization of either of the use cases defining the client scope and hence would be out of scope.

3.13.2.1 Concrete use cases

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

3.13.3 Defining common requirements and common actors across use cases

Use case generalization or abstraction enables one to specify functional requirements and actors at various levels of abstraction. For example, in Figure **??** the mandatory functional requirement of verifying the card and pin with the bank applies to all client

services including the electronic banking services (e.g. withdraw cash, ...) and the token vending services (e.g. buy bus ticket, ...).

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

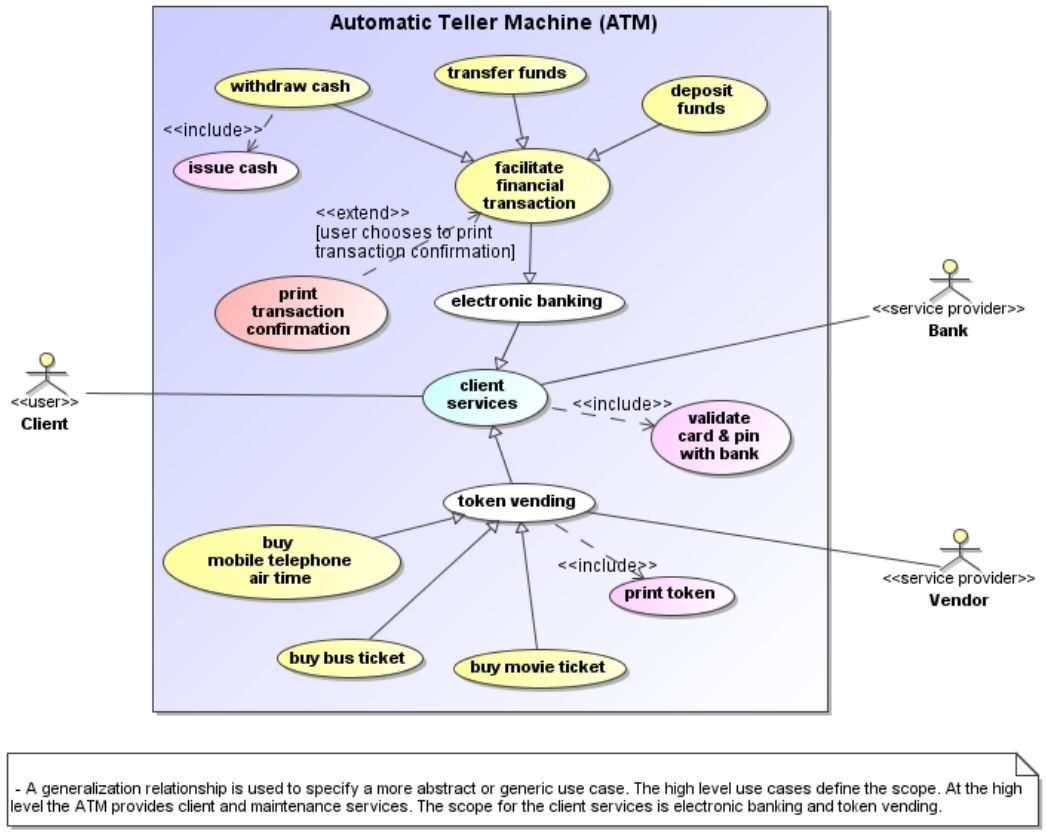

Figure 3.13: Using use case abstraction to specify commonalities across use cases

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

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

3.14 Defining the scope of an organization

Use case abstraction can be used to specify the scope of an organization's services. For example, One may specify that the scope of the organization's client services includes training and consulting services.

The consulting services cover business and technical consulting. The business consulting includes consulting services around business process design and organizational architecture.

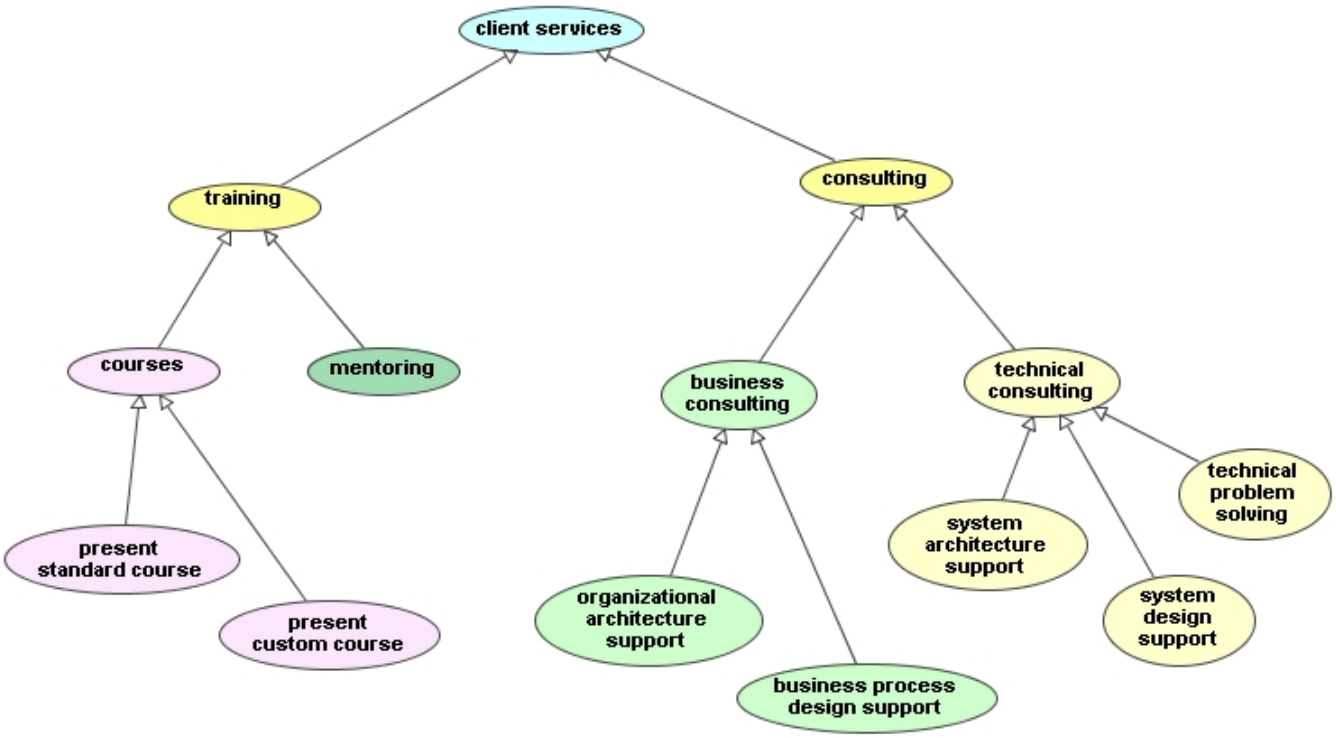

Figure 3.14: Defining the scope of the operations of some training and consulting company

3.15 Actor abstraction

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

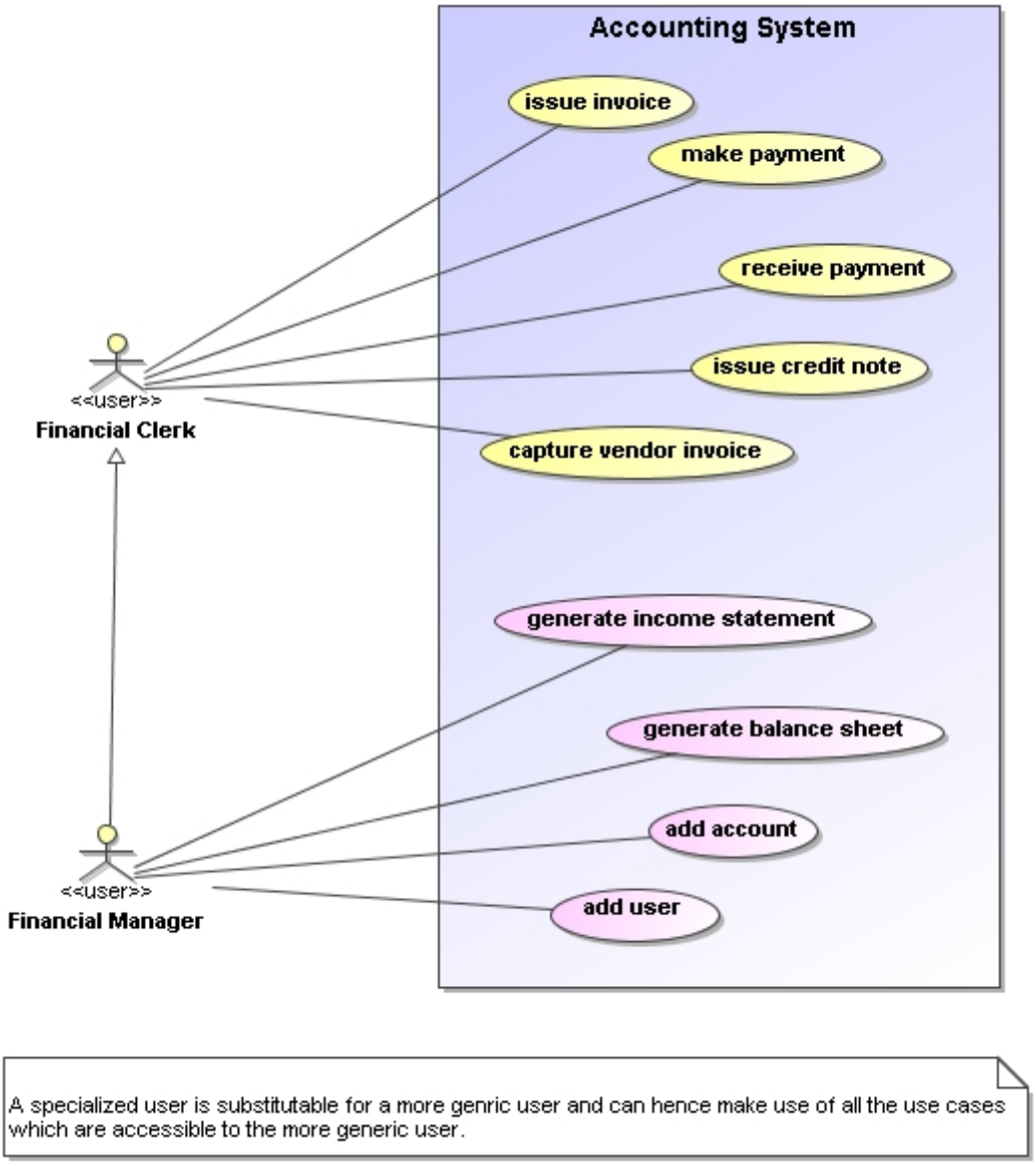

Figure 3.15: Specialized users have access to the services the more abstract/generic users can access

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

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

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

3.16 Summary of the UML notation for use case diagrams

Figure **??** summarizes the UML notation for use case diagrams.

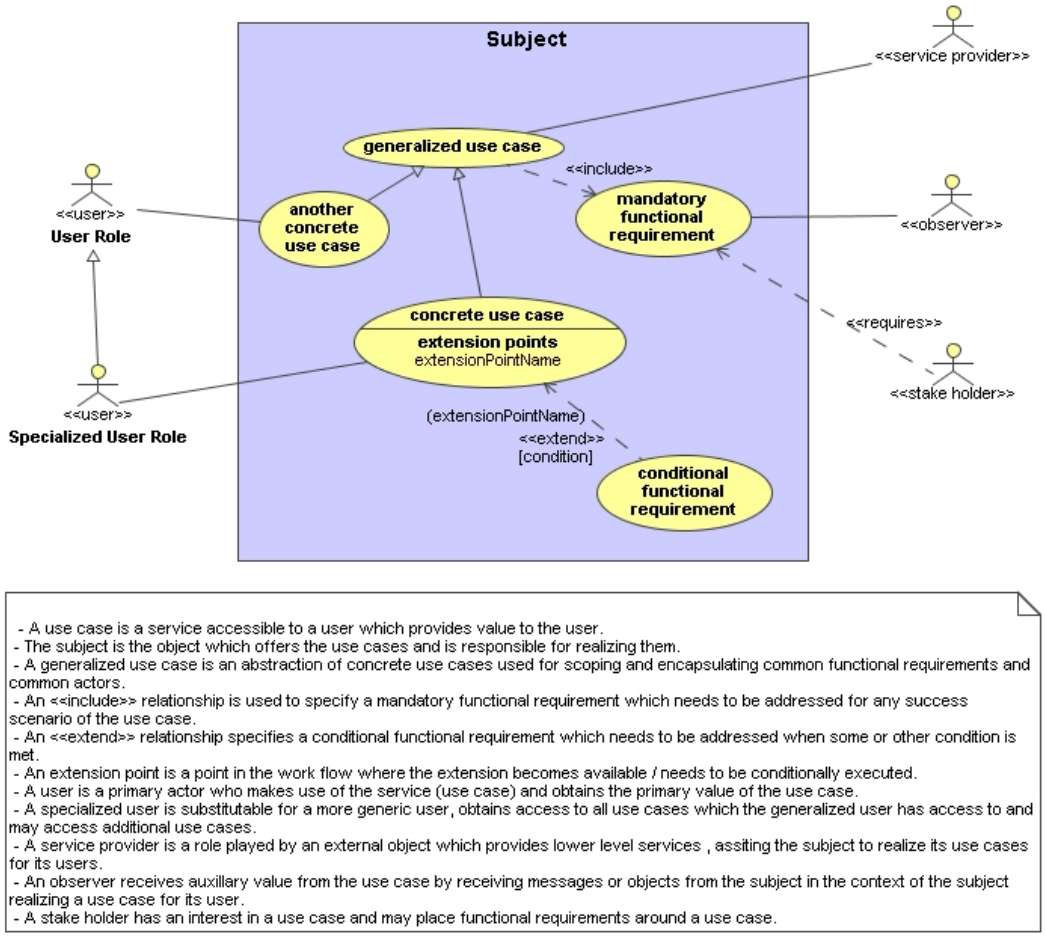

Figure 3.16: Use case diagram

Chapter 4

Class diagrams

4.1 Introduction

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

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

In particular, class diagrams can be used to specify

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

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

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

4.1.1 Objects and Classes

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

4.1.1.1 Objects

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

4.1.1.1.1 Common examples of objects

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

4.1.1.1.2 Identifying objects

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

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

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

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

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

Table 4.1: Objects can be identified from nouns.

4.1.1.2 Classes

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

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

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

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

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

4.1.1.2.1 Identifying classes

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

> *We issue invoices for purchases.*

or

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

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

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

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

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

Client	Account	Currency	Transaction	Date	Period	Statement

Table 4.2: Classes as abstraction of specific objects.

4.2 Basic object and class diagrams

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

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

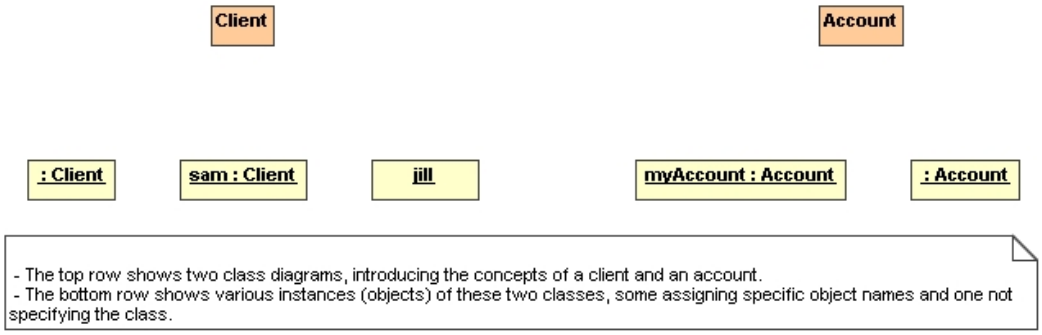

Figure 4.1: Simple UML class and object diagrams

In Figure **??** the top row (`Client` and `Account`) represents classes, while the bottom row specifies instances of these classes. We are specifying that `sam` is an instance of the *Client* class.

Object diagrams are commonly used to document example scenarios of a state of a business process execution. In either case one often omits the object name. One thus simply specifies that one is working with some instance of a class without giving that instance a name. In such cases one still uses the instance of operator (the colon) to specify an instance of a client (`:Client`) or an account (`:Account`).

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

4.3 Attributes

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

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

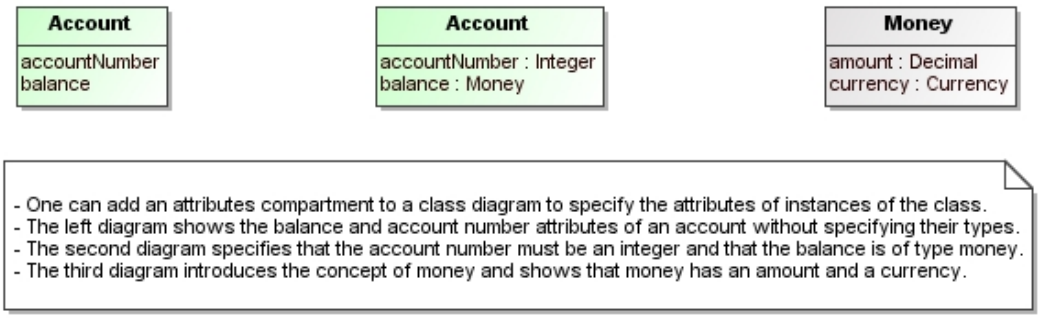

Figure 4.2: Adding an attributes compartment to a class.

4.3.1 Collection attributes and multiplicity constraints

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

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

specifies that the `telephoneNumbers` attribute is a collection of 3 telephone numbers. Similarly

```
telephoneNumbers:TelephoneNumber[1..3]
```

specifies that between 1 and 3 telephone numbers are required.

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

```
claimItems:ClaimItem[1..*]
```

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

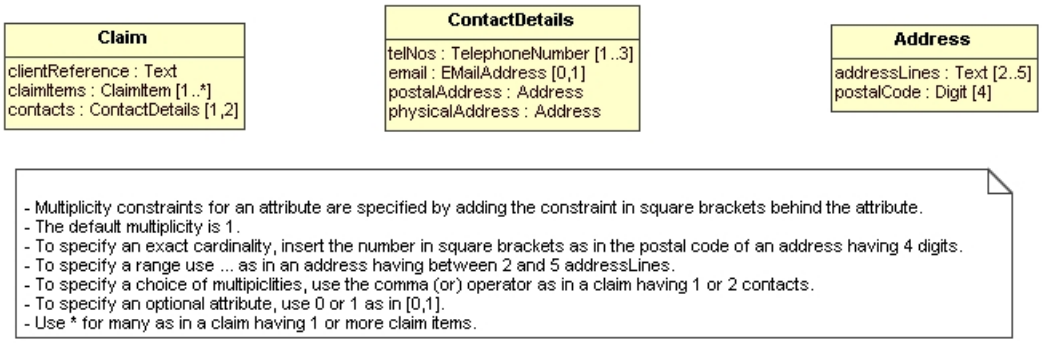

Figure 4.3: Multiplicity constraints on attributes

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

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

4.3.2 Derived attributes

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

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

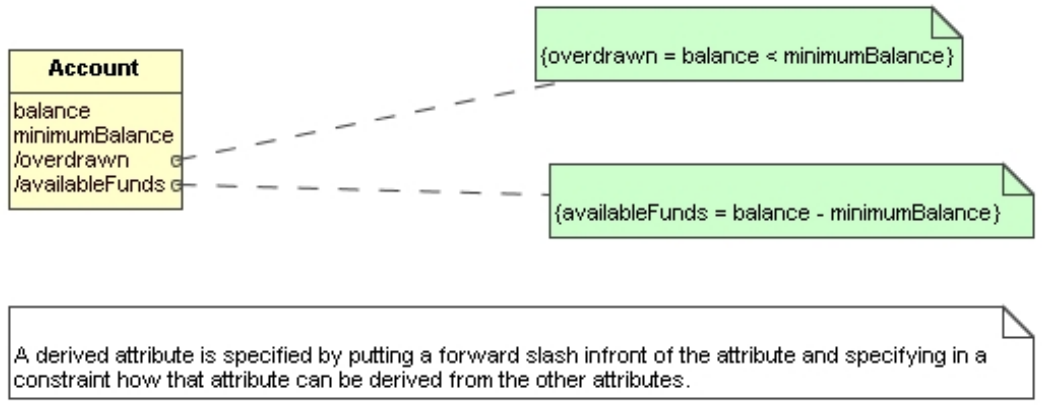

Figure 4.4: Derived attribute and the constraints specifying how their values are determined from the current value of the other attributes.

4.4 Services

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

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

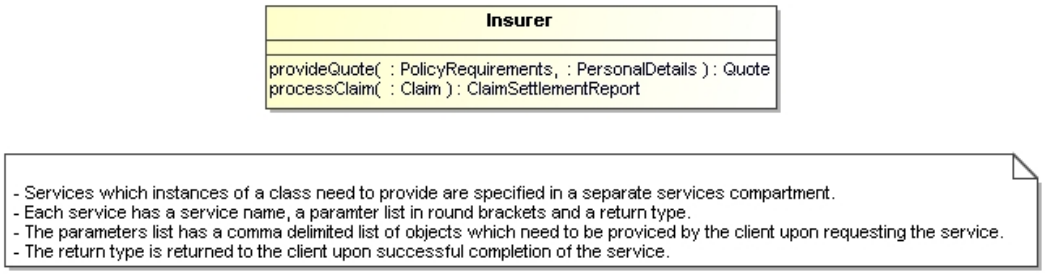

Figure 4.5: Services are shown in a separate services compartment.

4.4.1 Service inputs

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

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

The information conveyed with these objects is specified in separate class diagrams. We would thus add a class diagram for the `PersonalDetails` class specifying in detail the type of personal information the insurer requires as well as class diagram specifying the information which would be provided as the `PolicyRequirements`.

Note
For each input and return type one would introduce a class diagram to specify the information which is conveyed. For example, the class diagram for a Quote would specify the information which would be contained in the quote.

4.4.1.1 Assigning a business process to a service

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

4.4.1.2 Assigning role names to the input objects

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

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

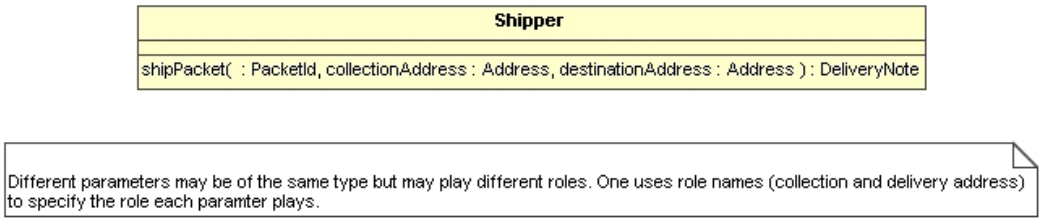

Figure 4.6: Specifying the role names for the different input objects.

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

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

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

At times service providers are willing and able to deliver a service without the user/client having to provide any other information than to request the service itself. In such cases the input objects list is left empty.

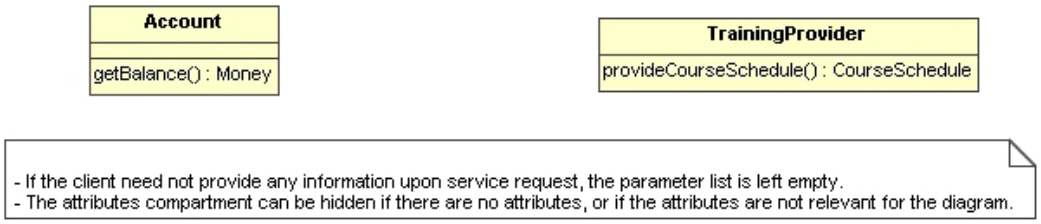

Figure 4.7: A service without any input objects.

Figure **??** specifies that training institutions can provide the current course schedule without requiring any further information from the client requesting the course schedule.

4.4.2 Return types

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

4.4.2.1 What if multiple deliverables?

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

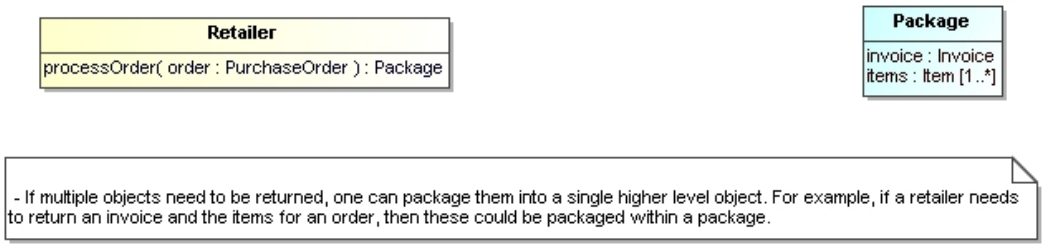

Figure 4.8: Compound return values

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

4.4.2.2 What if nothing is returned?

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

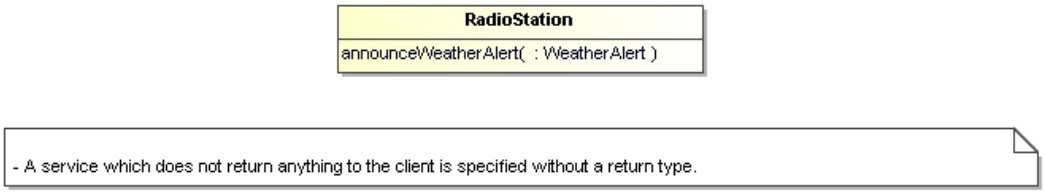

Figure 4.9: Services need not have a return type.

4.4.3 Input, output and input/output parameters

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

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

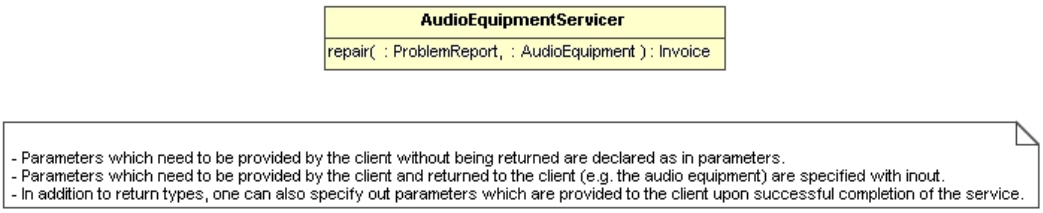

Figure 4.10: Parameters may be specified as in, out and inout.

In this case the full service signature would be

```
repair(in problemReport:ProblemReport, inout item:AudioEquipment) :Invoice
```

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

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

4.4.4 Multiple services with same service name (overloading)

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

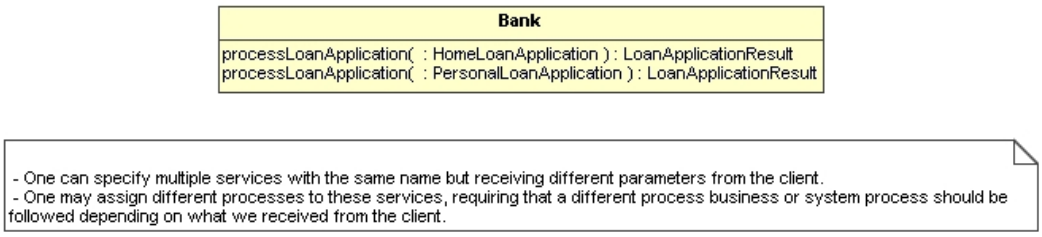

Figure 4.11: Different business processes may be followed for different input parameters.

4.5 Access levels (visibility)

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

UML defines 4 access levels:

- **public (+)** The member can be accessed from any object which has a handle/message path to the container. For example, a client of an organization can use the public services of the organization. Similarly any system object which has a message path to another system object can access that objects public methods and attributes.
- **package (~)** The member can be accessed from any object which is in the same package as the container.
- **protected (#)** The member can be accessed from within the container itself or from within instances of specializations of the container. For example, protected members of a class can be accessed from within sub-class instances.
- **private (-)** The member can only be accessed from within the container itself. For example, the private attributes of an instance of a class can only be accessed from within any of the business or system processes of that same instance.

4.6 Public and private services of an organization

While organizations usually provide a range of services to its clients, they also usually implement a number of services which are available to members of the organization, but not to the public. The former would be public services, while the latter would be private services.

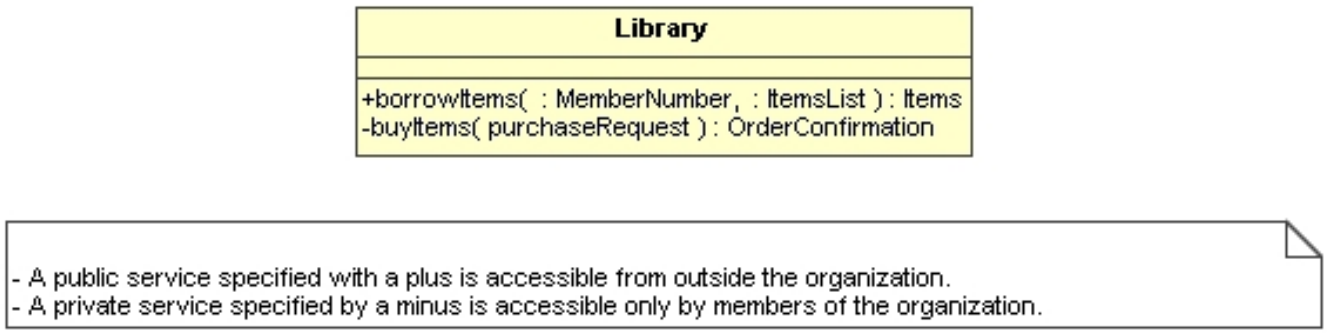

Figure 4.12: Public and private services of an organization

Figure **??** shows a public and a private service of a library. Clients of the library may borrow library items, but only members of the library may request the purchasing of new items.

4.7 The camel naming convention

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

The convention is very simple:

- Class (and interface) names start with capital letter.
- Everything else, including services and object names, starts with a lower case letter.
- Word boundaries are capitalized in either case.

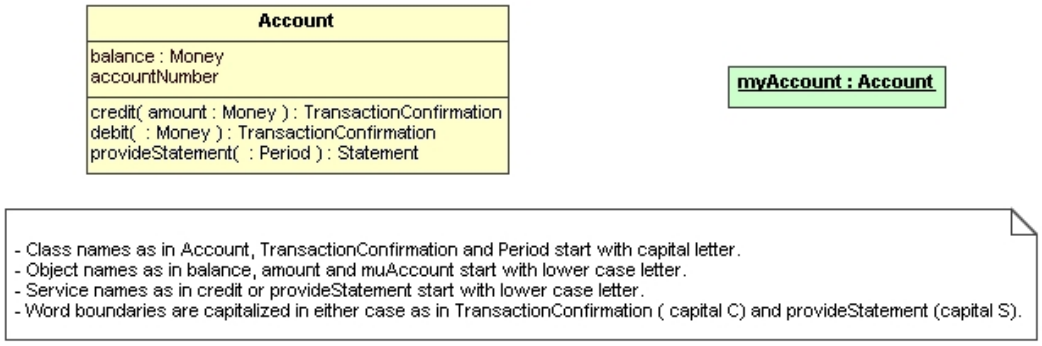

Figure 4.13: The camel naming convention.

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

- Class names like *Account*, *Value* and *TransactionConfirmation* start with capital letter, with word boundaries capitalized as is done using a capital *C* for *TransactionConfirmation*.
- Object names like *myAccount*, the parameters *amount*, *balance* as well as the class attributes or properties like *accountNumber* all start with lower case letter with word boundaries still capitalized.
- Service names like *credit* and *generateStatement* start with lower case letter.

4.8 Interfaces

4.8.1 Introduction

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

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

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

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

4.8.2 UML notation for interfaces

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

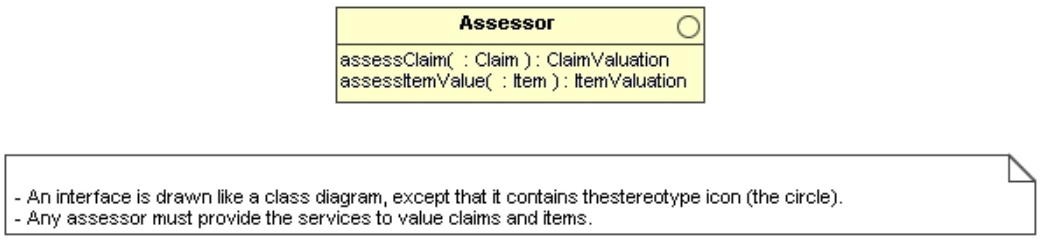

Figure 4.14: An interface specifying the services required from an assessor.

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

4.8.3 Realizing/implementing interfaces

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

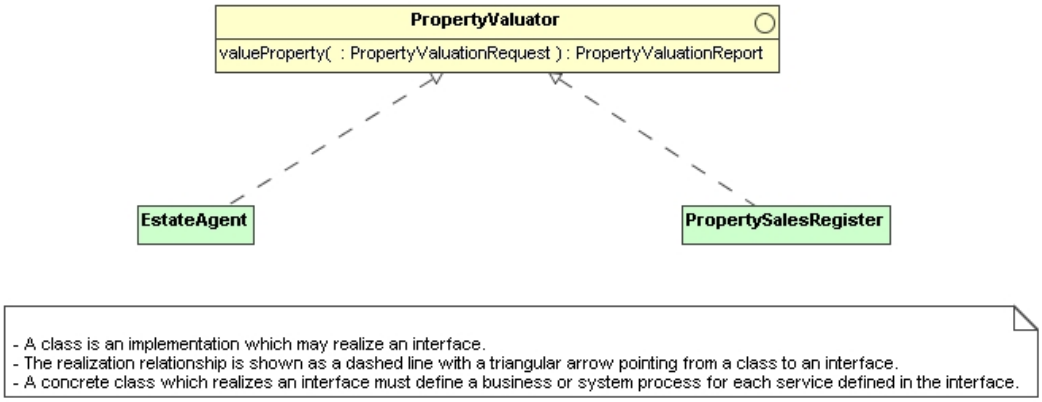

Figure 4.15: Estate agents and the property sales register both realize the services required from a property valuator.

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

4.8.4 Decoupling from service providers

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

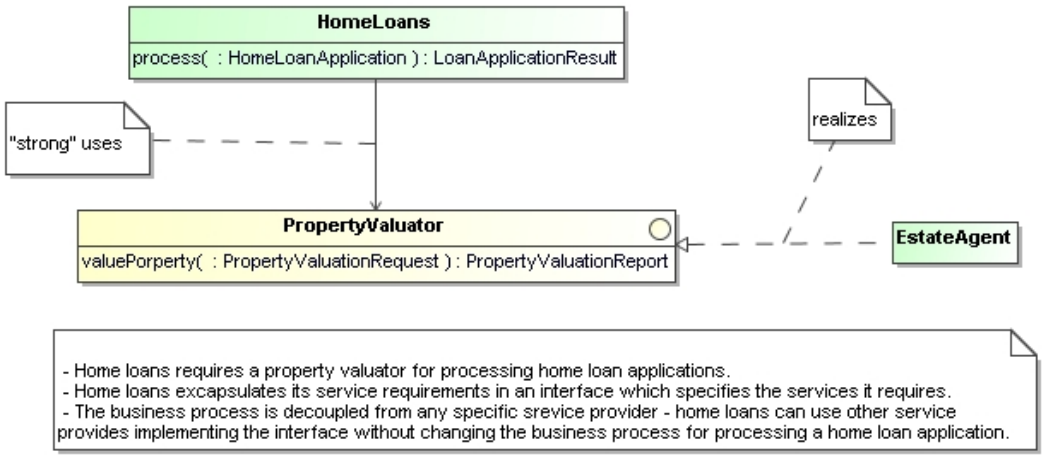

Figure 4.16: Home loans is decoupled from any concrete realization of a property valuator.

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

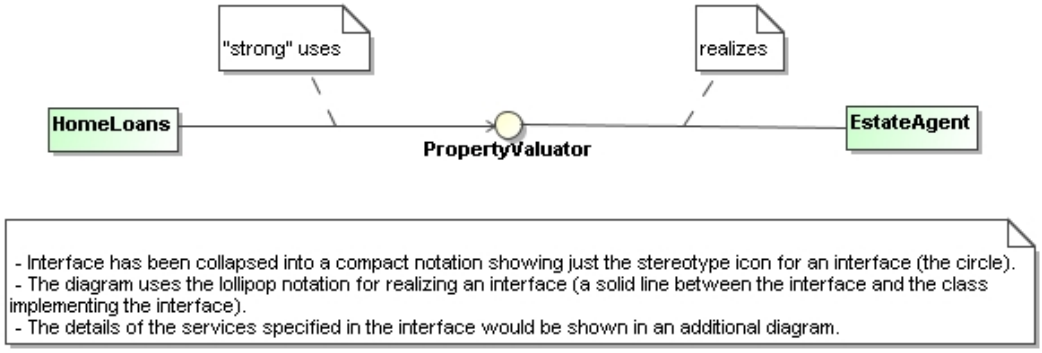

Figure 4.17: The property valuator interface collapsed into its stereotype icon.

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

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

4.8.4.1 Required and provided interfaces

UML supports a second notation highlighting an interface required by a client and the interface provided by a service provider. The notation uses the dish to show the required side of an interface and the lollipop notation to show the provided side of an interface.

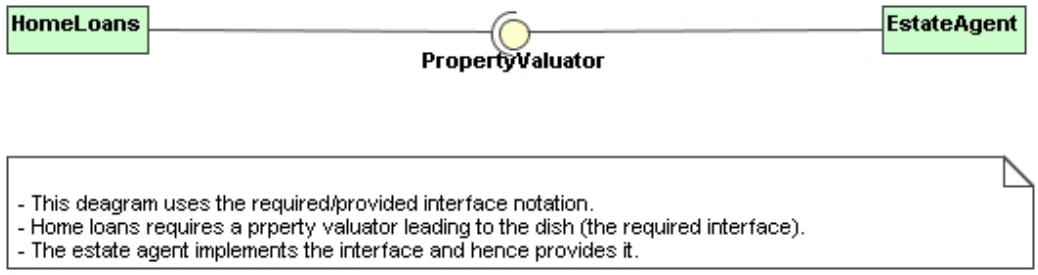

Figure 4.18: Showing the required and provided sides of an interface.

One can show a required interface without specifying any object which provides it as is illustrated in Figure **??**. This notation is particularly useful when designing system or business processes without being yet concerned about the implementation details. The latter would be addressed during the implementation phase where the design is mapped onto a concrete realization of the business or system process.

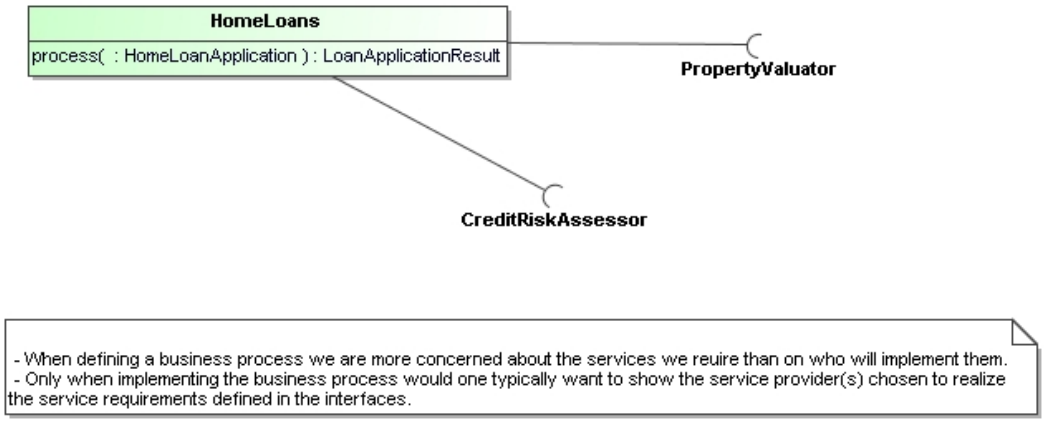

Figure 4.19: The property valuator as a required interface for home loans.

4.8.5 Extending interfaces

UML supports the definition of extended interfaces. The extended interface requires that service providers implementing it must realize not only the services defined in the extended interface, but also those defined in the interface it extends. In UML one uses the specialization relationship to specify that one interface extends another.

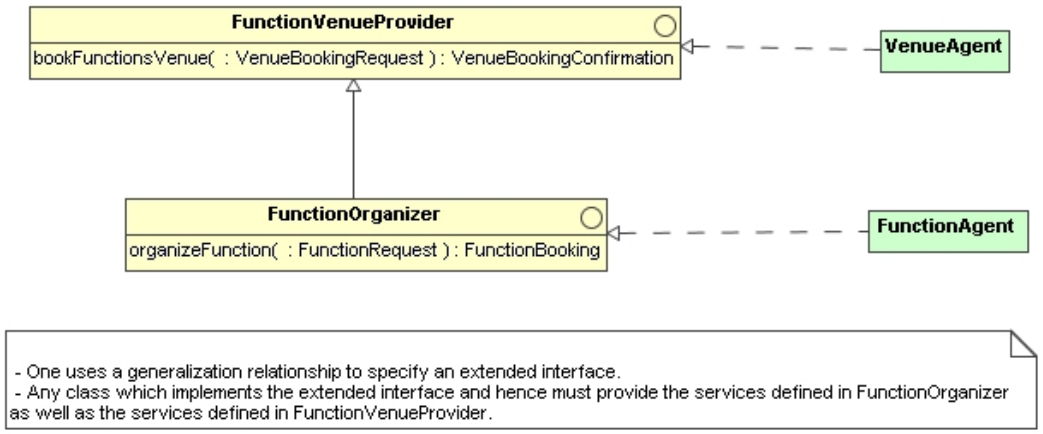

Figure 4.20: The function organizer extending the function venue provider interface.

Some service providers may choose to implement only the base interface, offering only the core services. Other service providers may choose to realize the extended interface. These service providers must realize all the services specified in the extended interface as well as those specified in the base interface.

4.8.5.1 Join interfaces

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

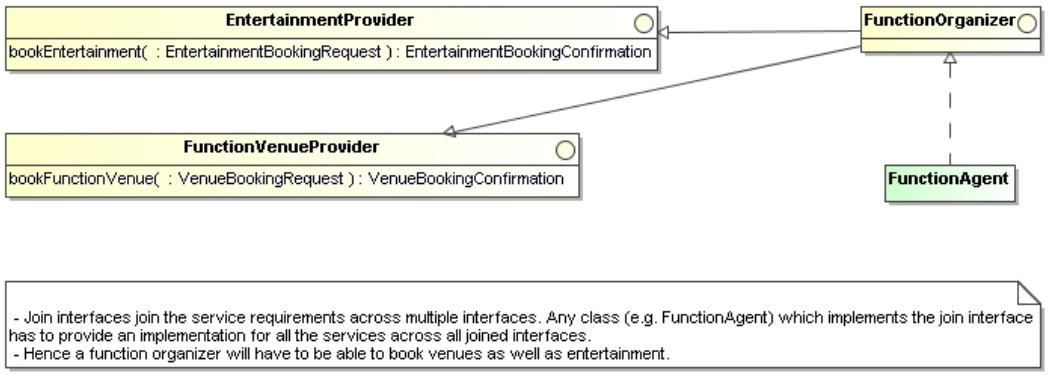

Figure 4.21: The function agent as a join interface.

4.8.6 Using a UML interface to specify a services contract

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

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

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

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

4.8.6.1 Example: a services contract for a caterer

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

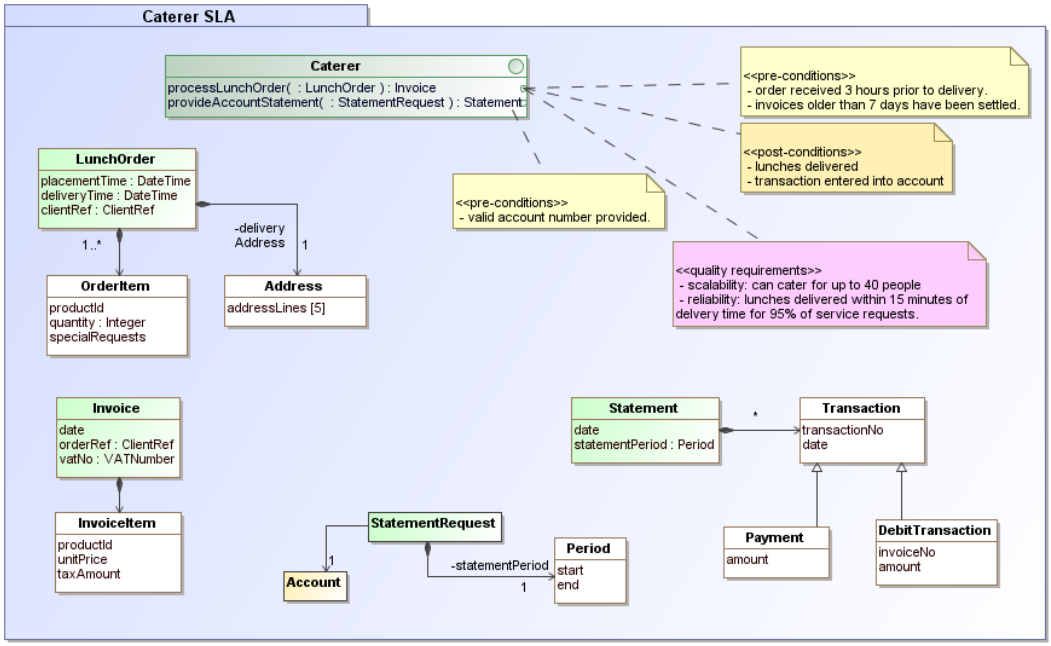

Figure 4.22: A simple services contract for a caterer

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

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

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

4.8.6.2 Example: a message sender

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

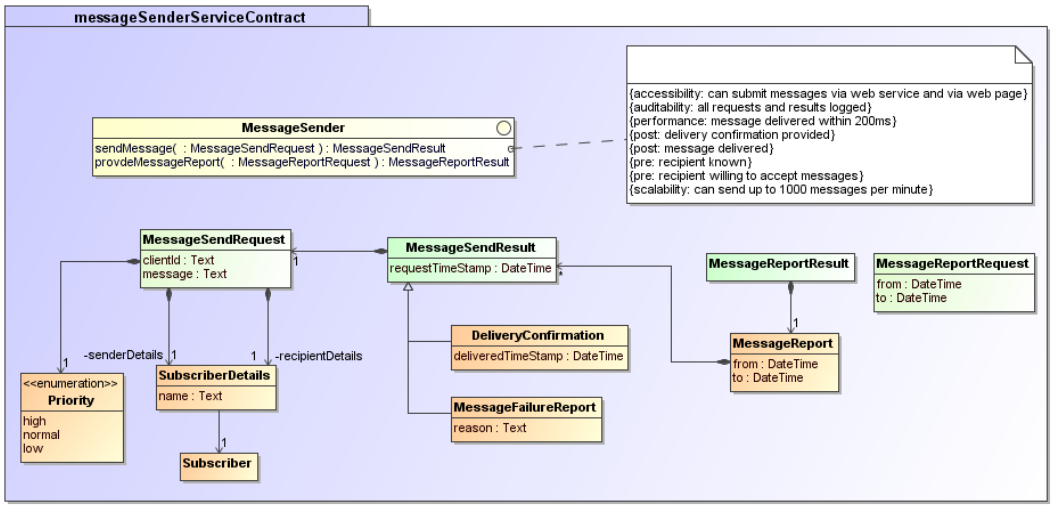

Figure 4.23: A services contract for a message sender

4.8.7 Guidelines for defining interfaces

1. **Define interfaces from the client perspective** An interface should be specified from the perspective of the client, not from the perspective of any particular service provider. It should encapsulate the service requirements for the client's business processes and enable the client to replace one service provider with another without having to make any changes to its business processes.
2. **Include only services from a single responsibility domain** A specific interface should only include narrowly related services around a single responsibility domain. A specific service provider can always implement multiple interfaces. This encourages conceptual simplicity and reusability.
3. **In a client-server relationship, always decouple via interfaces** Why would a client want to lock into a particular service provider? Specifying, in the business process a dependency on an interface instead of a particular implementation results in more flexible business processes where one service provider can easily be replaced with another without changing the business process.

4.9 Interfaces versus classes versus objects

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

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

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

4.10 Class specialization

4.10.1 Introduction

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

4.10.2 UML notation for specialization

In UML one specifies that instances of one class are specializations of instances of another class by drawing a solid line with a triangular head pointing from the specialized to the more general class.

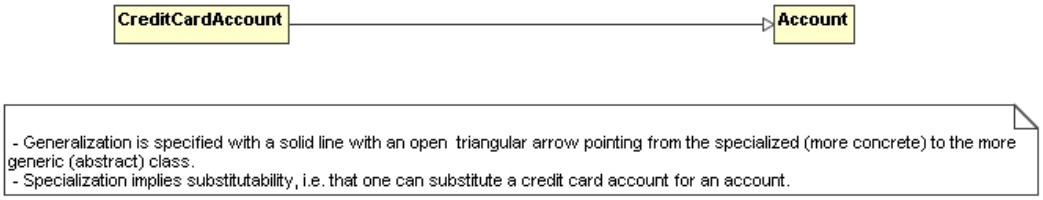

Figure 4.24: CreditCardAccount is a specialization of Account

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

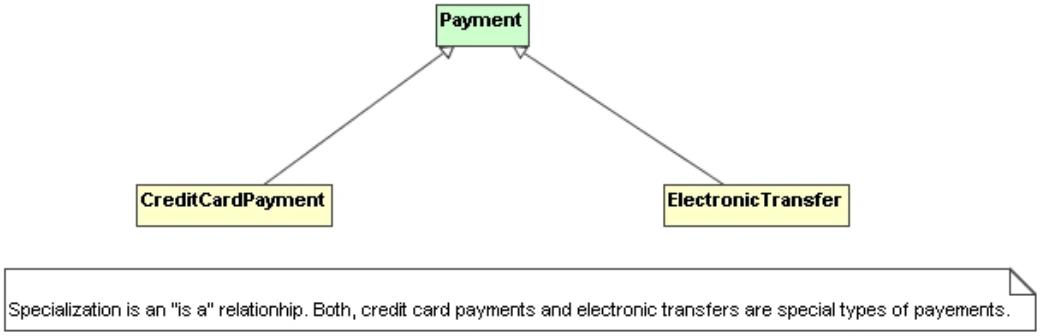

Figure 4.25: Both, credit card and electronic payments are special types of payments.

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

4.10.3 Substitutability

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

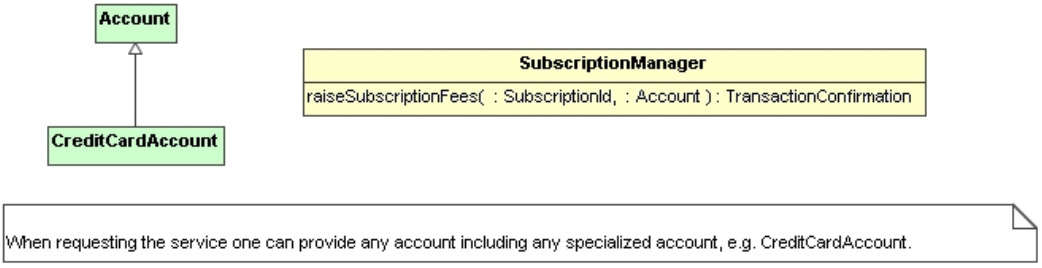

Figure 4.26: One may substitute a credit card account for an account when requesting to raise the subscription fee for a particular subscription number.

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

4.10.4 Inheritance

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

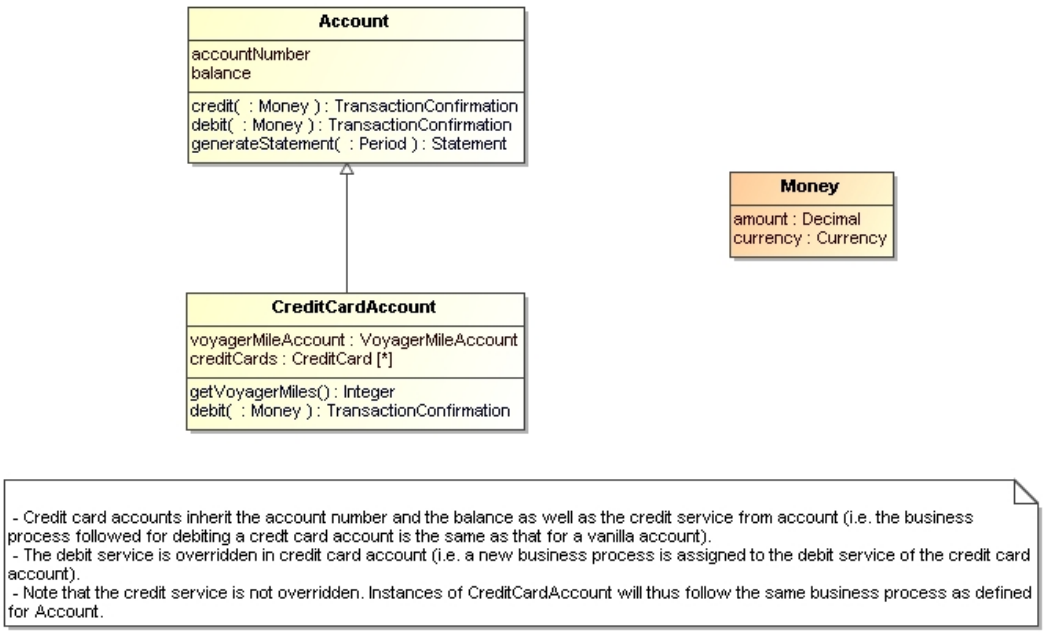

Figure 4.27: CreditCardAccount inherits all attributes and services of Account.

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

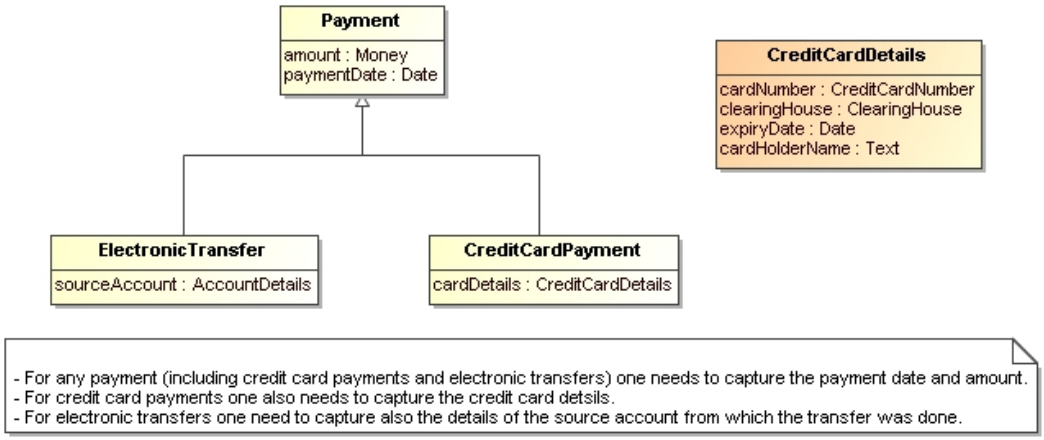

Figure 4.28: For all payments we capture the payment amount and date.

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

4.10.4.1 Inheriting and overriding business processes

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

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

4.10.5 Polymorphism

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

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

4.10.5.1 Polymorphism on message recipient

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

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

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

4.10.5.2 Polymorphism on message parameters

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

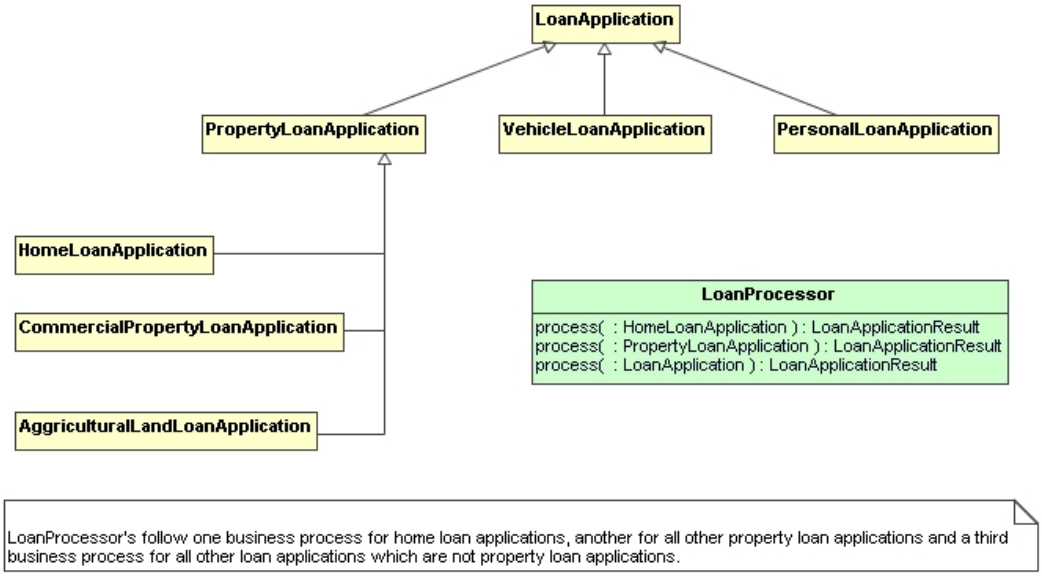

Figure 4.29: Different business processes will be followed for different types of loan applications.

For example, business may require that home loan applications are processed in one way, while all other property loan applications like business property and stand loan applications should be processed using another business process and all other loan applications including personal and vehicle loan applications should be processed via yet another business process. Thus, if the client requests the processing of a loan applications, the actual business process followed depends on the type of loan application provided.

4.10.6 Abstract classes

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

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

UML tools provide a mechanism which enables the user to declare a class abstract. The class name for an abstract class is rendered in italics.

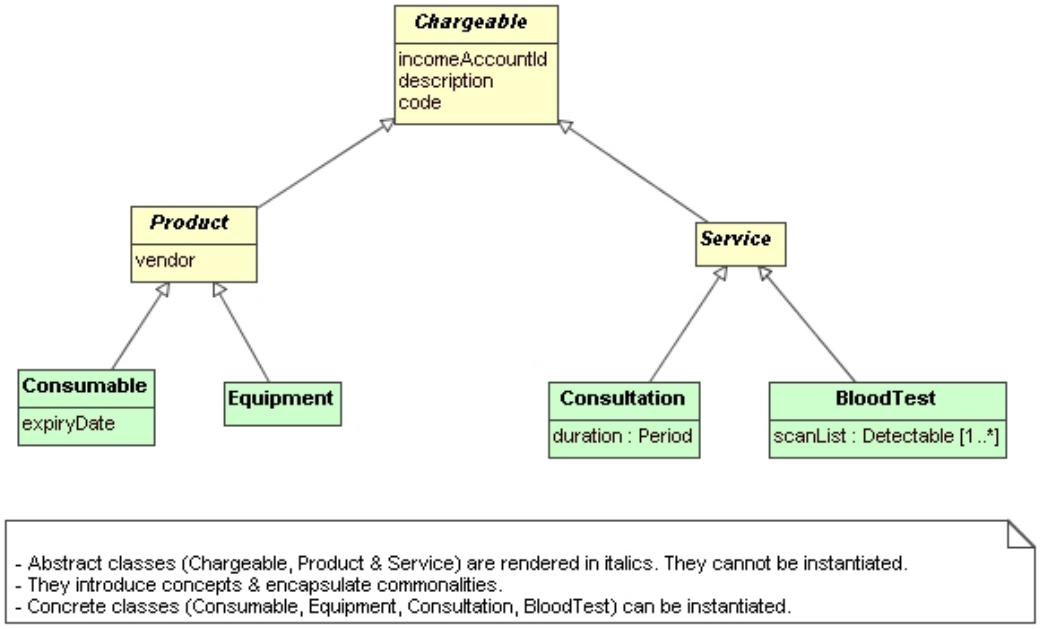

Figure 4.30: A class hierarchy of various levels of abstract chargeables with concrete leaf chargeables.

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

4.10.6.1 Working with abstract concepts

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

Having introduced the concept of a service enables us to specify logic around that concept. For example, we may want to do a resource allocation for scheduled services.

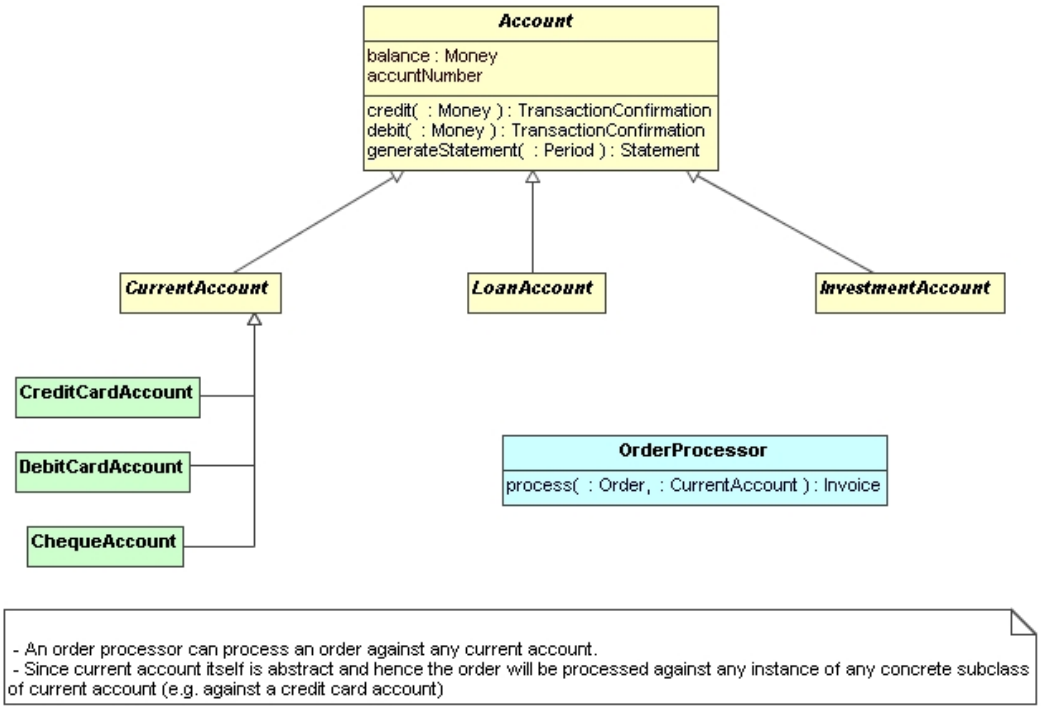

Figure 4.31: Orders are processed against any current account.

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

4.10.6.2 Abstract services

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

For example, we may have different property valuators following different business processes when valuating a property. Some property valuators may request the property valuation through a property agent, others may use a business process which takes the average of the last *n* properties sold within an area and algorithmically adjusts for above average features.

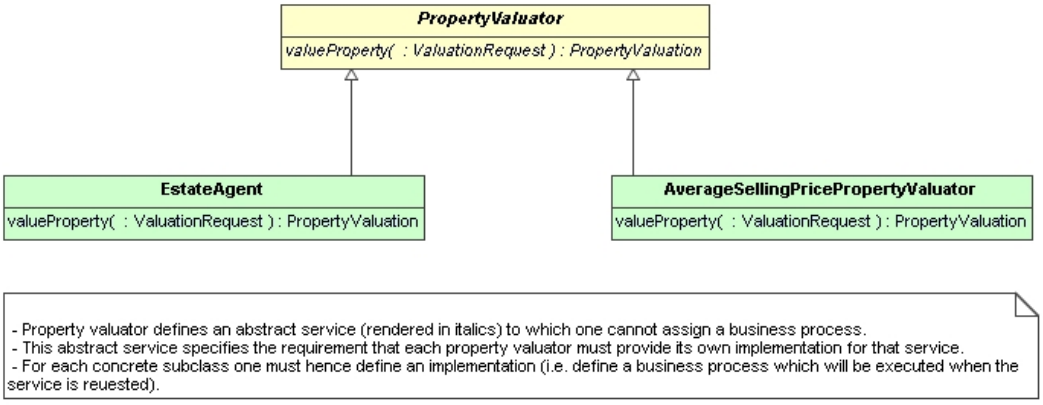

Figure 4.32: An abstract PropertyValuator class with an abstract valuation service.

We may want to specify that property valuators must be able to value a property, leaving the actual process which should be used for the valuation to the implementation of the individual concrete valuatos. In such a case we would declare the valuation service itself as abstract as is done in Figure **??**.

4.10.7 Multiple inheritance

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

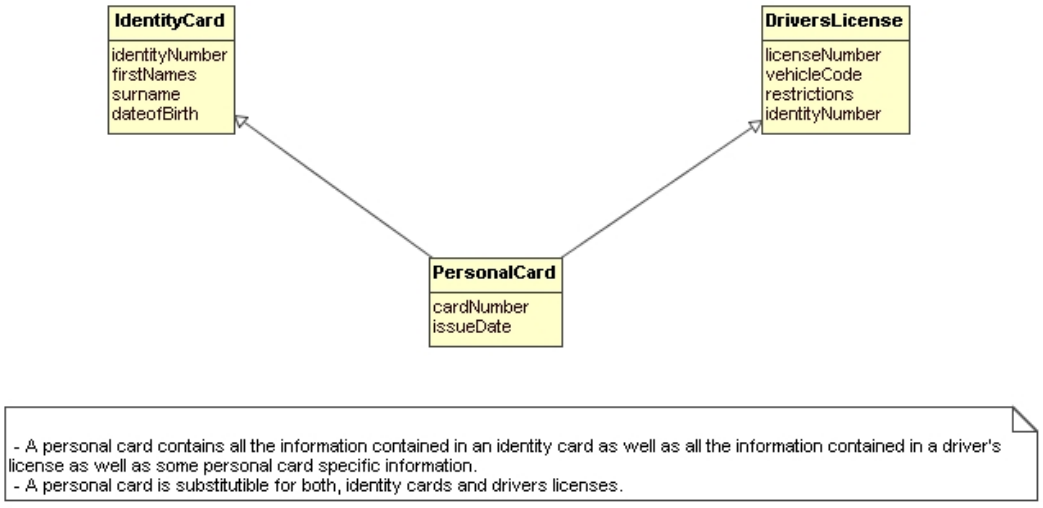

Figure 4.33: A personal card inherits all members of both, identity cards and driver's licenses and is substitutable for both.

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

4.10.8 Completeness constraints

If an aspect of a model should be ficed such that it cannot be refined any further, one assigns a *complete constraint* to that model element.

4.10.8.1 Fixing a business or system process

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

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

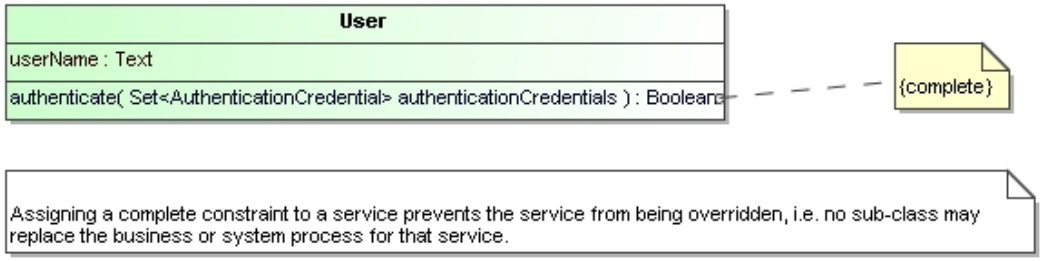

Figure 4.34: Complete constraint on a service prevents the service from being overridden

4.10.8.2 Preventing specialization

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

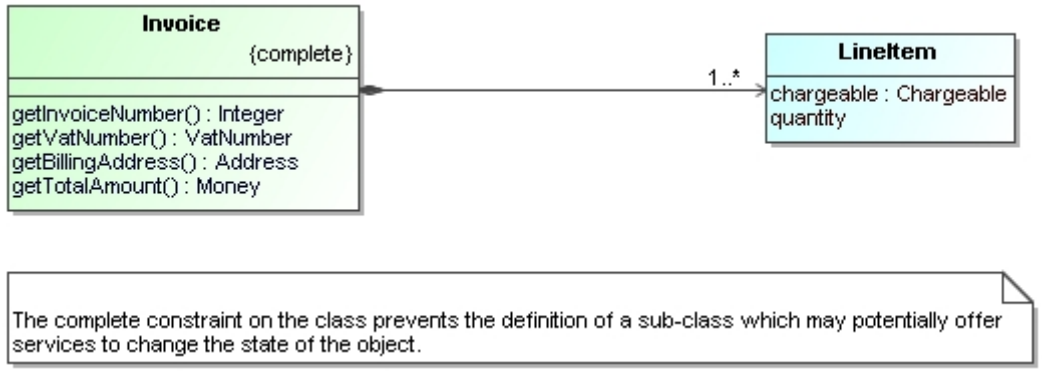

Figure 4.35: Complete constraint on a class prevents specialization of that class

4.11 Association

4.11.1 Introduction

Associations are largely used for two purposes:

- to specify service request channels which clients maintain for certain service providers, enabling the client to, at any stage, request further services from that service provider, and

- to specify navigation paths within data objects or entities which enables one to resolve one piece of information or one entity from another.

4.11.2 UML notation for association

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

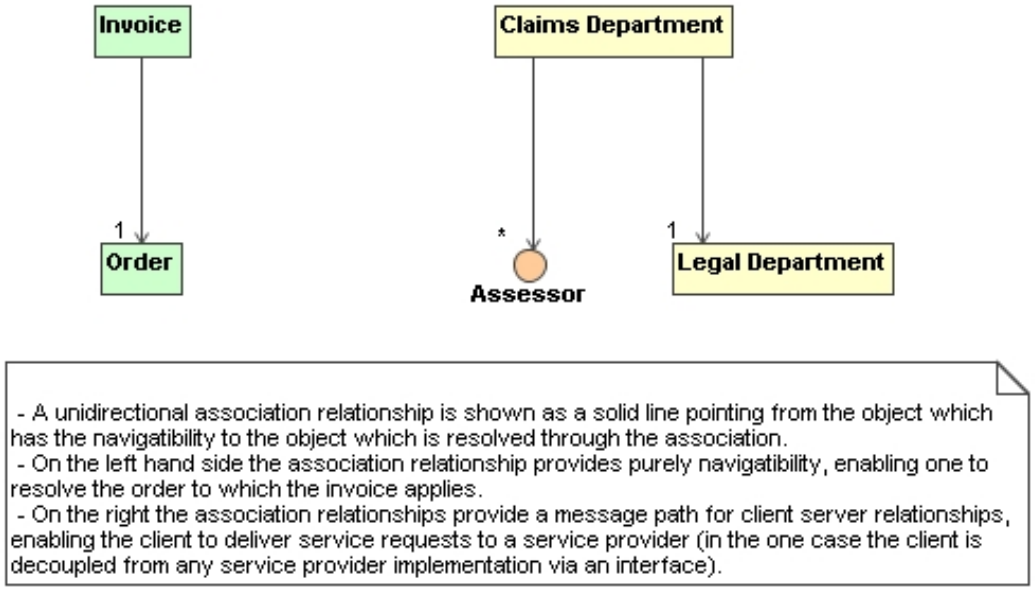

Figure 4.36: The basic notation for a unary association.

4.11.2.1 Role names and cardinalities

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

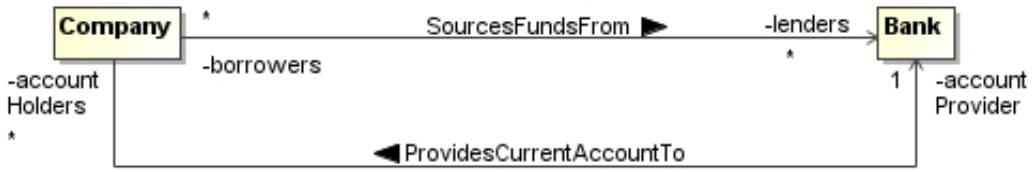

Figure 4.37: Role names, association labels and cardinalities

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

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

4.11.3 Navigatibility

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

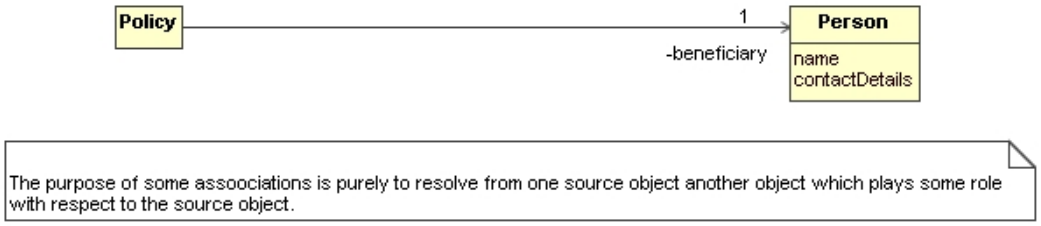

Figure 4.38: An association facilitating the location of associated information.

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

4.11.4 Association for client-server relationships

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

- an organization,
- a business unit within an organization,
- a person,
- a system used by an organization, or
- a system component.

4.11.4.1 Abstracting from service providers

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

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

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

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

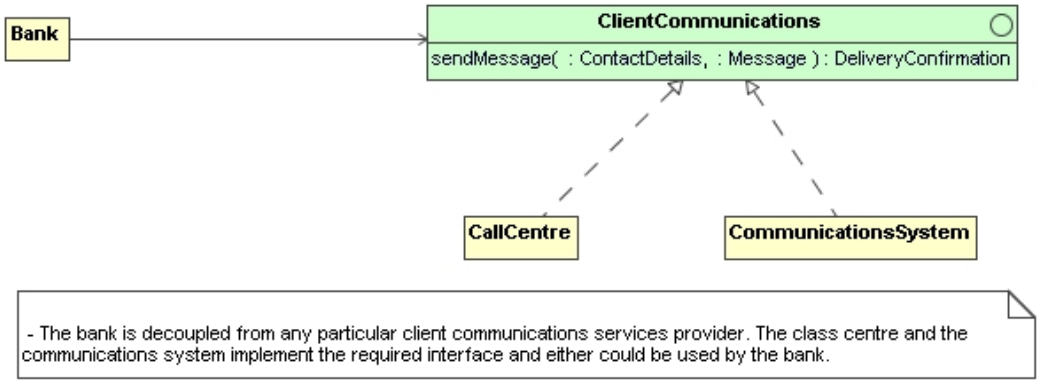

Figure 4.39: Decoupling from a service provider via interfaces.

4.11.4.2 Example message paths

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

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

4.11.4.2.1 When are the types of message paths to be used determined?

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

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

4.11.5 Peer to peer relationships

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

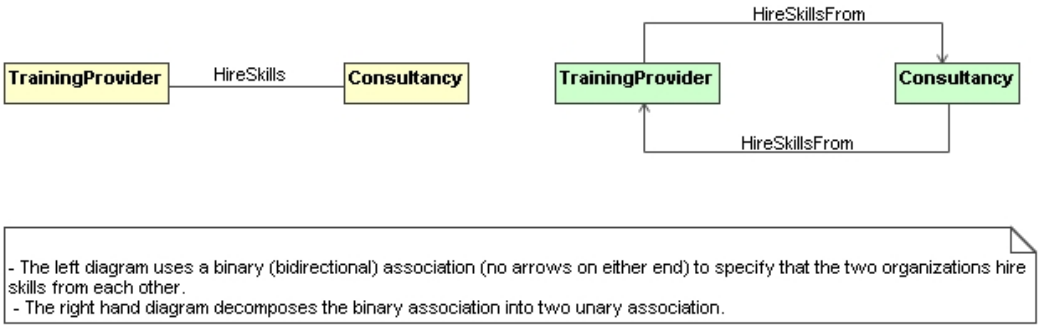

Figure 4.40: A peer-to-peer relationship can be documented using bi-directional associations.

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

4.11.5.1 Decoupling in peer-to-peer relationships

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

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

For example, a jazz club may source catering services from a neighbouring restaurant while the restaurant may use the jazz club to provide it with entertainment services. This binary association with strong coupling between the two classes can be decoupled by decomposing the binary association into two unary associations and inserting the relevant services contracts (see Figure **??**).

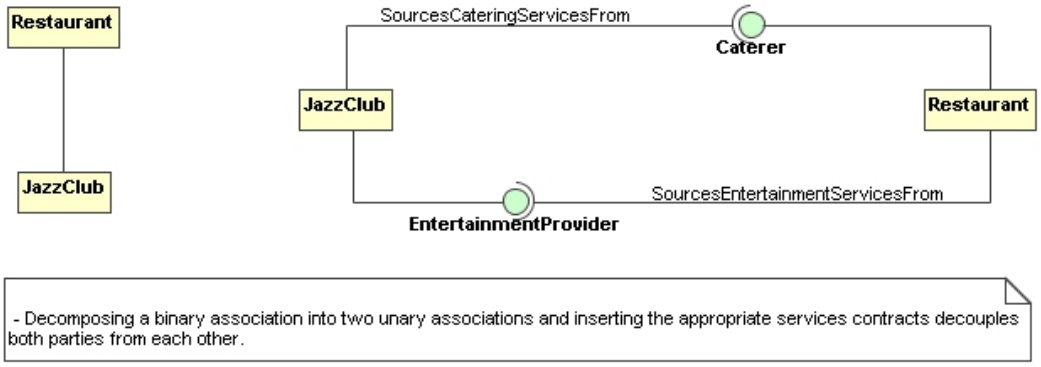

Figure 4.41: Decoupling peer-to-peer relationships via interfaces.

Should the two parties use each other for the same services, they would play the same role for each other. In such a case one would insert a single interface (contract) for that role with both parties realizing that interface and making use of another party which realizes that same interface. inserting services contract.

4.11.6 Association classes

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

4.11.6.1 Uni-directional association classes

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

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

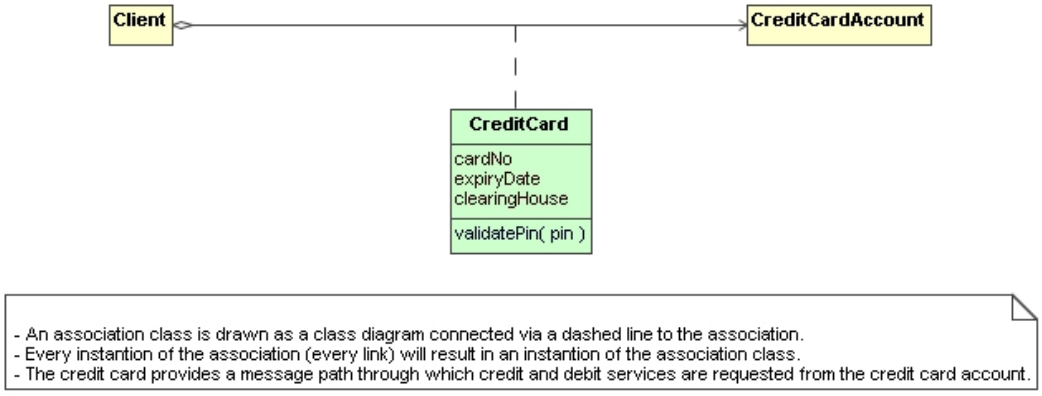

Figure 4.42: A credit card as an association class.

The credit card has its own attributes like the card number and expiration dates and may even potentially render services. For example, the card could have encoded the pin and could validate the pin through logic encoded on the card itself.

Note
Aggregation and composition relationships are special type of associations and hence association classes can be used for aggregation, composition and association relationships.

4.11.6.2 Bi-directional association classes

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

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

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

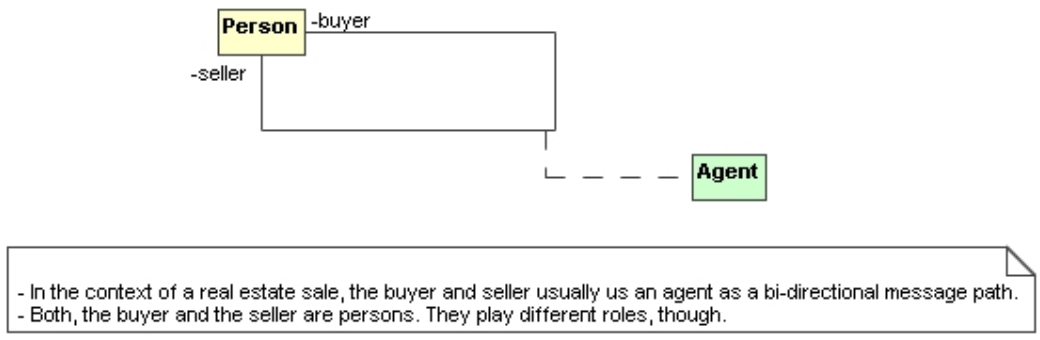

Figure 4.43: A property agent as an bi-directional association class.

4.11.7 N-ary associations

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

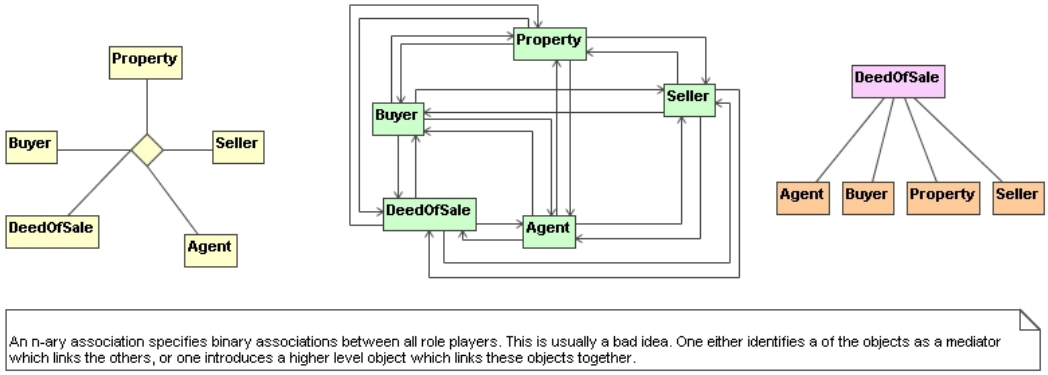

Figure 4.44: An n-ary association between the objects relevant for the sale of a property

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

4.12 Physical message paths used by organizations

Message paths between systems are often realized via message queues or network connections. However when providing an organizational infrastructure, one also needs to decide on the physical message paths used. Examples for such message paths include

- a internal or external postal service,
- a driver or a courier,
- a document chute realized as physical pipes connecting one department with another, ...

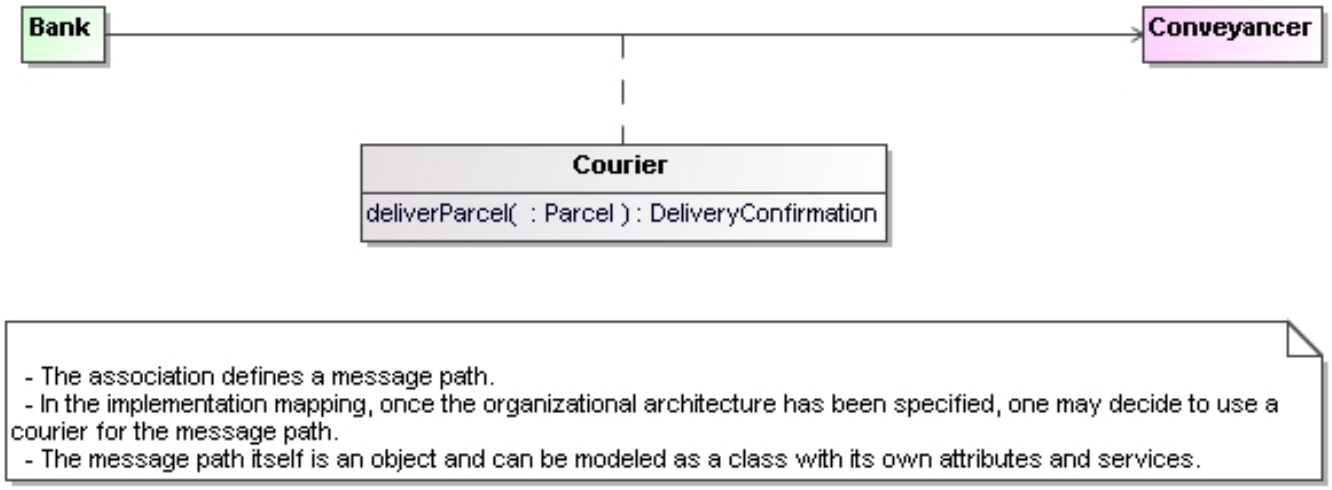

Figure 4.45: Modeling a courier as an association class

The message path used could be documented using association classes as is done in Figure **??**.

4.13 Composition

4.13.1 Introduction

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

4.13.2 UML notation for composition

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

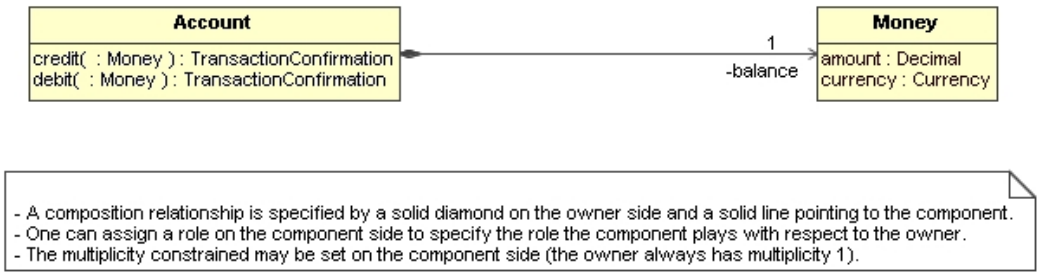

Figure 4.46: The notation for a composition relationship.

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

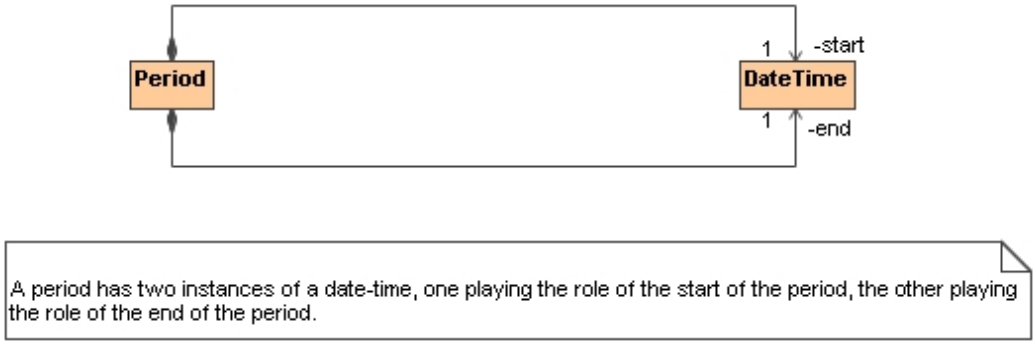

Figure 4.47:

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

4.13.2.1 Using the attributes notation for composition

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

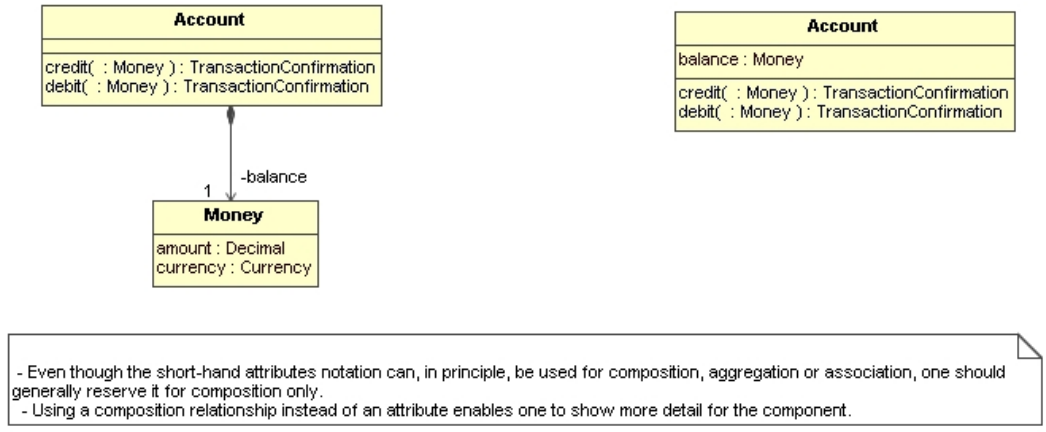

Figure 4.48: Using the attributes notation for composition.

4.13.3 Encapsulation

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

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

4.13.4 Limited life span

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

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

4.13.5 The owner takes responsibility for its components

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

4.14 Composition in organizational modeling

We can use the composition relationship to specify that a particular internal business unit is only accessible by its owner. For example, a bank may have a home loans department which, in turn, may have an internal legal department which, among other things, may generate the property bond contracts. In the case where no other business unit bar home loans is able to request services from this internal legal department, one would specify the legal department as a component of the home loans department via a composition relationship. Other departments would have to request the legal services from this internal legal department via the home loans department.

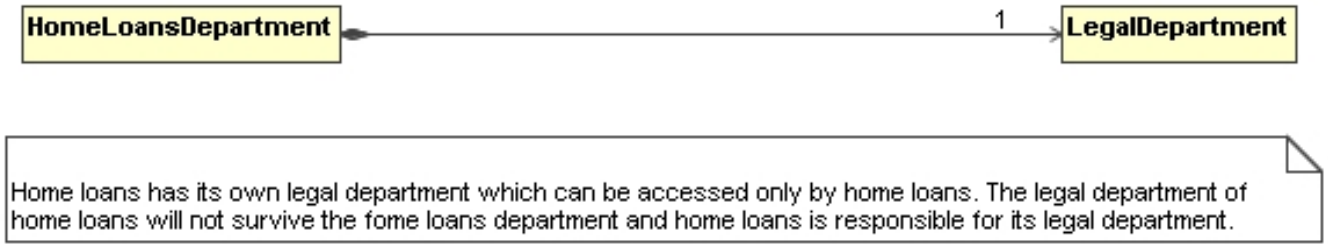

Figure 4.49: The legal department is only accessible via the home loans department.

Should the organization (e.g. the bank) decide to close its home loans department, the legal department would be closed too. Furthermore, the home loans department is responsible for its legal department. It is responsible for paying the salary of the staff of the legal department and any issues with the legal department could be escalated to the home loan department.

4.15 Aggregation

4.15.1 Introduction

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

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

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

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

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

4.15.2 UML notation for aggregation

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

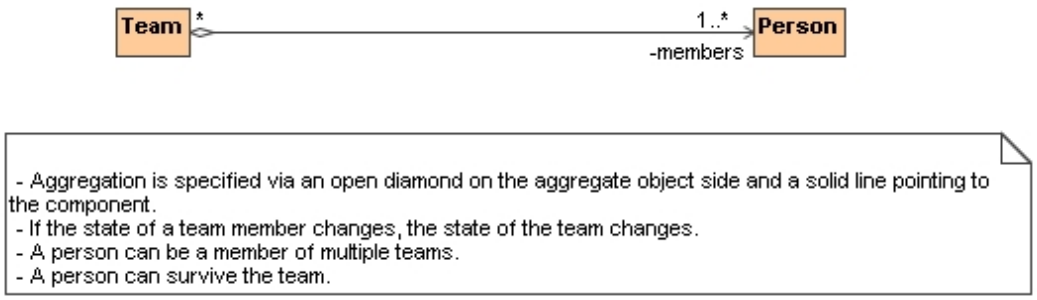

Figure 4.50: A team has multiple persons via aggregation.

4.15.3 Difference between aggregation and composition

To understand the difference between aggregation and composition, consider the example of a drawing composed of multiple graphics objects, each of which has a style. We could specify the graphics object has a style via composition or via aggregation.

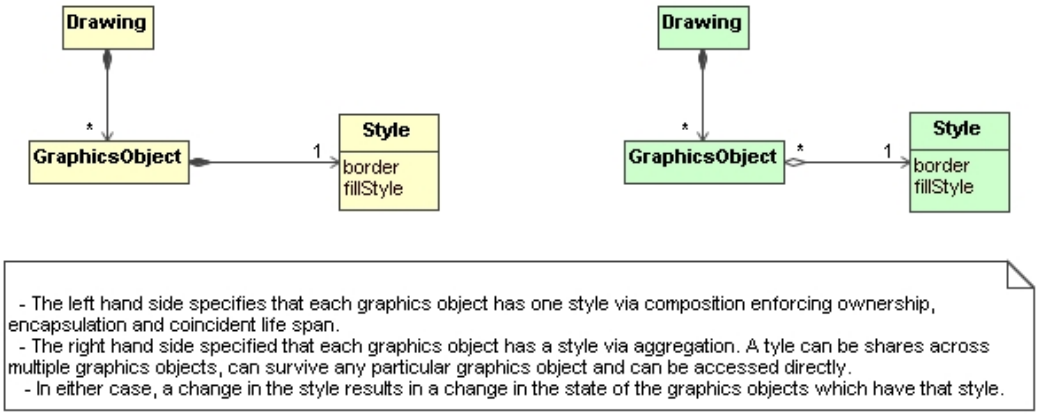

Figure 4.51: Graphics object has a style either via aggregation or via composition

Consider the first scenario where a graphics object has a style via composition. In this case the style can be accessed only via its owner, for example by right-clicking on a graphics object and editing its style. Changing the style of a particular graphics object would not change the style of any other graphics object as a component can have only a single owner. If the graphics object is deleted, so will its style be -- the style cannot survive the graphics object.

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

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

4.16 Dependencies

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

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

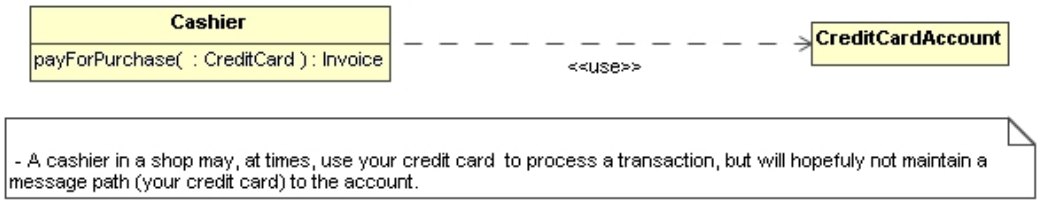

Figure 4.52: The cashier has a dependency with the credit card account class

For example, Figure **??** specifies that cashiers make at times use of credit card accounts without maintaining a service request path to any particular credit card account. The client requests the cashier to process a payment for a transaction, supplying a message path to his or her credit card account, the credit card itself. The credit card enables the cashier to send service request messages to the credit card account. However, with some luck, the cashier will not maintain the service request card to the client's credit card account (the credit card itself), but will instead return the credit card once the transaction has been processed.

As a second example, consider the relationship between an architect and a builder. The architect is asked to oversee the building of a house according to some plans. The architect is given a message path to the builder (perhaps a telephone number) as well as the architectural plans. The architect sends building instructions to the builder. However, once the construction of the house is completed, the architect no longer maintains a message path to that particular builder.

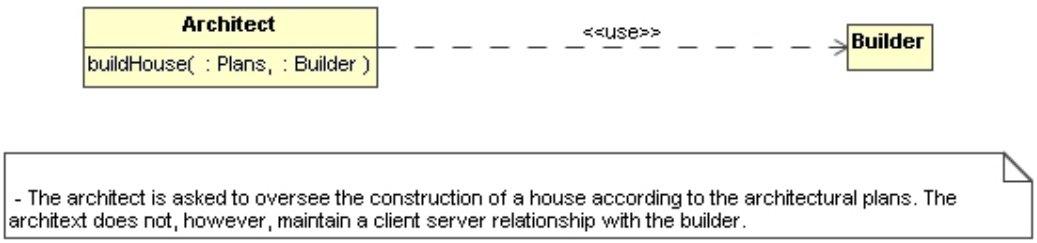

Figure 4.53: The architect has a dependency on the builder

4.17 Containment

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

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

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

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

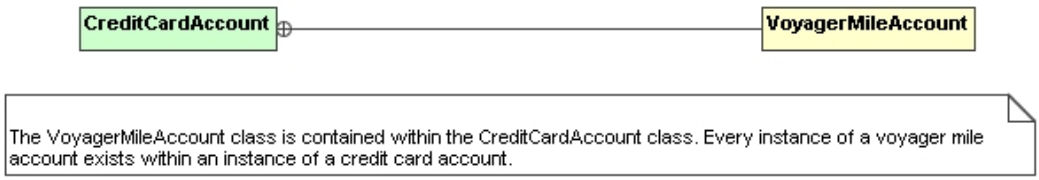

Figure 4.54: Voyager mile account as an inner class

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

4.18 Containment in organizational modeling

At times, the UML containment relationship can be used within organizational modeling. Consider, for example, a business model for a pharmaceutical retailer which specifies that it does not have any stand alone retail outlets (pharmacy), but that each pharmacy is deployed within an instance of a supermarket.

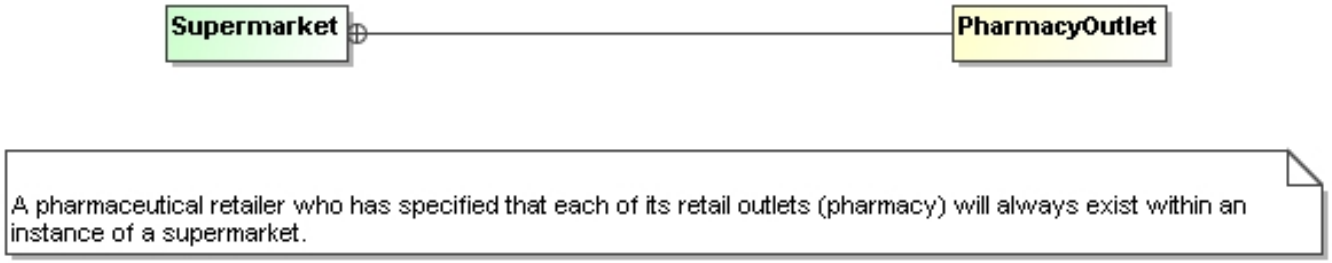

Figure 4.55: Embedded pharmaceutical retail outlets as inner classes

Such a business decision is aptly modeled using an inner class, i.e. a containment relationship. Not only will the containment relationship enforce that every instance of the pharmacy will have to exist within an instance of a supermarket, but it also has as consequence that the staff of the embedded pharmacy has access to the private services of the supermarket (e.g. the staff facilities).

4.19 Summary of UML relationships

Figure **??** summarizes the UML relationships. It shows that these are conceptually specializations of each other and that we have weak and strong variants of ‘ *is a* ’, ‘ *has a* ’ and ‘ *uses* ’.

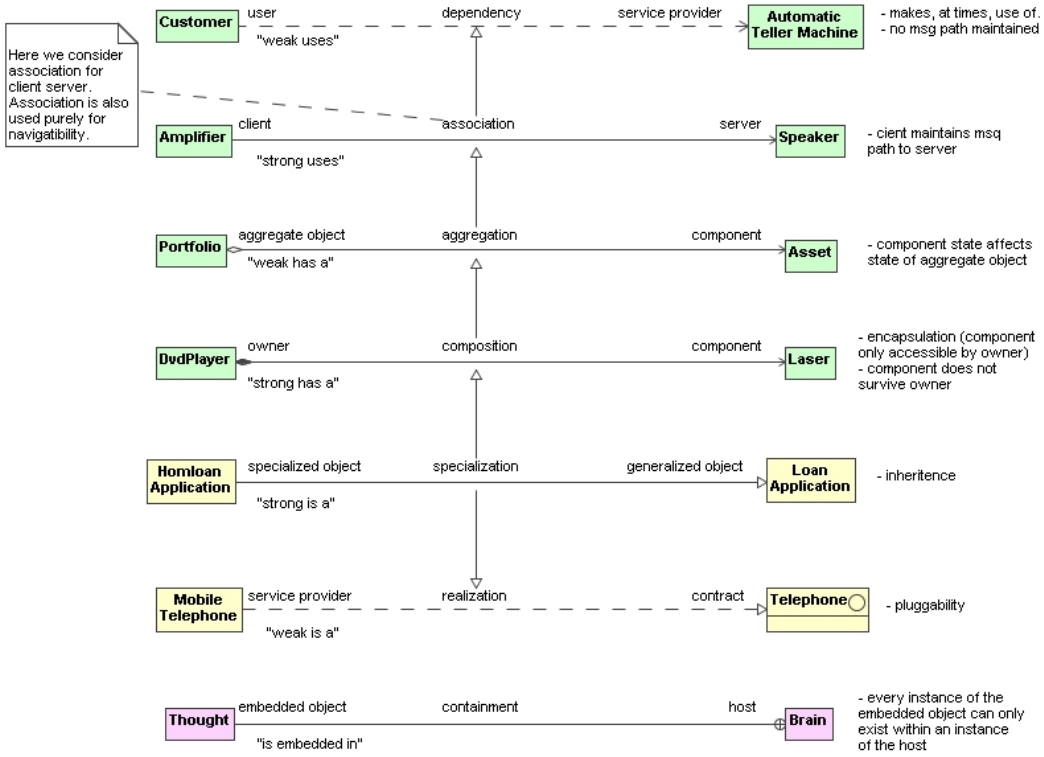

Figure 4.56: Summary of UML relationships

4.19.1 Dependency

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

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

4.19.2 Association

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

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

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

4.19.3 Aggregation

Aggregation is a special form of association. The aggregate object still maintains a message path to the component. It still can make use of the components. For example, in the context of a portfolio calculating its value, it will request the value of each asset and sum them up.

However, in aggregation a state transition in the component may imply a state transition in the aggregate object, i.e. aspects of the component state are part of the state of the aggregate object. In our example, a change in the value of any of the assets results in a change in the value of the portfolio.

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

4.19.4 Composition

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

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

4.19.5 Realisation

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

4.19.6 Specialisation

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

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

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

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

4.19.7 Containment

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

4.19.8 Shopping for relationships

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

The process of determining the correct relationship goes along two legs. On the one side you are trying to establish the type of dependency between the two classes. On the other side you will assess the level of substitutability and inheritance required.

First we assess whether there is a dependency between the classes. If instances of one class, A, never make use of instances of another class, B, and if one also does not need to be able to navigate from an A to a B, then there is not much of a relationship between these classes. Otherwise there is at least a dependency of A on B.

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

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

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

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

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

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

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

4.20 Templates

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

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

4.20.1 Template classes

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

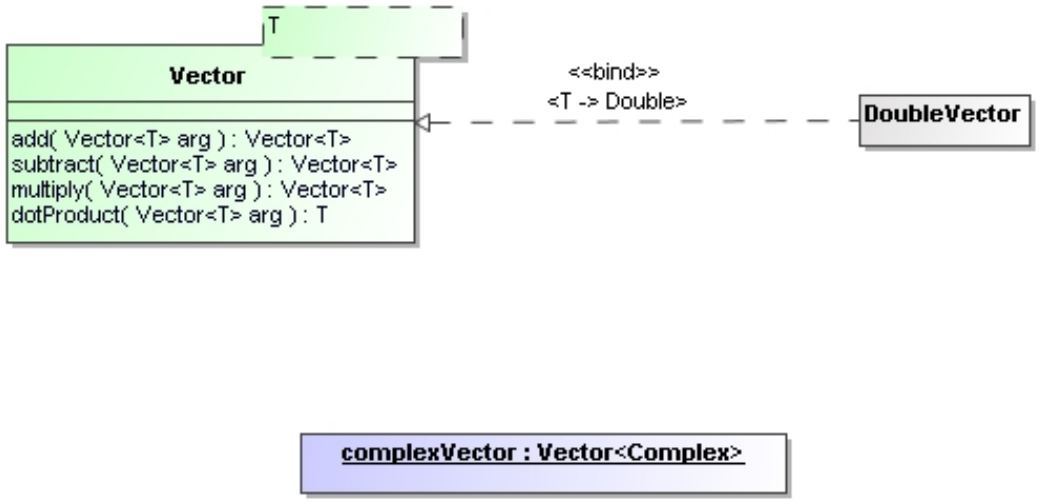

Figure 4.57: Vector template

For example, Figure **??** shows a vector class template, defining a parametrized vector class. `Vector<Double>` and `Vector<Complex>` are two different vector classes generated from that template by binding the template type to `Double` and `Complex` respectively.

We can either show how the data type, `DoubleVecotr` is generated from the template by binding the template paramter, `T` to `Double`, or we can directly define an object as an instance of `Vector<Complex>`.

4.21 Selective views

Recall that the UML diagrams provide a mechanism

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

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

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

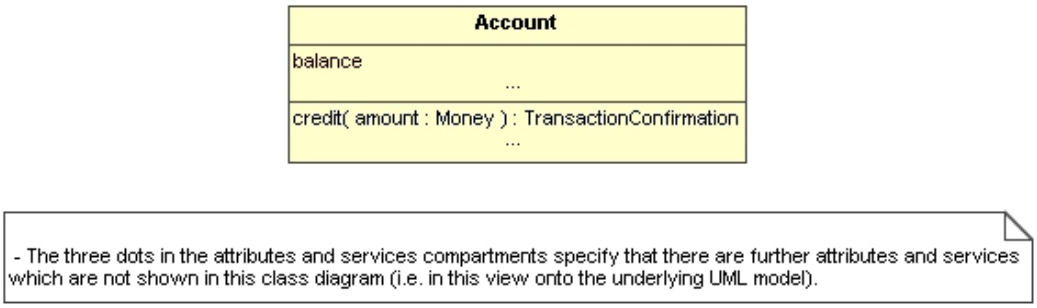

Figure 4.58: A class diagram which only shows a subset of the attributes and services of a class

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

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

Chapter 5

Sequence diagrams

5.1 Introduction

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

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

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

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

5.2 Simple sequence diagrams

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

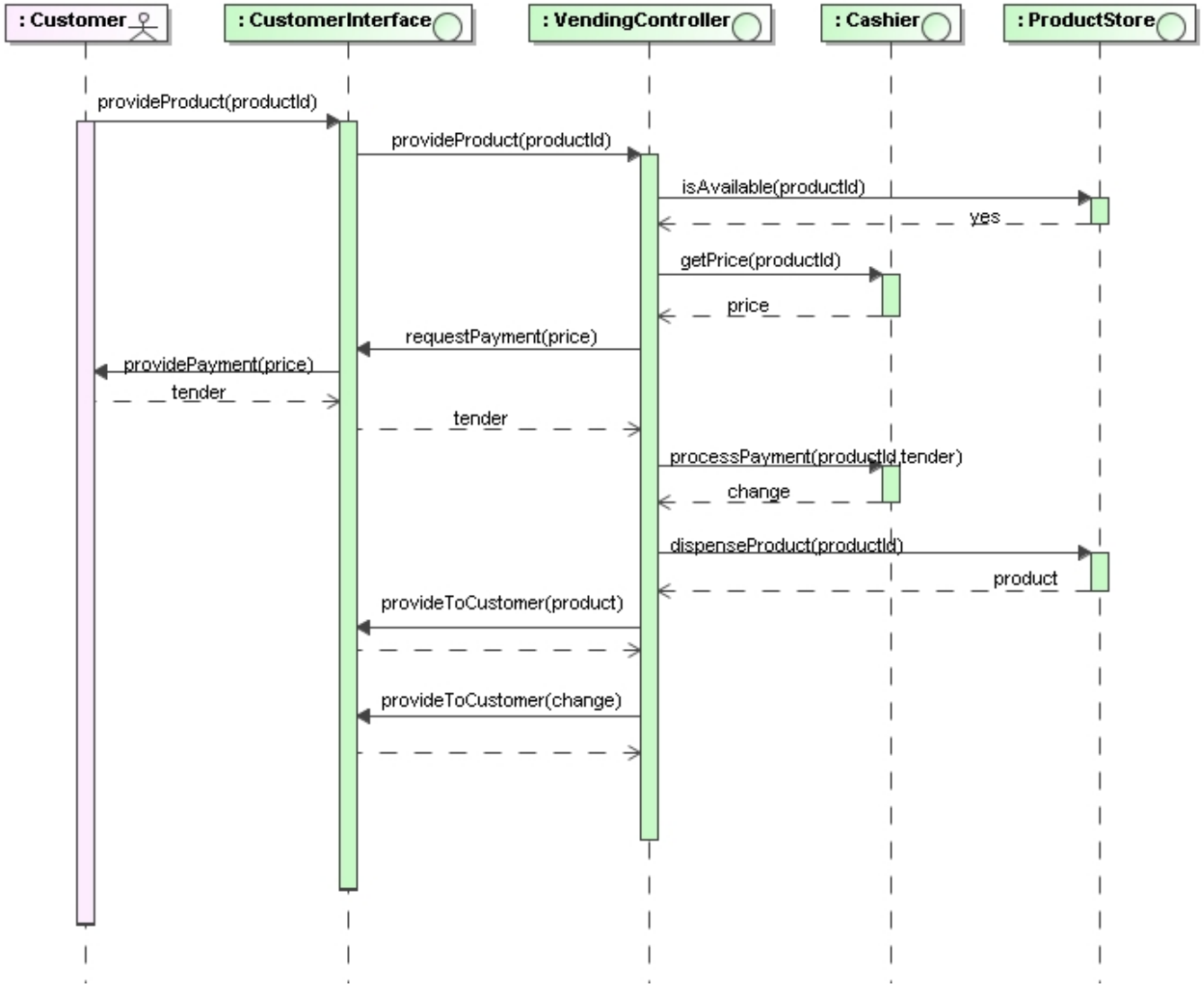

Figure 5.1: High level sequence diagram for the buy product use case of a vending outlet

5.2.1 The time axis

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

5.2.2 The objects

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

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

5.2.2.1 The life line

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

5.2.2.2 The activation bar

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

5.2.3 Service requests

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

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

5.2.3.1 Message to self

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

5.2.4 Returns

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

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

5.2.5 Levels of granularity

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

5.3 Message types

A message is used to either communicate some information or to request a service. In either case a client may choose whether to wait for a response or not.

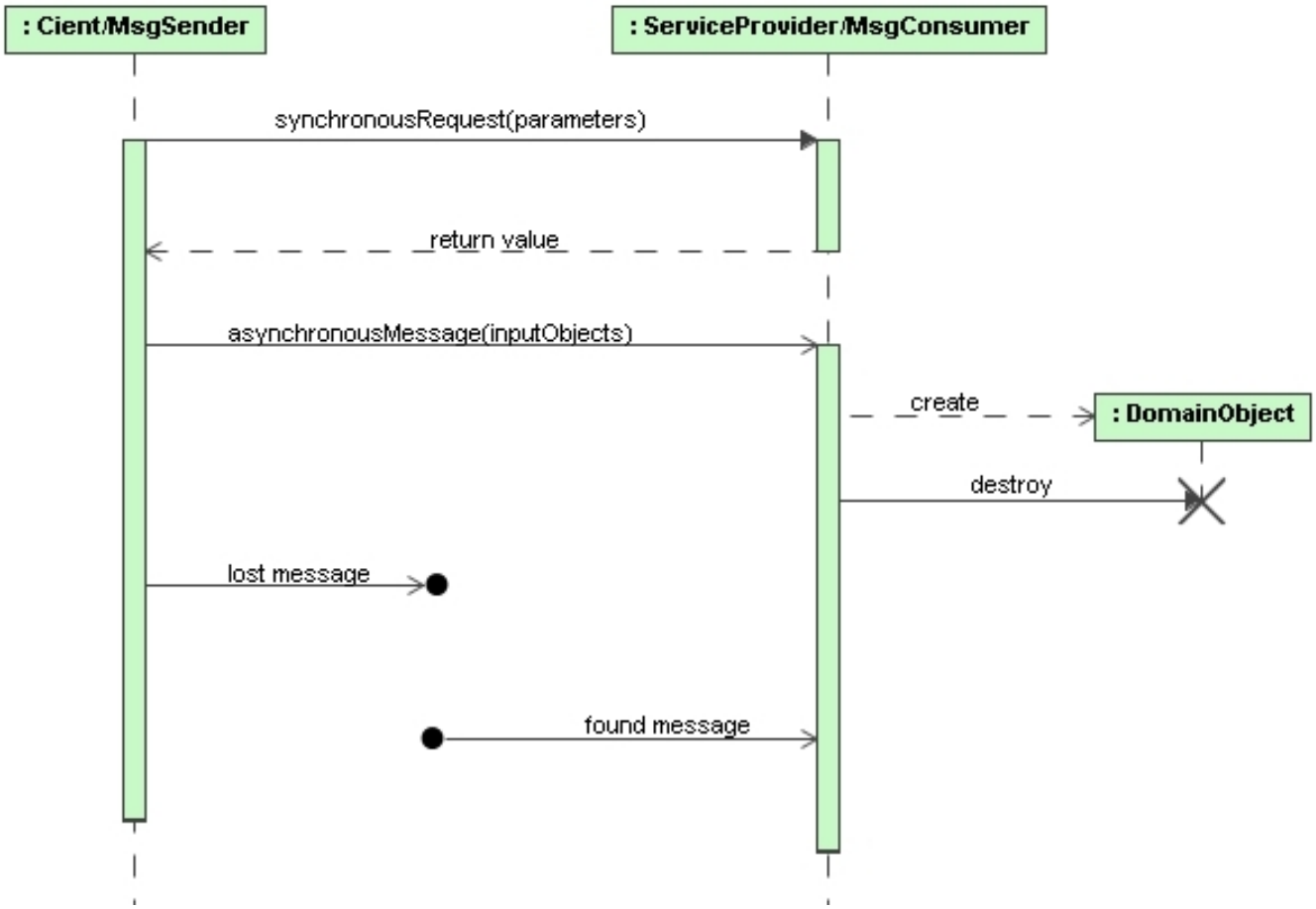

Figure 5.2: UML notation for various message types

UML supports the following message types in interaction diagrams:

- **Synchronous request and return** A synchronous request is a blocking service request where the business or system process of the client or sender blocks until a response has been received. A synchronous request is shown in UML using a solid line with a filled arrow head pointing from the client or message sender to the service provider or message consumer. The return of a synchronous request must always go back to the client who made the initial synchronous request. It is shown as a dashed line with an open arrow head pointing from the service provider or message consumer to the client or message sender.
- **Asynchronous messages** Asynchronous messages are non blocking. The client does not wait for a response and directly continues with its business or system process after having dispatched the message. Asynchronous messages are shown in UML using a dashed line with an open arrow.
- **Object creation** Often one needs to domain objects within a business or system process. For example, at some stage in a vending process one needs to create an invoice. This is done in UML using a create message which points to the actual object diagram. The object did not exist before the create message and no life line is present for these earlier times.

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

- **Destroy message** A destroy message destroys the object, i.e. the object will no longer exist after the destroy message has been received. This is shown in UML by putting a cross at the end of the life line of the destroyed object.

- **Lost message** A lost message is a message for which the sender is known, but the receiver is not. It is shown in UML by drawing a message from the life line of the sending object to a circle representing an unknown message recipient.
- **Found message** A found message is a message for which the receiver is known, but the sender is not known. A found message is shown by drawing a the message from a solid circle to the life line of the object which receives the message.

5.4 Timing constraints

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

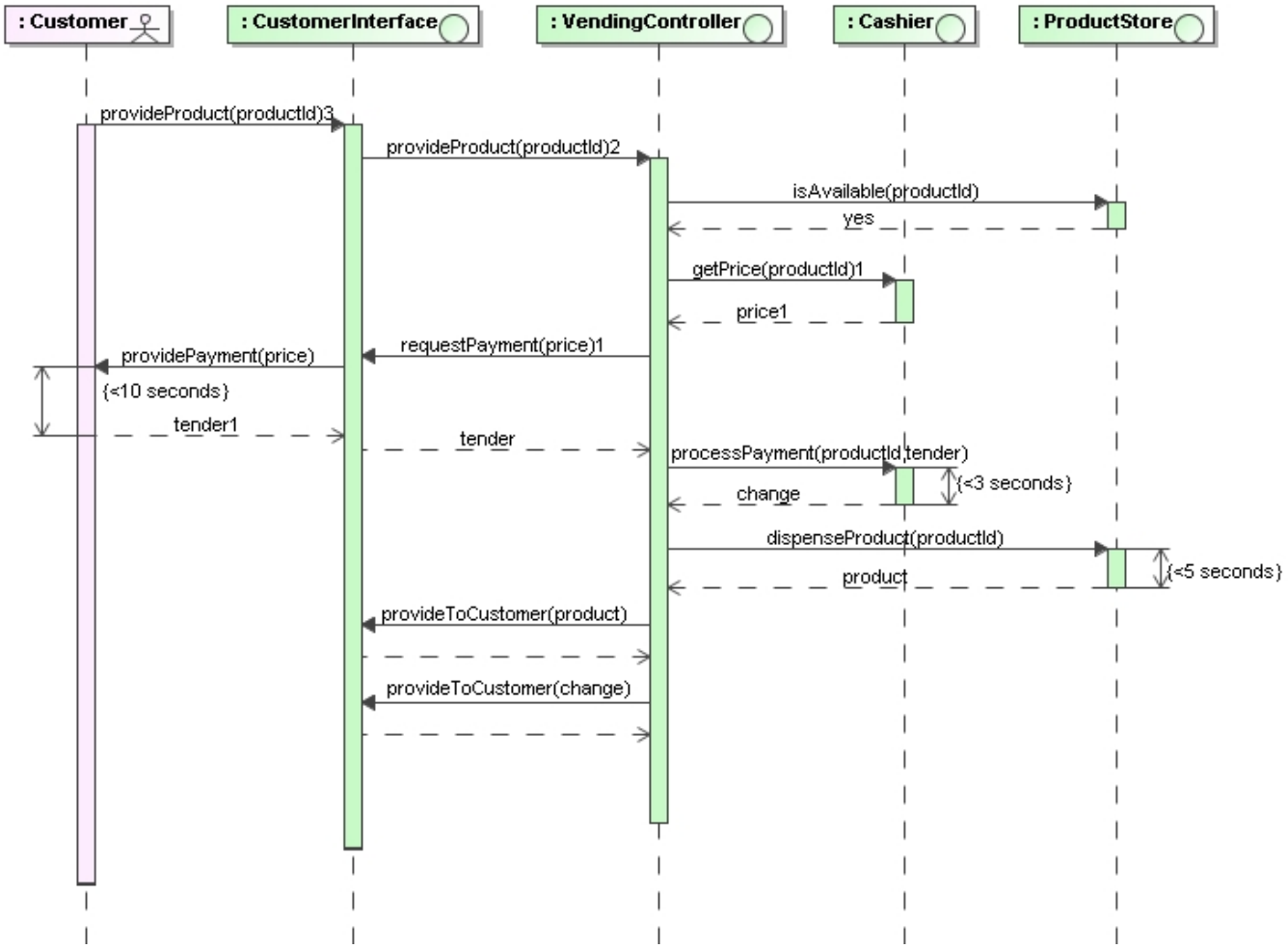

Figure 5.3: Duration constraints for the vending outlet

For example, Figure **??** specifies that

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

5.5 Interaction references

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

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

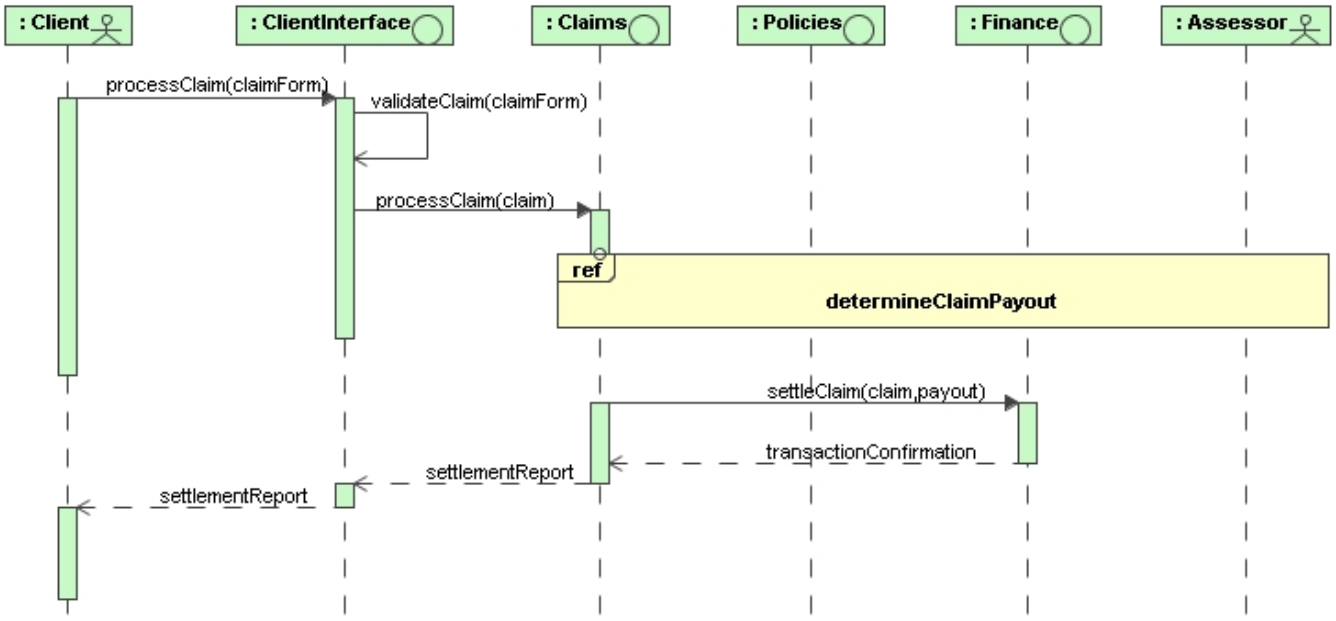

Figure 5.4: Interaction for a process claim use case references the determineClaimPayout interaction

The referenced interaction will have to specified in another sequence diagram. For example, the referenced *determineClaimPayout* interaction is shown in

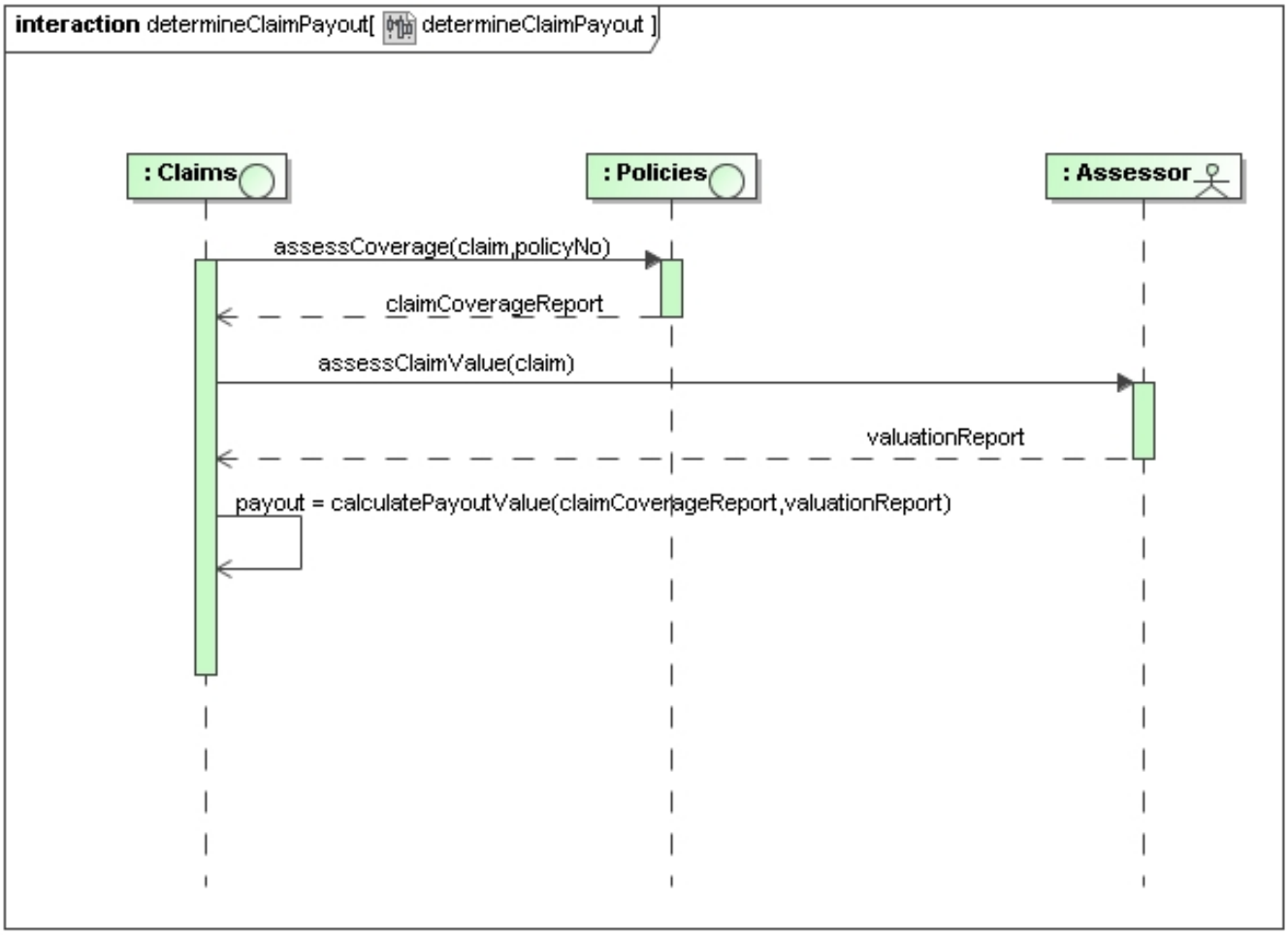

Figure 5.5: The determine claim payout interaction

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

5.6 Conditional flow (alt)

One would, for sake of simplicity, often use sequence diagrams to document only single scenarios, particularly when discussing a business process with the client. Nevertheless, sequence diagrams can be used to show alternate flows, i.e. multiple scenarios. This is done using `alt` fragments with multiple compartments for the alternate scenarios.

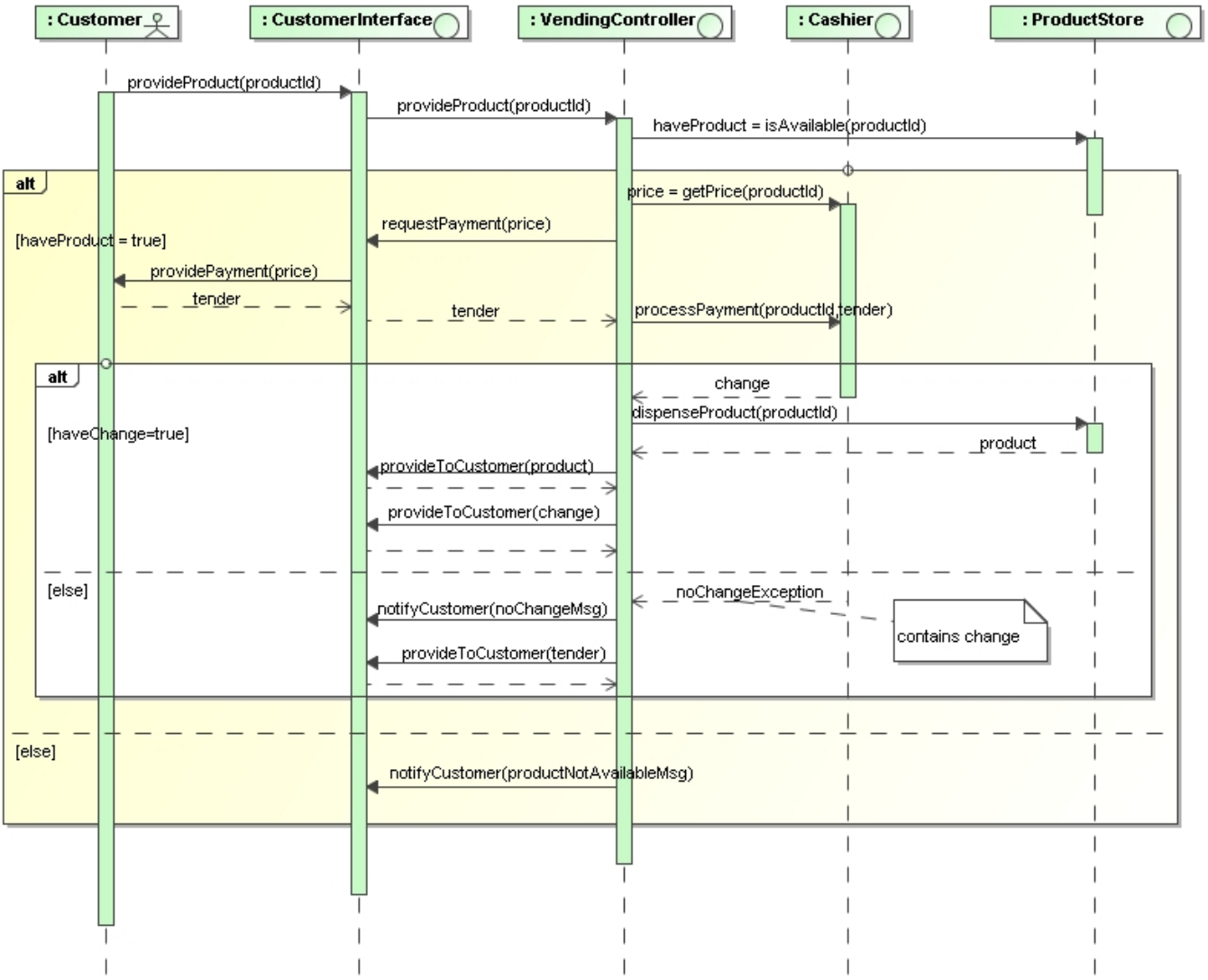

Figure 5.6: Sequence diagram for the buyProduct use case of a vending outlet showing alternative flows

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

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

In order to show multiple branchings of a flow, one may nest *alt* fragments as is done in Figure **??**.

5.7 Iteration in sequence diagrams (loop)

Iteration is not natural to sequence diagrams. UML has, however added a looping construct to support a notation for specifying iterative execution of an interaction. This is done by encapsulating in a fragment containing a loop operator.

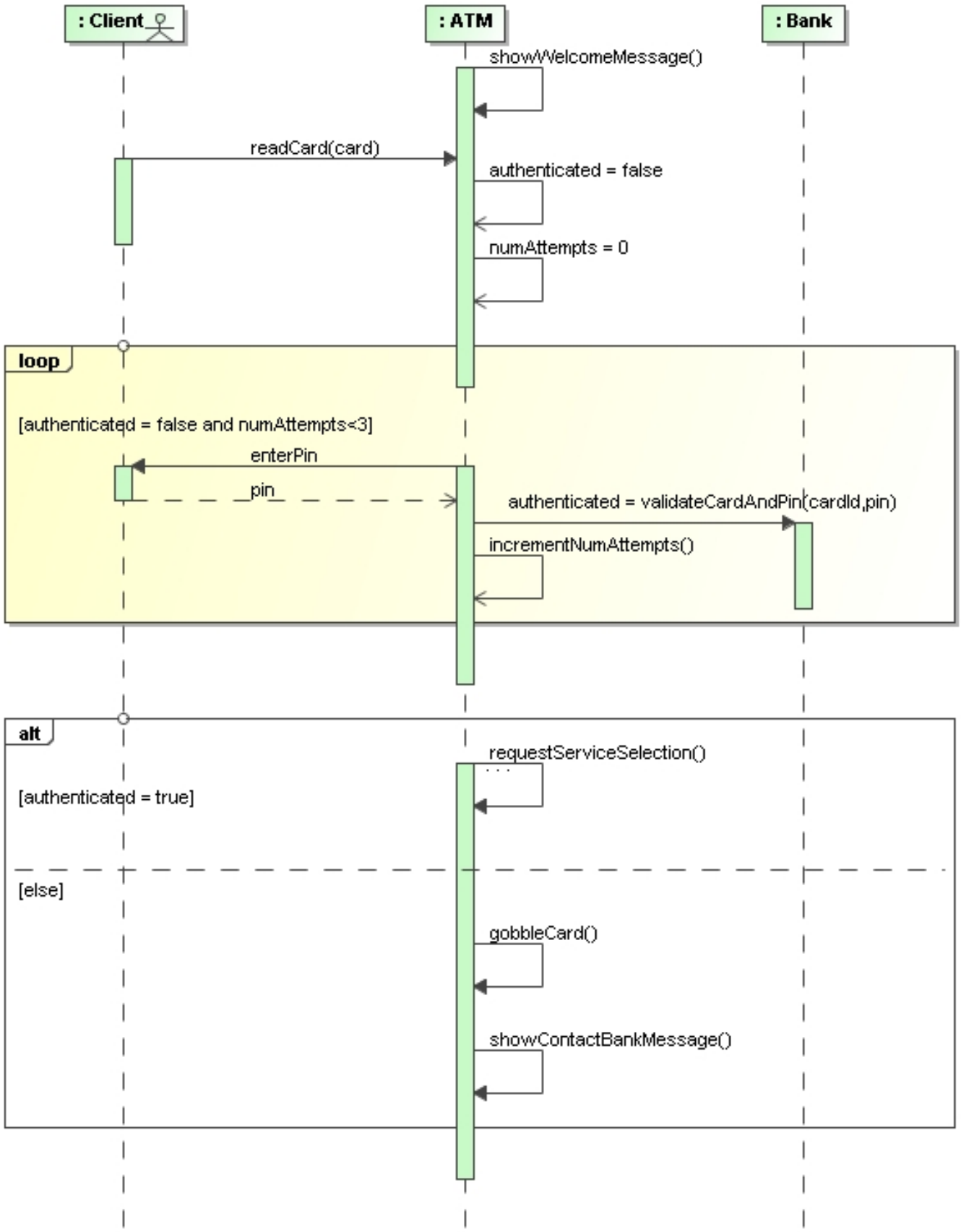

Figure 5.7: Iteration via the loop operator

Figure **??** shows an ATM repetitatively requesting the pin for an ATM card until either the correct pin has been entered or until the user entered the pin three times incorrectly.

5.8 Concurrencies in sequence diagrams (par)

Sequence diagrams do not accommodate concurrencies in a particularly natural way. Activity diagrams have a more intuitive and readable notation for concurrencies and synchronization points within a business or system process.

If one wants to show concurrencies in a sequence diagram, one uses a *par* (parallel) fragment which shows the messages exchanged concurrently in separate sub-divisions of the fragment. There may be two or more subdivisions, each representing a separate concurrent process (a separate thread of activities).

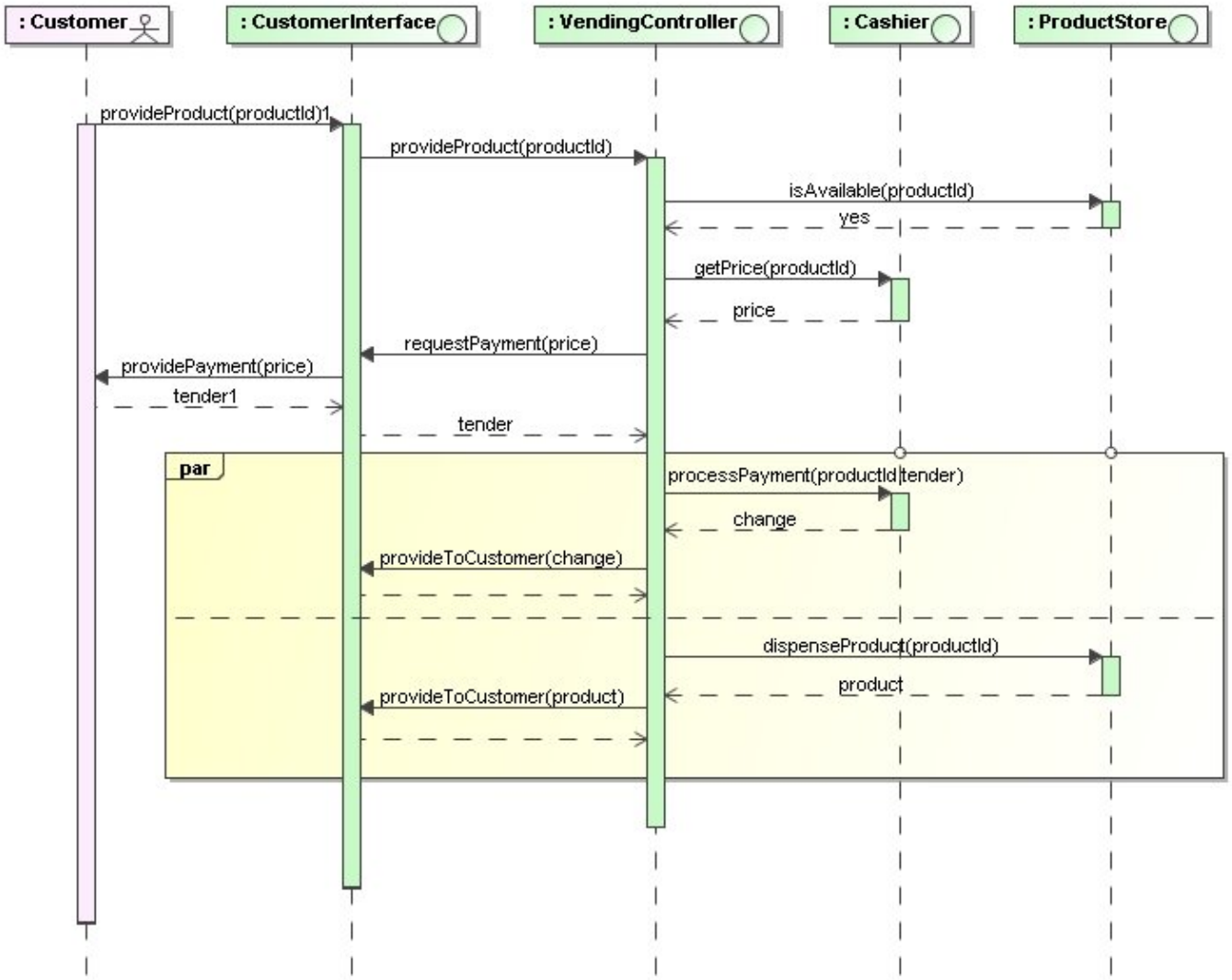

Figure 5.8: Concurrencies shown via a parallel fragment

Chapter 6

Activity diagrams

6.1 Introduction

An activity diagam represents a decomposition of an activity into its components

Activity diagrams provide a very natural and intuitive notation for documenting processes. They are commonly used to document both, system and business processes.

Unlike sequence and communication diagrams, which are particularly suited to documenting individual scenarios, activity diagrams are used to show a process in general.

6.2 Basic activity diagrams

A simple activity diagram shows a sequence of actions performed within some process defined by the activity diagram. For example, Figure **??** shows a simple activity diagram for the process of a sale.

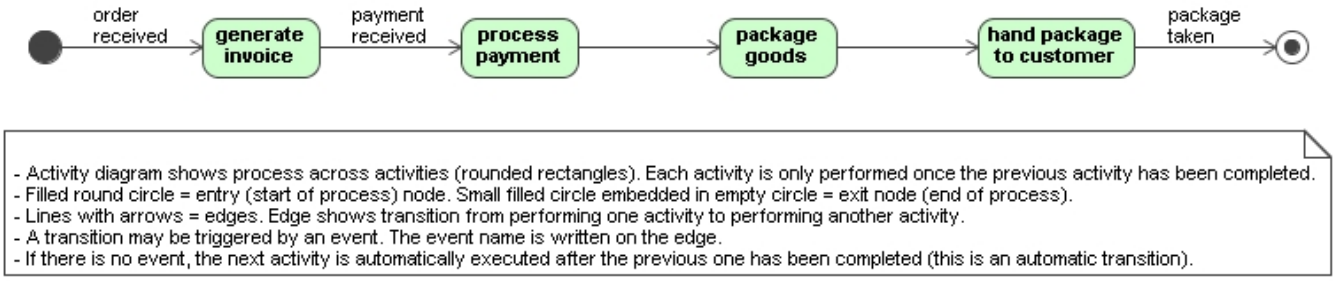

Figure 6.1: Simple activity diagram for the process of a sale

6.2.1 Activity and actions

An activity is an executable behaviour for which there may be a process defined. The process for an activity is shown in an activity diagram. It may be composed of lower level activities executed sequentially or concurrently, decision nodes and ultimately actions.

An action node represents a single processing step within an activity, i.e. a step which is not broken down into lower level work flow steps.

6.2.2 Edges and events

An activity edge represents the transition from one activity to another. The activity the edge points to is only executed once the source activity has completed executing.

6.2.3 Events and automatic transitions

If there is no event on the transition, the transition to the next activity is automatically executed after the previous activity has been completed. This is called an automatic transition.

Alternatively the transition can be triggered by an external event. In such a case the event name is put on the transition.

For example, the transition from the entry node to the *generate invoice* activity is performed when an order is received. On the other hand, the transition from *process payment* to package goods is an automatic transition which is not triggered by an external event. Automatically after having processed the payment, the goods are packaged.

6.2.4 Entry and exit activities

An entry node is shown as a solid circle. It represents the start of the process. An exit node, which represents the end of the process is shown as a filled circle embedded in an empty circle.

The advantage of entry and exit nodes is that one can encapsulate the entire activity and that external activities or states can use the activity without knowledge of the internal process.

6.3 Decision and merge nodes

Activity diagrams provide an intuitive notation for showing decision points in a business or system process and hence for showing multiple scenarios in a single diagram.

A decision node is shown in UML as a diamond. On each of the outgoing edges there must be a condition. The flow will continue along that path for which the condition evaluates to true.

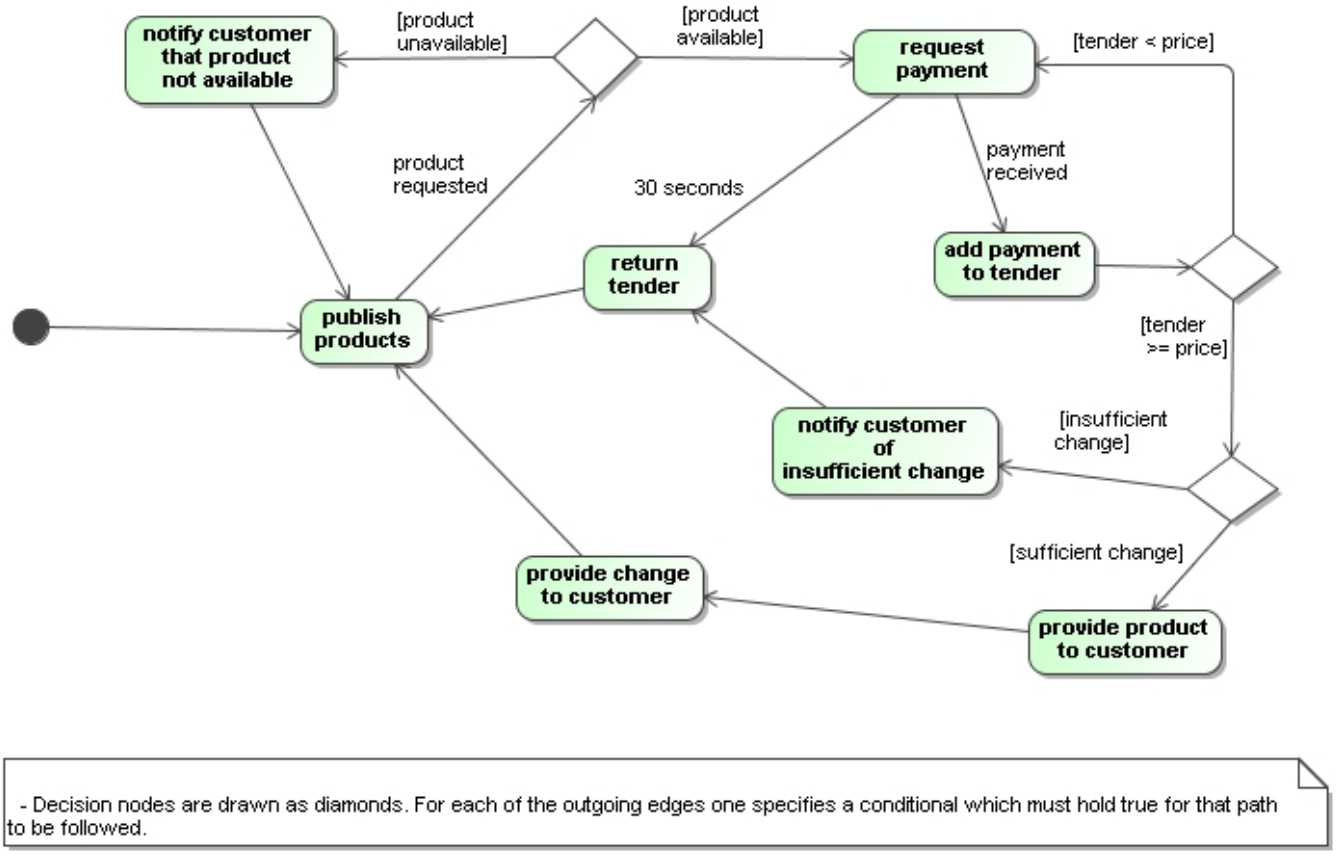

Figure 6.2: A simple activity diagram for the buyProduct use case of a vending outlet

6.3.1 Formulating the conditionals

In order to have a well defined process the conditionals on the outgoing edges must satisfy the following criteria

1. They must be non-overlapping or mutually exclusive -- only one of the conditionals may, for any scenario value to true.
2. The conditionals must cover the full domain in order to always lead to one path being followed. Otherwise the process would get stuck at the decision point.

6.3.2 Merge nodes

At times a process may follow different paths for some work flow steps before merging back into the same core process. To merge alternative paths back into a single flow, one uses a merge node drawn as a diamond with multiple edges leading into it and a single edge leading to the next activity of the main flow.

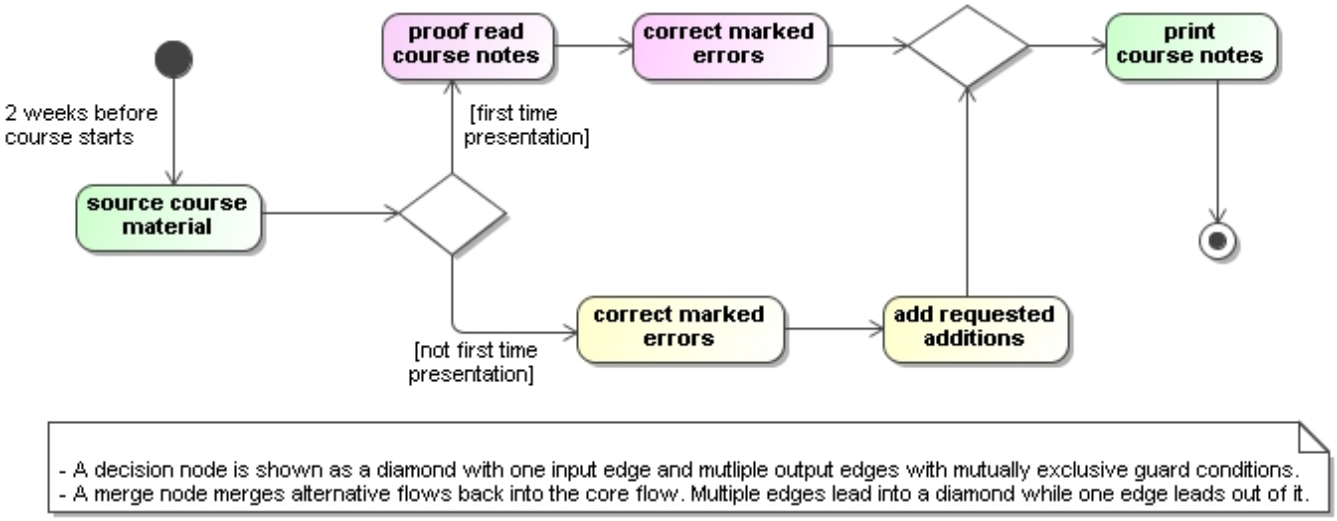

Figure 6.3: Alternative flows for course notes preparation

Figure **??** shows a work flow for the preparation of the course notes one week prior to the start of a course. If the course is presented for the first time, it is sent for proof reading. On the other hand, if the course has been presented before, then all corrections and additions requested by the previous presenter are applied. In either case we then merge the process into a common flow for the printing of the course material.

6.4 Activity partitions (swim lanes)

For both, system and business process modeling, it is often useful to show which object or role player is responsible for which activity. This is particularly important when one wants to show how different role players collaborate to realize a use case.

For example, Figure **??** has one partition for the customer and one for the vendor. It shows how the two parties collaborate to realize a sale. The activities drawn in the customer's partition are done by the customer, while those drawn in the vendor partition are the responsibility of the vendor.

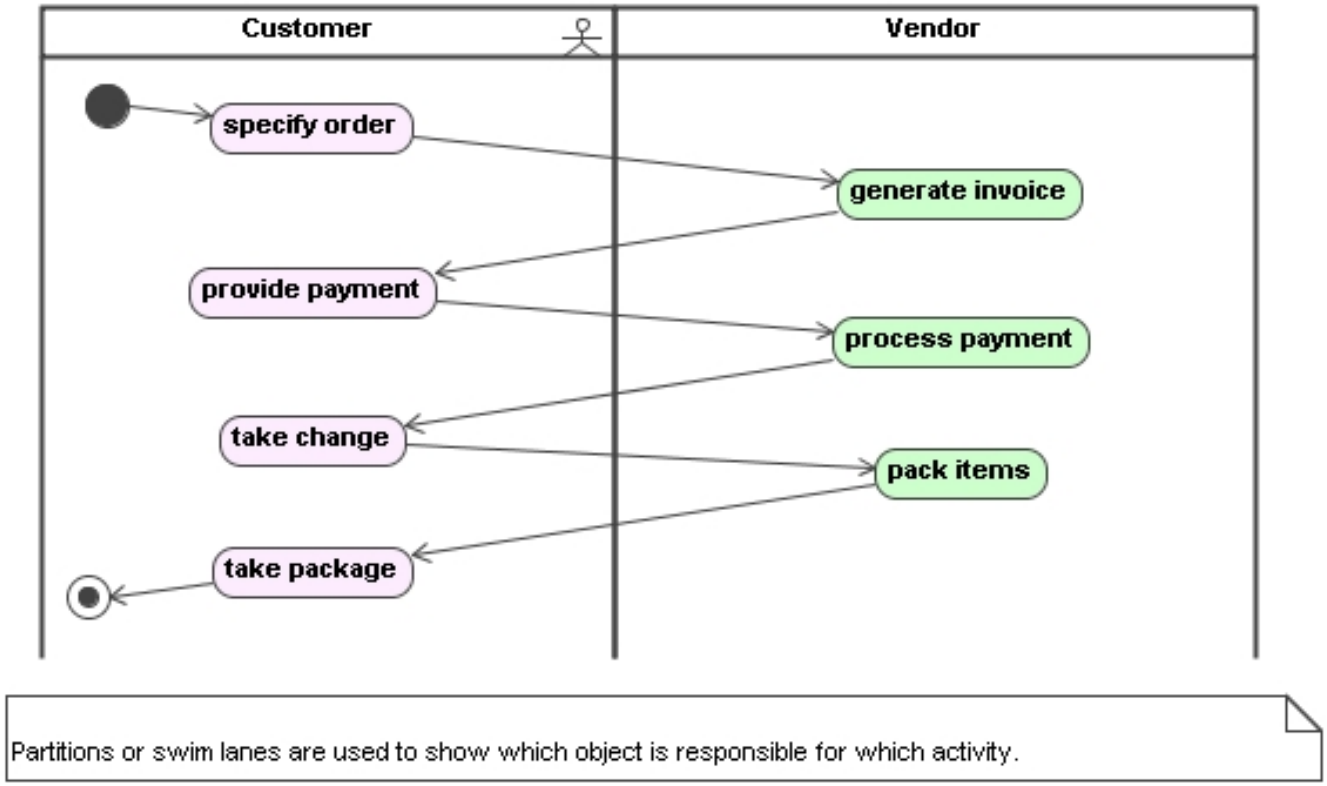

Figure 6.4: Partitioning for a simple sale

6.5 Object flow in activity diagrams

Interaction diagrams provide perhaps the clearest view of the objects exchanged between role players in a process. Activity diagrams can, however, also be used to show object flow within a business or system process. This can be done by inserting an object diagram into the edge representing the transition from one activity to another. The object flows from that role player who is responsible for the previous activity to that role player who is responsible for the next activity.

Figure **??** shows the objects exchanged between the customer and the vendor during the process of a sale.

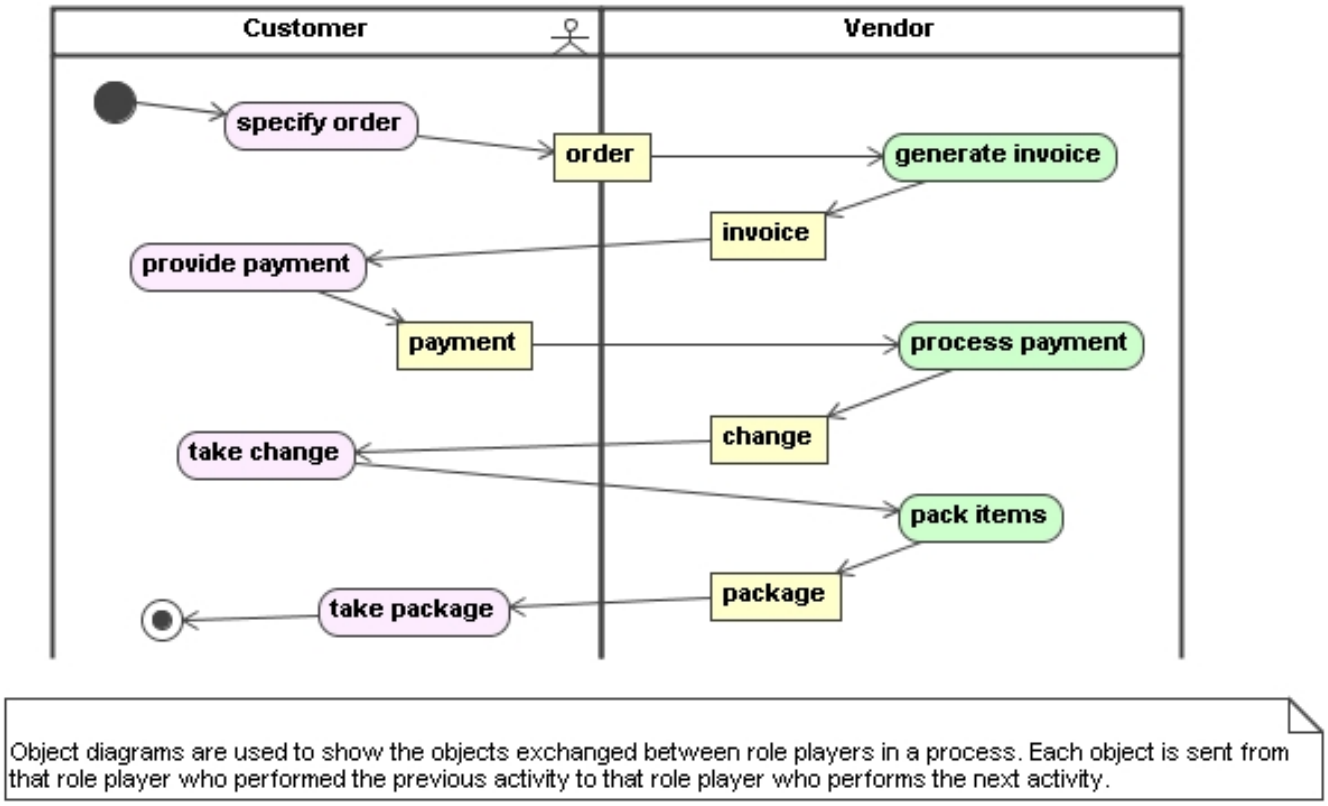

Figure 6.5: Object flow for a simple sale

6.6 Structured and nested activities

Structured or nested activities are used

- to abstract the business process by showing only higher level activities hiding the detailed actions for them, and
- to show common transitions which are available for all nested activities.

A structured activity is shown as a dashed rectangle with rounded corners with the stereotype `<<structured>>`. A transition which is common to the nested activities is shown by drawing an edge from the structured activity to the new activity.

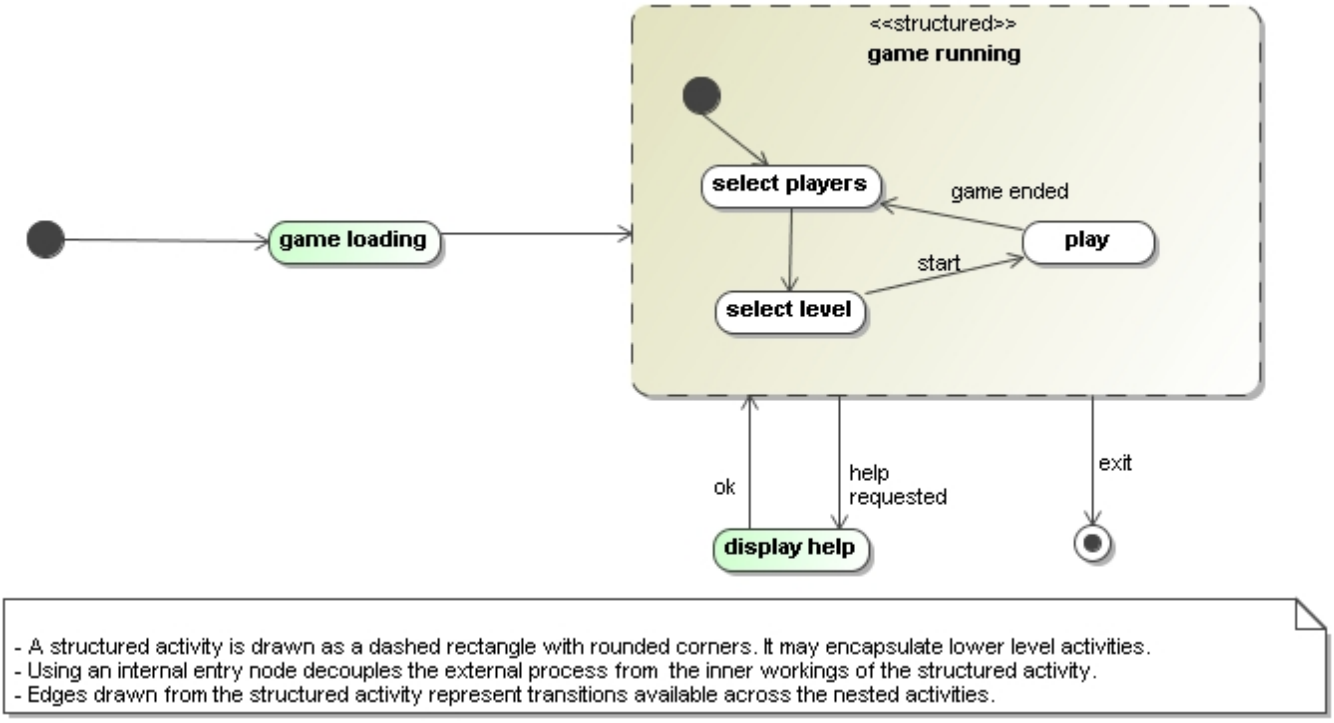

Figure 6.6: Using a structured activity to show common transitions for a game

In Figure **??** one can request help in any of the activities performed while the game is running. Upon the ok event the system is taken back to the activity it came from. Similarly, as your boss enters your office you can immediately exit the game, irrespective of whether you are busy selecting the players, the level or you are busy playing.

Note that we defined an entry node for the internal process. This decouples the external process from the internals of the structured activity.

Structured activities are also common used to show common cancel paths accessible for a subset of the user workflow. For example, in Figure **??** the customer can cancel the purchase while the product has not yet been fully paid for.

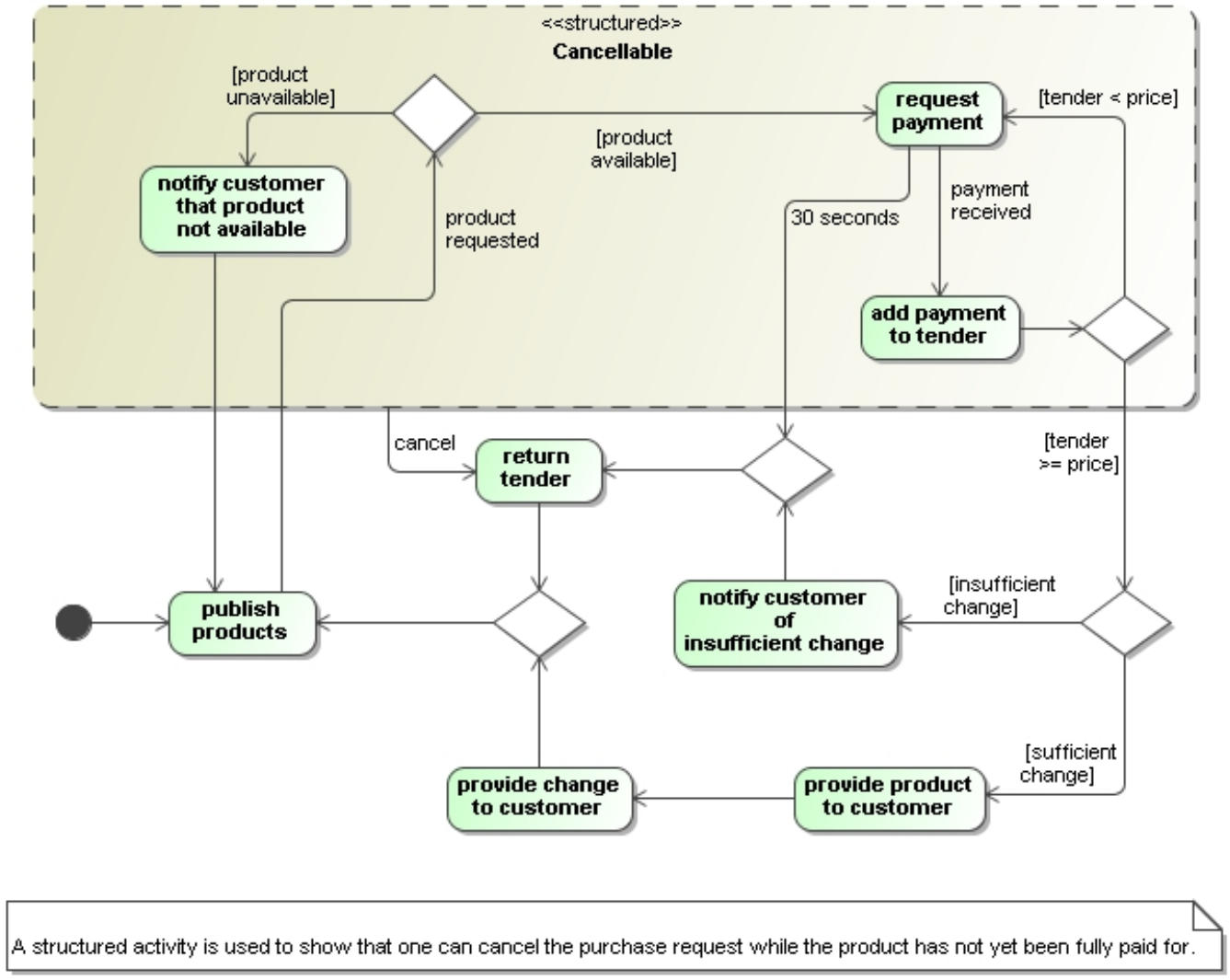

Figure 6.7: Using a structured activity to show transaction cancellation

6.7 Concurrencies in activity diagrams

If certain steps in a process are independent of one another, they can be performed concurrently. Such activities should be shown as concurrent in a implementation neutral process specification. When the implementation mapping is done, the implementors may still choose to implement such work flow steps sequentially. Modeling them as concurrent implies that they could be implemented concurrently.

6.7.1 Forking

Concurrencies are introduced in an activity diagram via forking, i.e. a single thread of activity forks into multiple concurrent threads of activity. A fork is shown as a bar with a single edge leading into the bar and multiple edges leading out of the bar, each edge representing a flow or thread of activity.

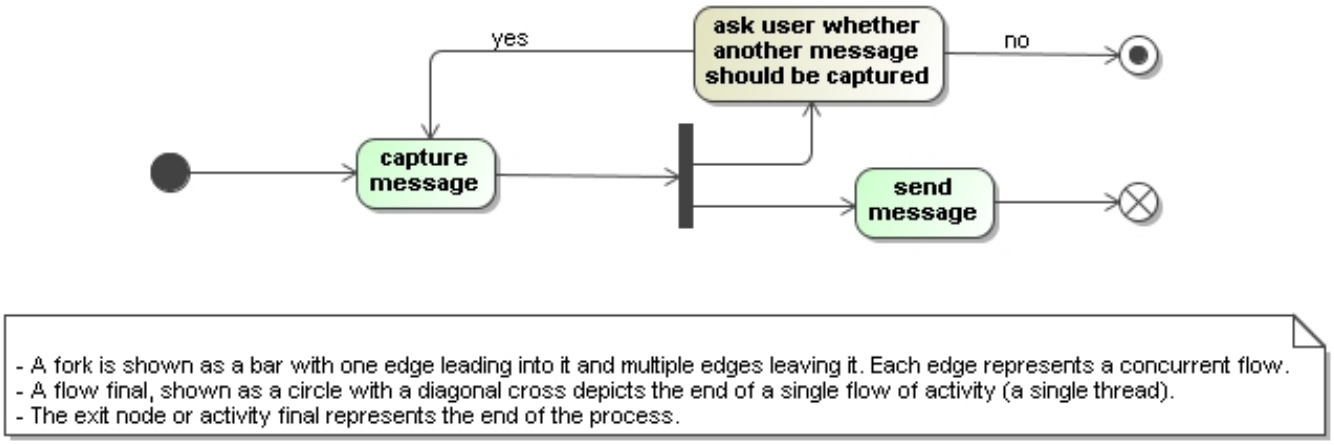

Figure 6.8: Sending messages in a background thread

6.7.2 Flow final node

A flow final node is used to show the end of a flow or thread of activity. It is drawn as a circle with a diagonal cross. The process may still continue as other threads of activity may still be active.

6.7.3 Synchronization

At times one needs to block a business or system process until certain concurrent activities have been completed. This is specified in UML using a synchronization bar. Synchronization bars are similar to forks except that multiple threads of activity lead into the bar while a single thread of activity leads out of the bar.

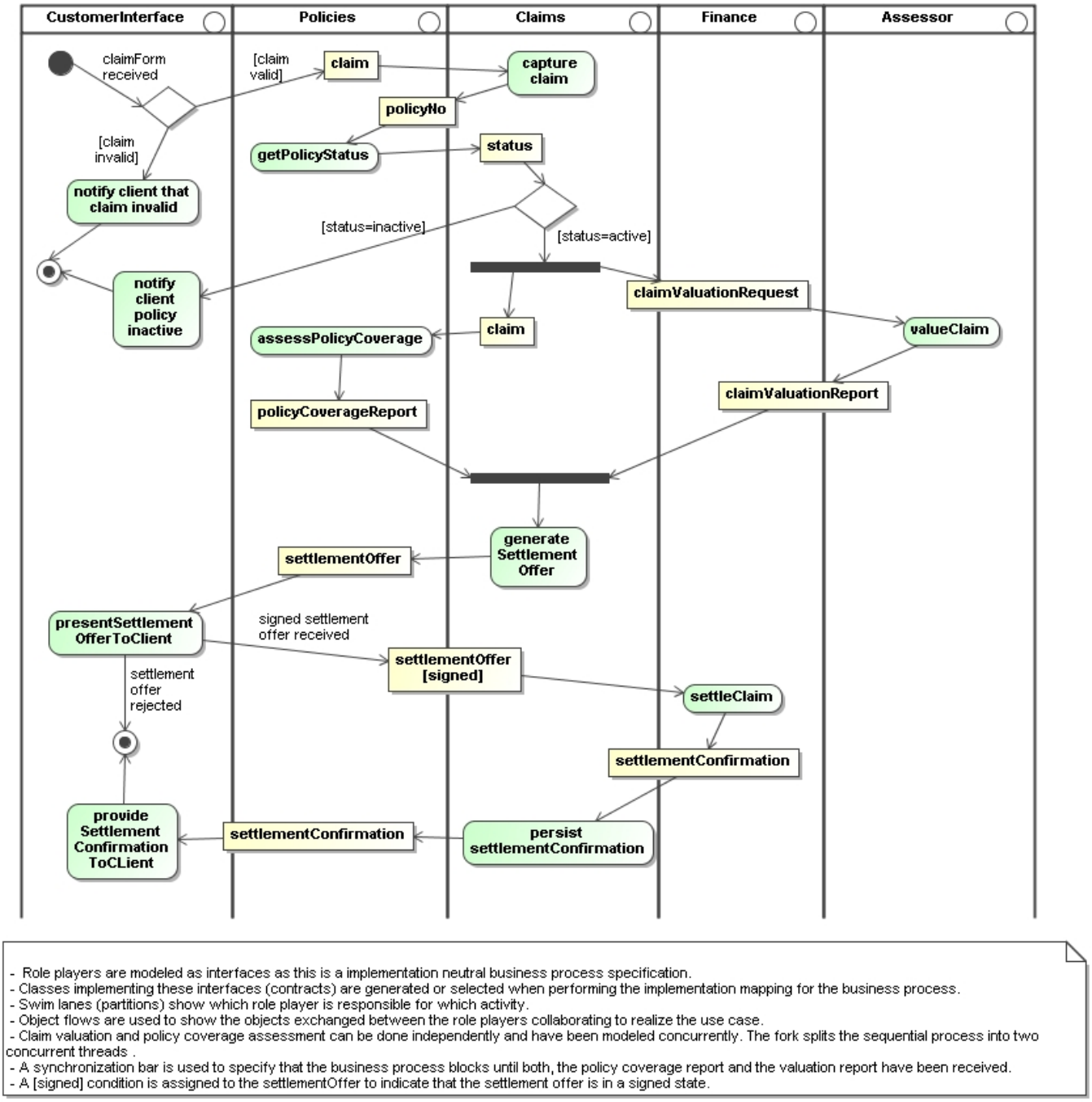

Figure 6.9: An activity diagram for the processing of a claim

Figure **??** uses a synchronization bar to block the business process until both, the policy coverage report and the claim valuation report have been received. Only then does finance go ahead to generate a settlement offer.

6.8 Sending and accepting signals

UML defines special activity types for sending and receiving signals.

Note
A signal is the asynchronous communication of information.

This enables one to specify that at certain points in a work flow one waits for a signal while other activities send signals to external objects.

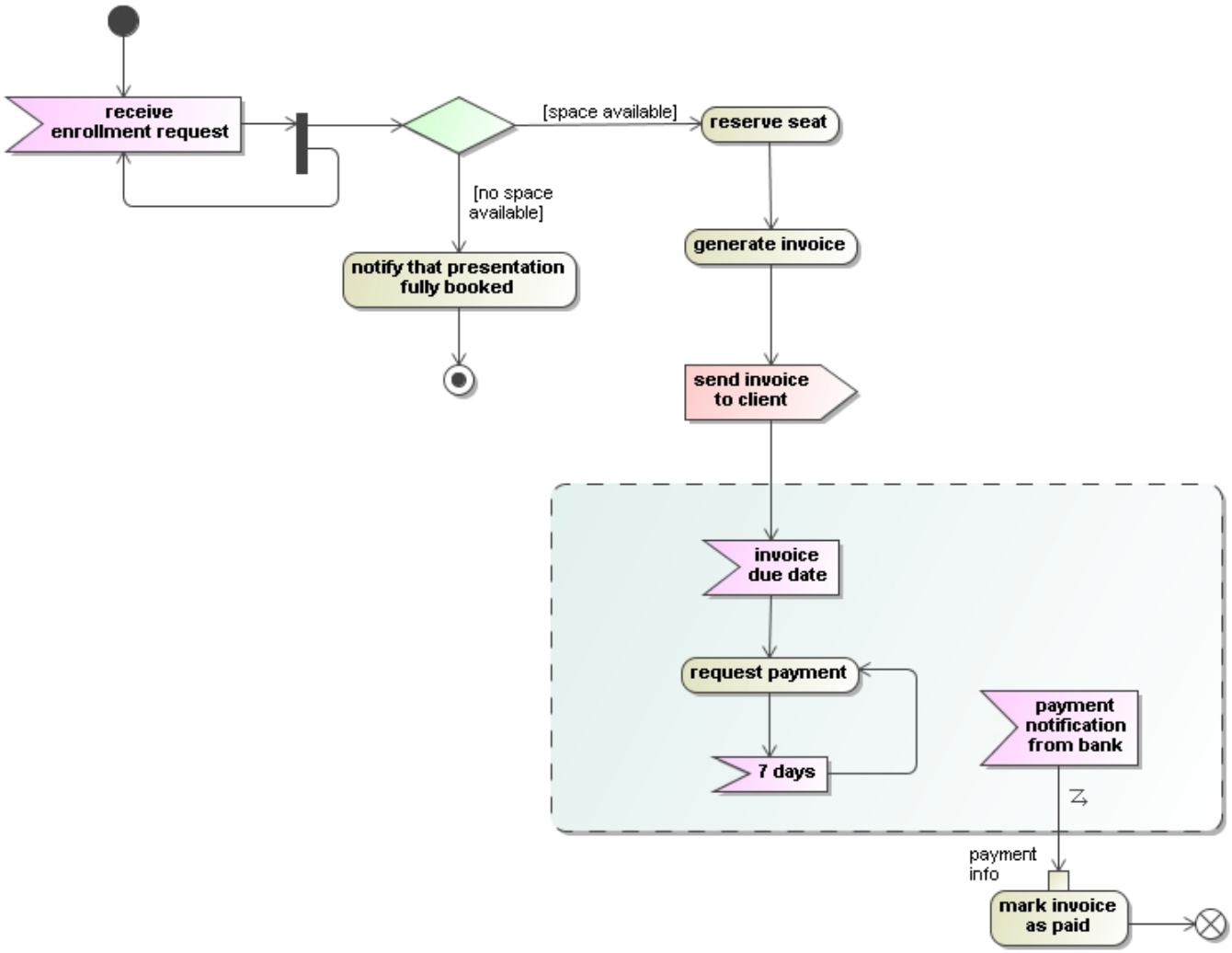

Figure 6.10: Sending and receiving signals within a business process

For example, Figure **??** shows a business process which waits for a enrollment request signal. Having received one, it spawns a work flow processing the enrollment request while waiting for the next enrollment signal.

At some stage in the business process signals are sent to the client, providing the client the invoice and sending payment requests asynchronously to the client.

6.8.1 Interruptible activities

Figure **??** uses an interrupable activity to specify that the process of sending payment requests in regular intervals to the client is interrupted by a payment notification.

6.9 Object pins

Instead of showing the object flows as object diagrams on the transition edge, one can specify the inputs and outputs of an activity more explicitly using object pins. This is done by attaching the object pin to the activity.

For example, Figure **??** shows that the pantry sources the order ingrdients from an order it receives from the waiter and provides the ingredients to the kitchen

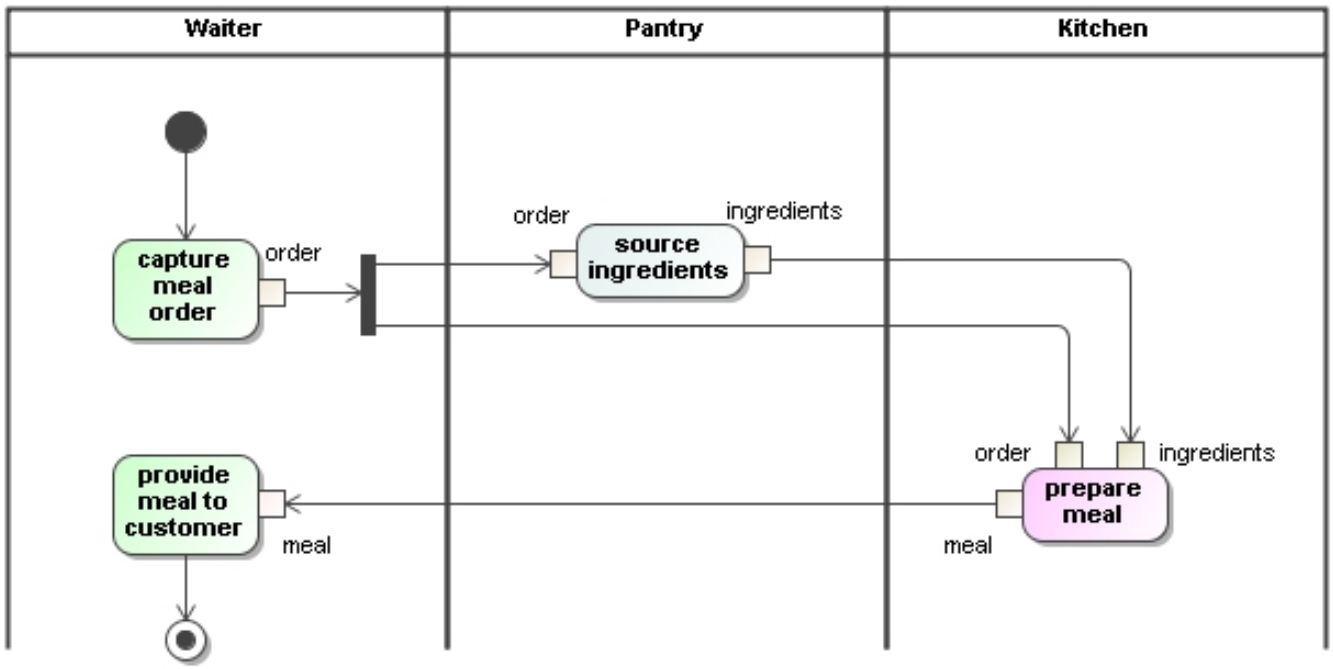

Figure 6.11: Object pins showing the inputs and outputs across the activities for preparing a meal.

An activity may generate multiple outputs or consume multiple inputs. In UML one can use one pin per input or output parameter. For example, the kitchen receives the order from the waiter and the ingredients from the pantry. It only commences the preparation of the meal once all input parameters have been received.

6.10 Expansion regions

Expansion regions are used to specify a set of activities which is done across a number of objects, typically on all the elements of a collection of objects received as input.

The activities per object can be performed

- in parallel (concurrently),
- sequentially (iteratively), or
- streamed (i.e. processing objects extracted from a input stream and feeding the results into an output stream).

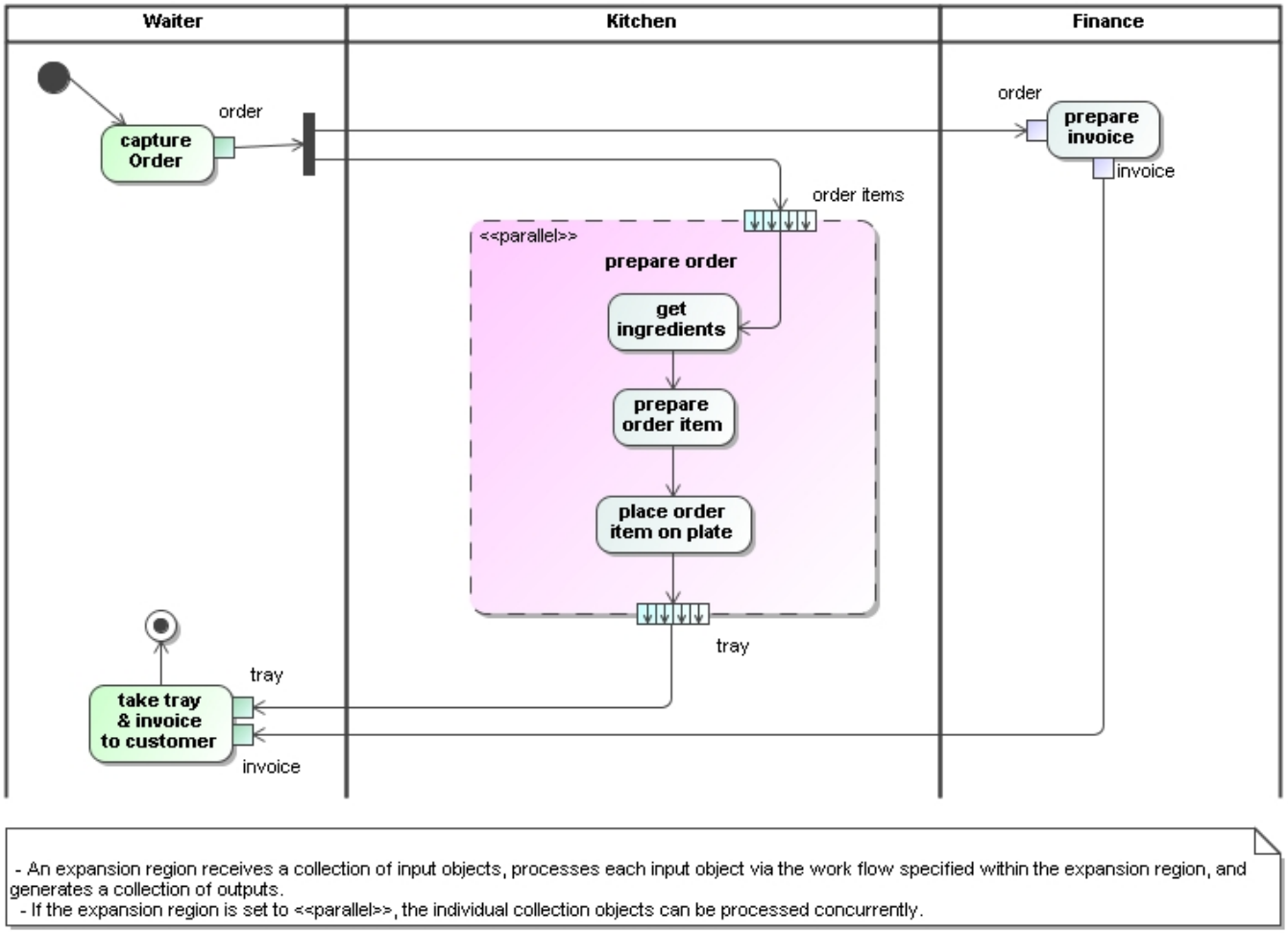

Figure 6.12: An expansion region showing the kitchen preparing the order items concurrently

For example, the kitchen receives a collection of order items. For each order item it sources the ingredients, prepares the order item and places it on a plate.

6.11 Exception and error handlers

Activity diagrams can be used to show the transfer of control to exception and error handlers. In a contracts based approach

- *exceptions* are used by service providers to notify clients that a service they requested is not going to be provided because a particular precondition was not met, while
- *errors* are used to notify the client that a service which was requested is not going to be provided because the service provider encountered a problem which prevents it to satisfy its contractual obligations.

In eiter case, the client may have a process which should be followed in the case where a service provider does not provide the requested service. This is done via exception and error handlers.

In UML the transition to an exception or error handler is shown by a transition which has the stereotype icon for a exception handler transition, that of a zig-zag line with an open error pointing to the exception or error node of an exception handler. The exception or error node is the object node for the exception which is caught. Alternatively the stereotype icon can be used directly, showing the transition itself as a zig-zag line.

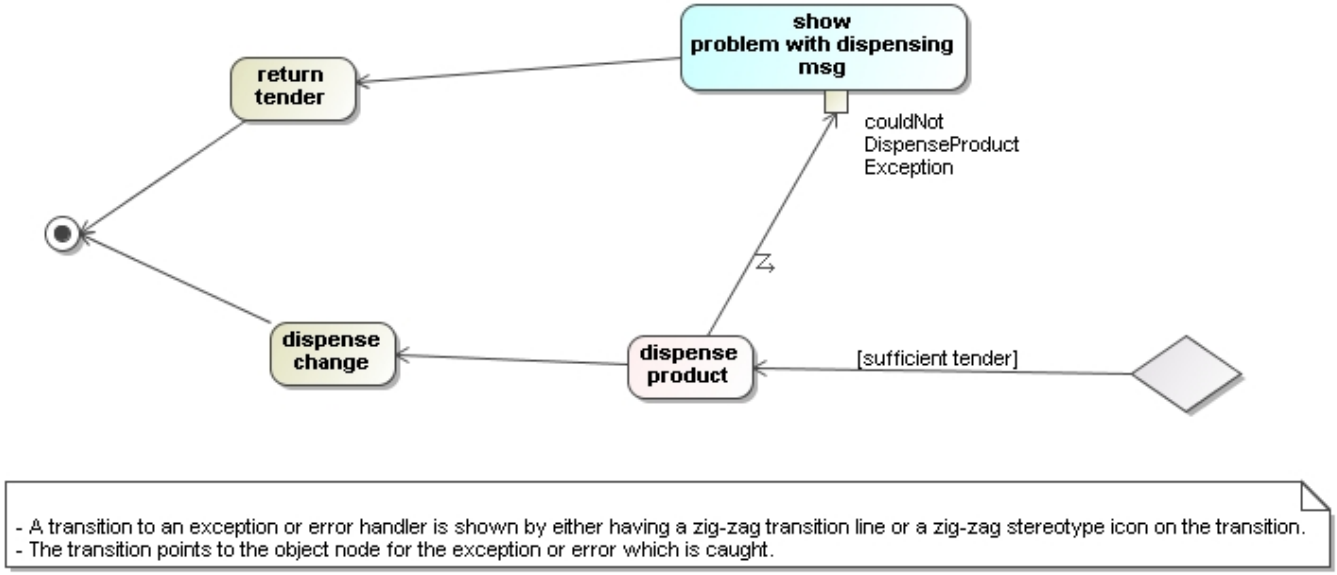

Figure 6.13: Specifying a transition to an error handler

For example, if the product dispenser encounters an internal problem preventing it from realizing the dispense product service, it may notify the client via a suitable error (or exception). Figure **??** shows the error handler which is executed when such an exception is caught.

6.12 Call operations, activity parameters and assigning behaviours/processes to services

UML supports the concept of a call operation resembling the activity of requesting a service from some object. This is particularly useful when showing how a service provider / controller assembles the process realizing the service it provides from lower level services it sources from other service providers. The core benefit is that the activity is directly related to a service providing traceability between the activities and the services they realize.

6.12.1 Call operations

UML supports the concept of an activity which requests a service. The service concrete service provider (class) or more abstract service provider role (interface representing a services contract) responsible for realizing the service is then shown in round brackets. The inputs and outputs of the service are shown as input and output nodes and do correspond to the service signature specification of the class or interface.

Note
Most UML tools will allow you to drag a service from a class or interface onto an activity diagram, generating a corresponding call operation for that service.

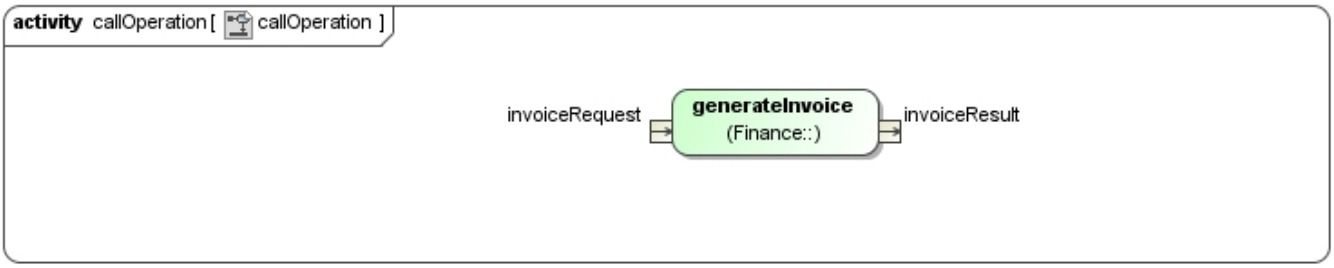

Figure 6.14: Call operation

6.12.2 Assigning an activity/process to a service

Your UML tool will enable you to assign an activity diagram/collaboration to a service, thereby specifying that the service provider uses the process as specified in the collaboration to realize one of the services it offers.

As soon as a process has been assigned as behaviour for a service, the input and output parameter nodes corresponding to the inputs and outputs of the service are shown on the boundary of the structured activity representing the higher level process.

One can now use call operations to show how the service provider assembles the service it offers from services it sources from other service providers. An example is shown in figure

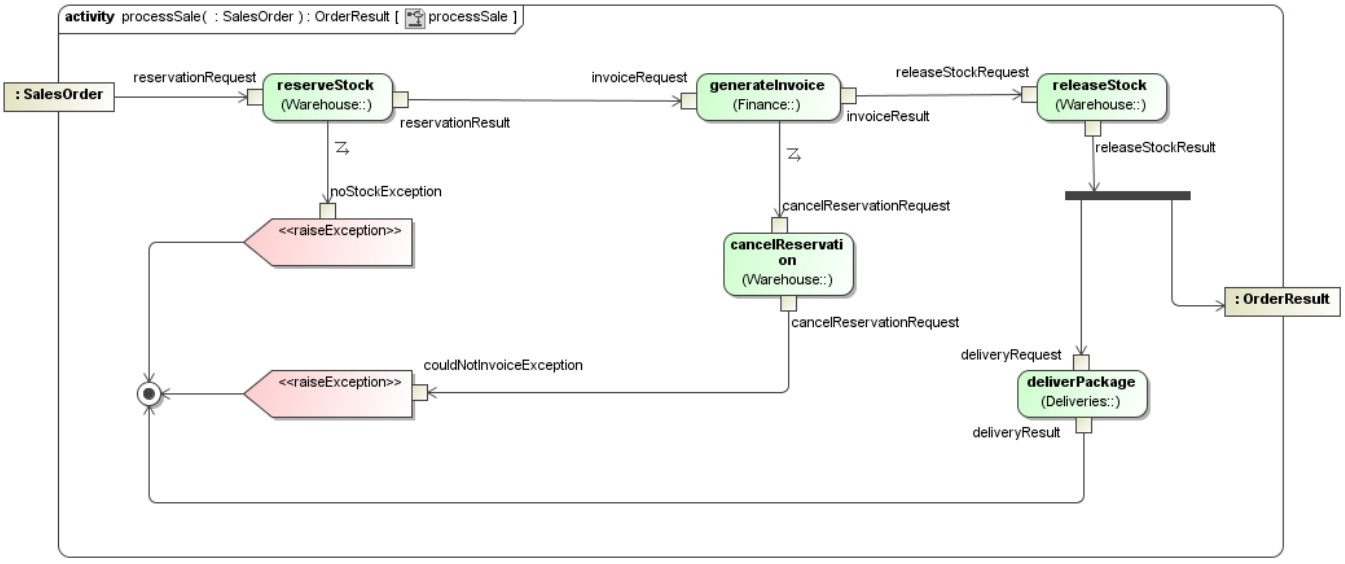

Figure 6.15: The business process for processing a sale

Note
This is an alternative and often preferable notation to swim lanes, showing how a controller assembles a business process from services available to the organization/system.

Chapter 7

URDAD for technology neutral business process design

7.1 Background

7.1.1 Introduction

URDAD, the *Use Case, Responsibility Driven Analysis and Design* methodology [Solms_2007_URDAD, KGK_2007_AFCS], [Solms_2008_GeneratingMdaModelViaUrdad] aims to provide a simple and implementable algorithmic analysis and design methodology which generates a technology neutral design model satisfying accepted requirements for a "good design". The resultant model is meant to represent the Platform Independent Model (PIM) of the *Model Driven Architecture* (MDA) published by the *Object Management Group* (OMG), [Siegel_2001_DIOM, Frankel_2003_MDAA].

URDAD has evolved from the need to

- make design a repeatable engineering process with well defined inputs and outputs,
- make the design simpler by defining a step for step algorithm for the design process,
- enforce accepted requirements for 'good design' through the design methodology itself,
- separate conceptual design from its realization (e.g. the implementation technologies and deployment architecture), i.e. to have a design which survives architecture and technology changes,
- have a design methodology which enables one to manage levels of granularity/refinement effectively, and
- the need to project out services contracts for all service providers required across all levels of granularity.

7.1.1.1 Bibliography

[Frankel_2003_MDAA] David S. Frankel, *Model Driven Architecture*, ISBN 978-0471319207, Applying MDA to enterprise computing, 2003, John Wiley & Sons.

[KGK_2007_AFCS] Riaan Klopper, Stefan Gruner, and Dereck Kourie, *Assessment of a framework to compare software development methodologies*, ACM Press, 226, 2007, 56-65.

[Siegel_2001_DIOM] Jon Siegel, *Developing in OMG's Model-Driven Architecture*, Object Management Group, November 2001.

[Solms_2007_URDAD] Fritz Solms, *Technology Neutral Business Process Design using URDAD*, IOS Press, 161, 2007, 52-70.

[Solms_2008_GeneratingMdaModelViaUrdad] Fritz Solms and Dawid Loubser, *Generating MDA's platform independent model using URDAD*, Elsevier, 22, 2009, 174-185.

7.1.2 Architecture versus design

The architecture of an organization provides the core infrastructure within which the business processes are deployed. It needs to provide a suitable environment within which the organization can realize its vision and mission and its core quality offerings.

Business process design, on the other hand, is concerned with defining the processes through which the services which generate stake holder value are realized. These are driven by specific needs and functional requirements. The processes will specify what needs to be done when and ultimately which organizational components are responsible for which work flow steps.

The business processes deployed within a particular architecture acquire through the architecture the core organizational qualities

- *reliability*, the ability to realize one's obligations to clients and other stake holders reliably,
- *scalability*, the ability to handle large and/or varying volumes of service requests,
- *low cost*, the ability to offer services or products at *low cost*

and potentially a range of other quality requirements.

The architecture of an organization should encompass the full infrastructure into which business processes are to be deployed. This infrastructure which we refer to as the enterprise architecture spans organizational and systems architecture. The business processes are deployed across the entire entireprise architecture.

The enterprise architecture must realize the architectural requirements for the organization. These requirements include

- the scope of services to be deployed within the enterprise architecture,
- the quality requirements which are to be realized across the business processes deployed within the enterprise architecture,
- the integration requirements between the organization and its environment (clients, service providers, and observers),
- as well as any architectural constraints placed by the stake holders on the architecture.

The functional requirements for a user service (use case), on the other hand, include

- the deliverables as required across the stake holders,
- the conditions under which the service may be refused without breaking the contract,
- the desired user work flow
- the messages and objects which need to be exchanged with any other external role players as well as any requirements requirements around them (e.g. data structure requirements), and
- any activities required by any of the stake holders.

While the functional requirements are realized through work flow, the architectural requirements are realized through infrastructure.

7.1.3 URDAD in the context of OMG's Model Driven Architecture (MDA)

OMG's MDA forms the bases for most model driven development (MDD) processes [Selic_2003_TPMD, Schmidt_2006_MDE]. A high level view of a model driven approach is shown in Figure **??**}.

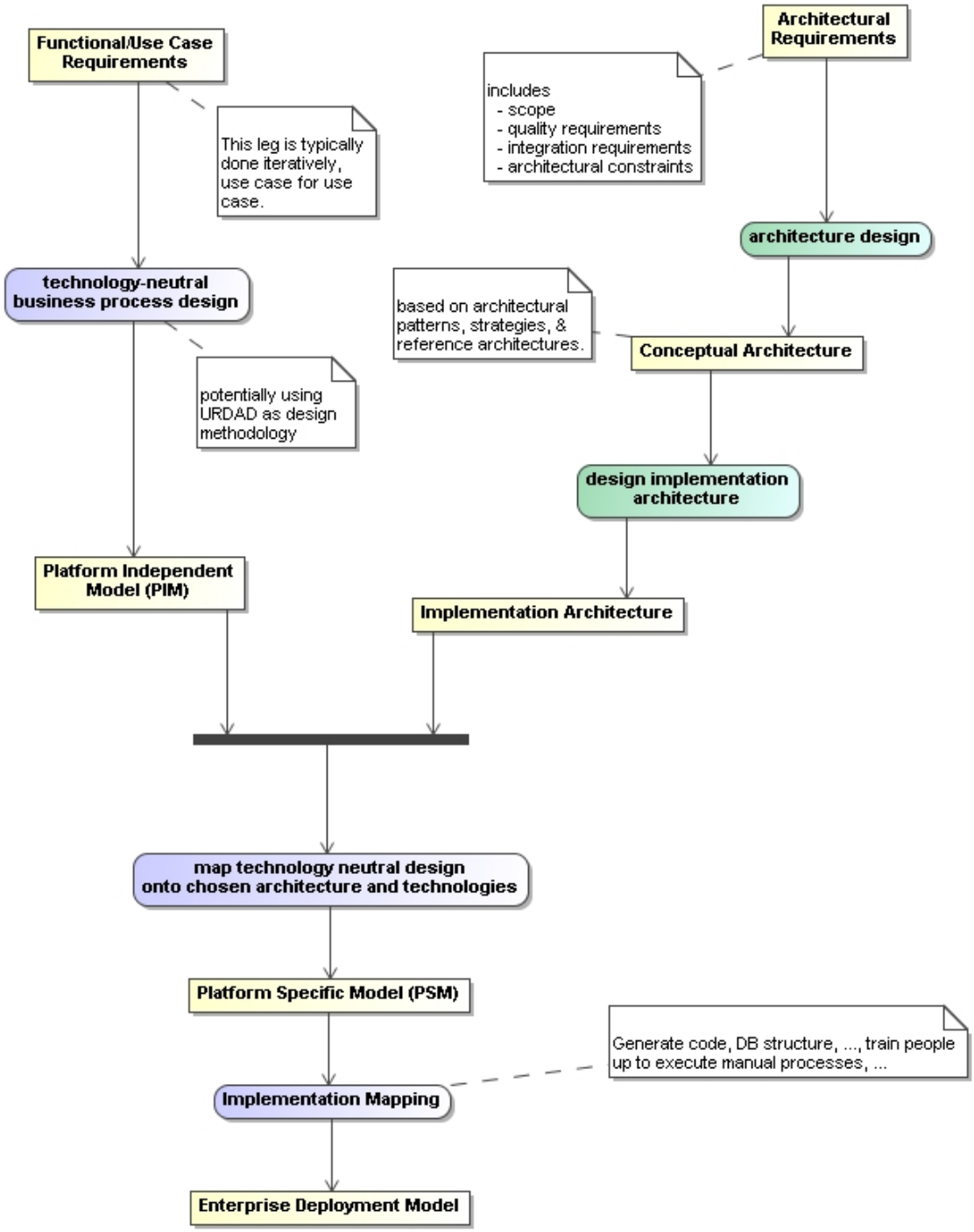

Figure 7.1: High level view of URDAD in the context of a model driven approach.

The input for the technology neutral business process design are the functional or use-case requirements. The output of the design phase is the Platform Independent Model (PIM) which can be mapped onto one's choice of implementation architecture and technologies resulting in a Platform Specific Model (PSM). Both, the PIM and the PSM are UML models. The Platform specific model is then taken through an implementation mapping which includes the generation of all deployable artifacts including the code, the database structures, the deployment scripts, the user documentation. MDA effectively separates design from architecture.

URDAD targets

- the analysis phase resulting in a use case contract, as well as
- the design phase resulting in a technology neutral business process design.

The use case contract contains both, functional and non-functional requirements around a use case. The functional requirements drive the design process while the non-functional and in particular the quality requirements like scaleability, reliability, integrability, ... requirements drive the architectural process. The output of the architectural process is an infrastructure into which the business processes are to be deployed. This may span, both, organizational and systems architecture as business processes will often be realized across manual and automated processes [1]

7.1.3.1 Bibliography

[Schmidt_2006_MDE] Douglas C. Schmidt, *Model Driven Engineering*, 39, February 2006, 25-31.

[Selic_2003_TPMD] Bran Selic, *The Pragmatics of Model Driven Development*, 20, September/October 2003, 19-25.

7.1.4 URDAD in the context of OMG's Model Driven Architecture (MDA)

URDAD is usually embedded within an iterative realization or development process. A typical model driven development process is shown in Figure **??**. Note that the technology neutral business process design is performed by business analysis. The technical team comprising both, architecture and implementation (development), is responsible for the realization of the business process within the chosen architecture and technologies.

[1] The implementation mapping around manual business process steps would typically involve training of workers.

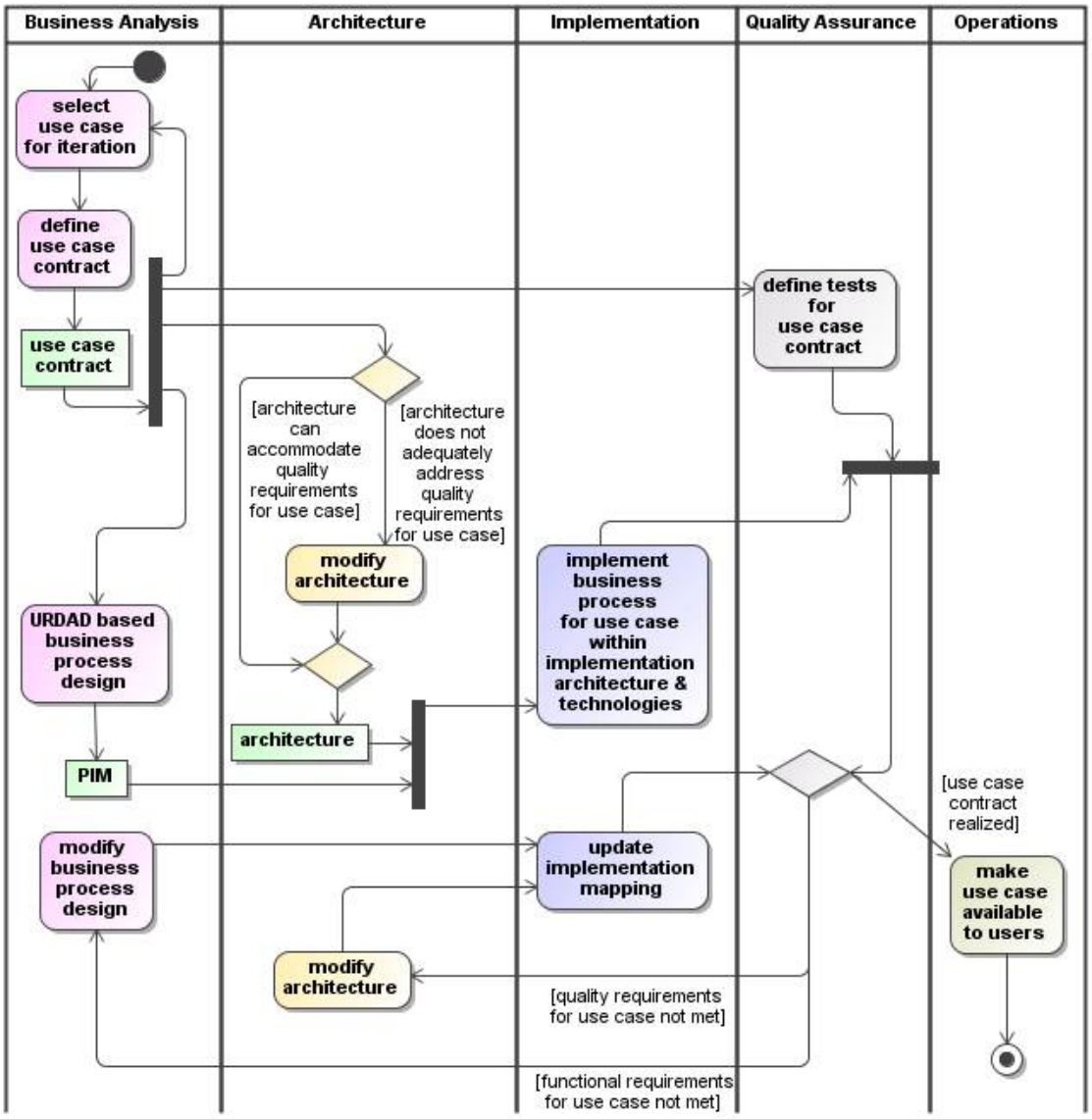

Figure 7.2: High level view of URDAD in the context of a model driven approach.

After passing quality assurance and actual deployment, operations takes over the management of the business process execution.

7.1.5 Architecture versus design

URDAD has grown out of Responsibility Driven Design (RDD) methodology pioneered by Rebecca Wirfs-Brock and Brian Wilkerson (see [WW_1989_OOD, WM_2002_OD, WirfsBrock_2007_TDS]). . Like RDD, URDAD focuses during the early stages of the design on identifying and assigning responsibilities. Also, like RDD, URDAD puts a lot of emphasis on client-server contracts. URDAD adds a step-for-step algorithm which generates different layers of granularity, enforces decoupling within each level of granularity via work flow controllers and enforces, through the methodology, a number of widely accepted design principles.

The *ICONIX* process from Doug Rosenberg discussed in [RD_1999_UDOM] provides a structured process for evolving the static model from the collaboration requirements, but are not really responsibility driven, nor do they project out clean layers of granularity.

Methodologies like the Rational Unified Process [**?**] and Extreme Programming are incorporate aspects of a design methodology, but are, in many respects more software development than design methodologies.

7.1.5.1 Bibliography

[Kruchten_2000_RUP] Philip Kruchten, *The Rational Unified Process*, An Introduction, 2000, Addision-Wesley Professional.

[RD_1999_UDOM] Doug Rosenberg and Kendall Scott, *Use case driven object modeling with UML*, A practical approach, 1999, Addision-Wesley Professional.

[WM_2002_OD] Rebecca J. Wirfs-Brock and Alan McKean, *Object design*, Roles, responsibilities and collaboration, 2002, Addision-Wesley Professional.

[WW_1989_OOD] Rebecca J. Wirfs-Brock and Brian Wilkinson, *Object-oriented design*, A responsibility-driven approach, October 1989, John Wiley & Sons, 71-75.

[WirfsBrock_2007_TDS] Rebecca J. Wirfs-Brock, *Toward Design Simplicity*, 24, March/April 2007, 9-11.

7.2 Requirements for an analysis and design methodology

7.2.1 Introduction

The design methodology should satisfy general requirements placed on a methodology. Here we can be guided by Edward Berard who lists some general requirements for a methodology.

In addition, the design metodology should lead one to a ":good design" To this end we need to obtain an understanding of the criteria one would use to assess two various design solutions in order to determine "the best solution". For this we will look at widely accepted design principles.

7.2.2 Berard's requirements for methodologies in general

Berard has formulated some general requirments for a methodology [Berard_1995_WIAM]. These are useful for assessing the usefulness of a methodology in general.

In particular, Berard states that one should require of any methodology that it

- can be used repeatedly, each time achieving similar results,
- can be taught to others within a reasonable time frame,
- can be applied by others with a reasonable level of success,
- is applicable to a relatively large percentage of cases, and
- is able, on average, to achieve better results than either other techniques, or an ad hoc approach.

7.2.2.1 Bibliography

[Berard_1995_WIAM] Edward V. Berard, *What is a methodoloy?*, 67-69, 1995, Information Technology Management Web.

7.2.3 Stake holder requirements for the technology neutral (business) model

We take the approach that there is no absolute quality, but that quality is a measure of the extend to which the stake holder's quality requirements are fulfilled. Thus, before we can assess the quality of a (business) model, we need to identify the stake holders who have an interest in the model and then we need to elicit their quality requirements for the model.

The stake holders who have an interest in the analysis and technology neutral (business) model include

- *Strategic management* which is responsible for evolving the organization's or systems vision and mission and the the business strategies through which these are realized.

- *Business Analysis* which is responsible for performing the stake holder requirements analysis and the technology neutral (business) process design itself.
- *Architecture* which is responsible for designing a suitable infrastructure hosting the functionality (business processes) in such a way that it enables the organization/system to realize its vision and mission.
- *Implementation* which is responsible for for performing the implementation of the designed business processes. This includes developers who develop the automated implementation mapping as well as managers who train their staff to perform certain business process steps manually.
- *Quality assurance* which is responsible for assessing the extend to which the model realizes the stake holder's functional and non-functional requirements and reporting any defects.
- *Operations* which is responsible for executing / overseeing the execution of the business processes and ensuring that the services are rendered to the user's satisfaction.

	Stake holders					
Quality requirement	**Business, Strategic managment**	**Business Analysis**	**Architecture**	**Implementation**	**Quality assurance**	**Operations**
Completeness		x	x	x	x	x
Consistency		x	x	x	x	x
Simplicity, understandability		x		x	x	x
Modifiability	x	x				
Cohesion		x		x		
Testability		x		x	x	
Traceability		x		x		
Reusability		x		x		
Technology neutral	x	x	x			

Table 7.1: Stake holders in the technology neutral (business) model and their quality requirements

7.2.4 Accepted design principles

Robert C. Martin has compiled a widely quoted list of accepted design principles [Martin_2002_ASD]. Many of these design principles including the interface segregation, dependency injection and the Liskov substitution principles are realized by following a strict contracts based approach. One of the most important principles is the single responsibility principle which requires that at any level of granularity, each contract or class should address only a single responsibility. All the services should be narrowly aligned with the this responsibility focus. The reuse/release equivalence principle can be addressed by enforcing that one only reuses components which are released with realizing a published contract. If one would like to enforce the open-closed principle, one would do so separately from a design methodology.

Other widely accepted design principles include

- the simplicity principle, [WirfsBrock_2007_TDS],
- realizing a high level of reusability [LSW_1987_SRTB],
- enforcing clean layers of granularity, [Martin_2002_ASD, **?**],
- testability across the levels of granularity [VM_2006_STNV], and
- bidirectional traceability across layers of granularity [Dick_2005_DT, ANRY_2006_MT].

7.2.4.1 Bibliography

[ANRY_2006_MT] Netta Aizenbud-Reshef, Brian T. Nolan, Julia Rubin, and Yael Shaham-Garifi, *Model traceability*, 45, 2006, 515-526.

[Dick_2005_DT] Jeremy Dick, *Design traceability*, 22, November 2005, 14-16.

[LSW_1987_SRTB] Manfred Lenz, Hans A. Schmid, and Peter F. Wolf, *Software Reuse Through Building-Blocks*, 4, 1987, 32-42.

[LSW_1987_SRTB] David J.N. Artus, *SOA realization*, Service design principles, February 2006.

[Martin_2002_ASD] Robert C. Martin, *Agile software development*, ISBN 978-0471319207, Principles, Patterns, and Practices, 2002, Prentice-Hall.

[Martin_2002_ASD] Robert C. Martin, *Agile software development*, ISBN 978-0471319207, Principles, Patterns, and Practices, 2002, Prentice-Hall.

[VM_2006_STNV] Jeffrey M. Voas and Keith W. Miller, *Software testability*, The new verification, 12, May 1995, 17-28.

[WirfsBrock_2007_TDS] Rebecca J. Wirfs-Brock, *Toward Design Simplicity*, 24, March/April 2007, 9-11.

7.3 Analysis and design principles supporting design qualities

Using guidelines provided for methodologies in general as well as the widely accepted design principles, we have identified a list of design activities which assist in enforcing these design principles. Figure Table **??** shows the final list of attributes we would want to realize within a design generated by the design methodology.

7.3.1 Enforce the single responsibility principle

The single responsibility principle and thus a high level of cohesion is enforced by grouping functional requirements into responsibility domains and assigning each responsibility domain to a separate services contract. Any system or organizational component as well as any external service provider realizing the full contract would be pluggable. Enforcing the single responsibility principle directly drives pluggability and reusability.

In addition, it makes each object more understandable through being able to understand the contract(s) it realizes without having to understand the way in which the services specified in the contract are implemented.

Finally, enforcing the single responsibility principle also facilitates simpler maintainability as

- maintenance is often required around a particular responsibility (enforcing the single responsibility principle leads to localized maintenance), and
- one can verify whether, after maintenance work, the contractual obligations are still met.

7.3.2 Fix the levels of granularity

In the context of a work break down structure, one is automatically led to define different levels of granularity [DM_1979_SASS]. In order to generate clean layers of granularity, URDAD starts by identifying the first level responsibilities and assigns them to contracts. The business process and ultimately the service provider contracts are specified for this level of granularity before going, in a structured way, to the next lower level of granularity.

This improves the maintainability of the design as changes to a business process often only need to be applied to the controller of a particular level of granularity.

Fixing the levels of granularity also improves the understandability and usability. It facilitates incremental understanding of a design, enabling one to look at a high level business process before specifying how each of the individual high level work flow steps are realized through lower level business processes. Furthermore, a particular role player often only needs to work at a specific level of granularity without there being a need to understand either the higher or lower levels of granularity.

	Design qualities								
	Complete	Consistent	Simple, understandable	Modifiable	Cohesive	Testable	Traceable	High reusability	Technology neutral
Single responsibility principle			x	x	x	x		x	
Clean layers of granularity			x	x		x	x	x	
Decouple via contracts			x	x		x		x	x
Controller assemble process across service providers		x	x	x	x				x
Connect layers of granularity	x				x		x		
Structure from process	x	x	x		x		x		
Well defined PIM elements	x				x				
Diagrams as views onto model		x	x		x		x		
Standard modeling language			x						x

Table 7.2: Analysis and design principles supporting model qualities

7.3.3 Decoupling via services contracts

For each responsibility domain one assigns a separate services contract. The business process is designed to be realized across abstract service providers realizing these contracts. Design by contract rules are enforced ensuring the pluggability of service providers realizing the contract as well as the pluggability of specializations. This results in a loosely coupled design.

Enforcing that service providers realize services contracts increases the reusability of such service providers as the client can compare the services requirements with what is guaranteed through the contract.

Furthermore, enforcing contracts facilitates testability. It is difficult to write a sensible test if one does not know what the contractual obligations are which need to be tested.

A contracts based also approach improves maintainability and extensibility through enhanced pluggability and testability.

If all participants in a business process lock into contracts, the individual contracts can be realized within different technologies. A contract driven approach can be used to generate a technology neutral design.

7.3.4 Define for each level of granularity and each responsibility domain a controller

The controller is that object which takes ownership of the service and which controls the business process realizing the service. Introducing for each level of granularity and each responsibility domain a controller localizes the business process information within the controller and decouples the service providers from that level of granularity. Taking any business process decisions out of individual service providers and localizing it within a controller results in simpler business process management and maintenance [2] Furthermore, the increased decoupling leads to a higher level of reusability.

The introduction of a controller for each level of granularity also simplifies the design and improves understandability as one only needs to look at the controller logic to understand the business process for the current level of granularity.

7.3.5 Derive structure from process

Going from a use case directly to defining structure is difficult and often leads to complexity which may not be required. A simpler approach which leeds to reduced complexity is that of defining first the process through which the use case is realized at a particular level of granularity. One can then project out the minimal structure required to support the process.

Furthermore, driving structure from process facilitates the traceability of any structural element to the process it supports and across the layers of granularity to the use cases for which it is required. Similarly, one can trace from a use case to the structural elements required across the levels of granularity to realize the use case.

7.3.6 Document relationships between layers of granularity

Finally, documenting the relationships between the layers of granularity is required to support full bidirectional traceability across the layers of granularity.

7.3.7 Business benefits of desired design attributes

Ensuring that the technology neutral business process design has the desired design attributes results in some direct business benefits.

In particular, maintainability/extensibility, simplicity and a high level of reusability all contribute to reduced cost and reduced time to market. The design being technology neutral, simplicity and maintainability/extensibility all result in increased business flexibility. Reliability is increased through simplicity and testability. Finally, design simplicity and the design being technology neutral enables business itself to take control and ownership around the business processes - i.e. instead of technology being in control of the business processes.

[2] This strategy is also directly used within Services Oriented Architectures.

7.3.8 Bibliography

[DM_1979_SASS] Tom DeMarco, *Structured analysis and system specification*, 1979, Yourdon Press.

7.4 URDAD - the methodology

7.4.1 Introduction

URDAD provides a use case driven algorithmic analysis and design methodology generating a technology neutral business process design. One thus starts with a concrete user service for which the business process is to be designed.

The methodology enforces those design activities which have been identified as drivers for the desired design attributes and the resultant business benefits.

One of the core aspects of URDAD is the consistent management of level of granularity. Each level of granularity has both, an analysis and a design phase. This ensures that one does not have to identify all requirements across all levels of granularity up-front.

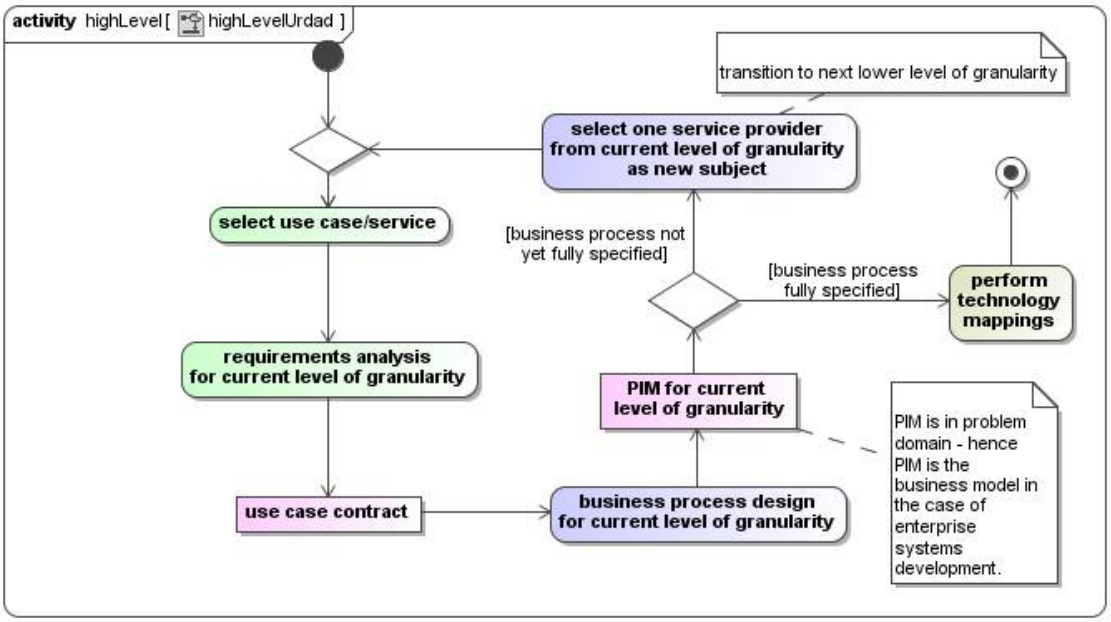

Figure 7.3: High-level view of the URDAD methodology

The methodology starts with the initial analysis phase followed by a design phase for the current level of granularity. This will project out the high level service providers required to realize the use case. One of these is selected as the subject for the next lower level of granularity. One then repeats the process for each of the lower level services (use cases) required from that services provider. One thus performs the requirements analysis followed by the business process design for each of these lower level services.

Figure **??** shows the URDAD analysis and design methodology in more detail.

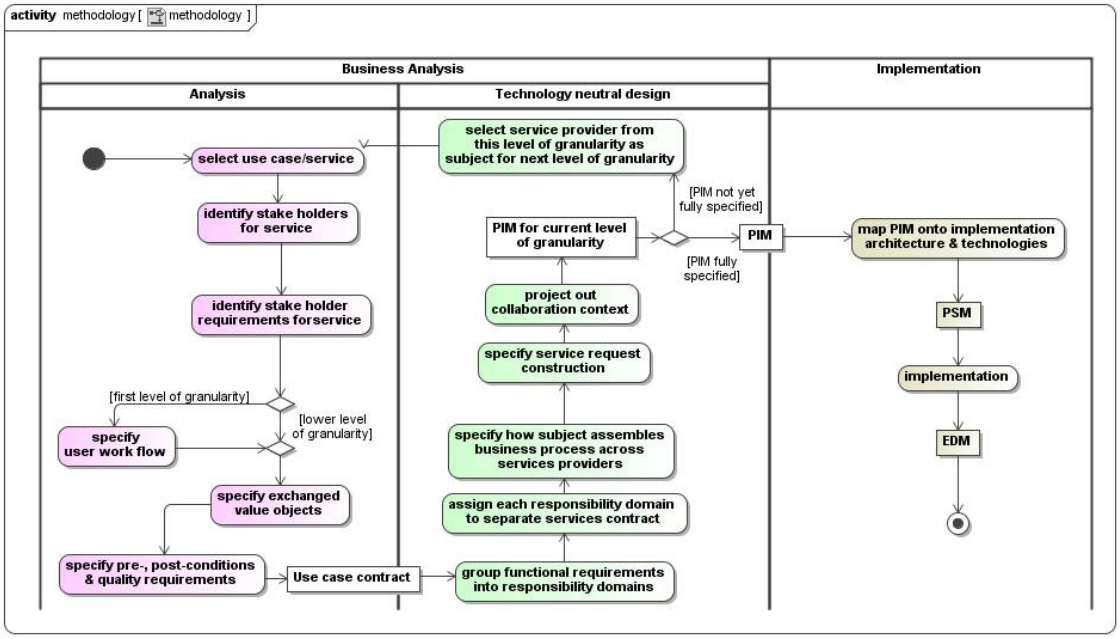

Figure 7.4: More detailed outline of the URDAD methodology

We use the example of processing an insurance claim to illustrate the algorithm. This example is taken through two levels of granularity in order to illustrate the incremental refinement of the technology neutral business process design.

7.4.2 The analysis phase

7.4.2.1 Introduction

The analysis phase aims to elicit, verify and document the stake holder requirements. As one takes the business process design through lower levels of granularity, one revisits the analysis phase to elicit the lower level requirements around the individual workflow steps.

Often the high level analysis around the core business process and the lower level analysis around a functional requirement from a specific responsibility domain is done by different business analysts who focus on different business areas.

For example, a business analyst from the claims department would analyze the requirements and design the high level business process for for processing a claim. The lower level requirements and business processes around how, for example, the claim is to be paid out or how the claim is to be valued could be performed by business analysts from the finance and valuations department of the organization.

7.4.2.2 Functional requirements

During the analysis phase one first identifies all those stake holders who have an interest in the use case. Stake holders are those objects who place requirements around a use case. Only once one has identified all the stake holders in a use case can one hope to elicit all functional requirements for that use case. Maintaining the linkage of any functional requirement to the stake holder who requires it facilitates full traceability of any business or system activity back to the stake holder requirements they realize and ultimately to the stake holder itself.

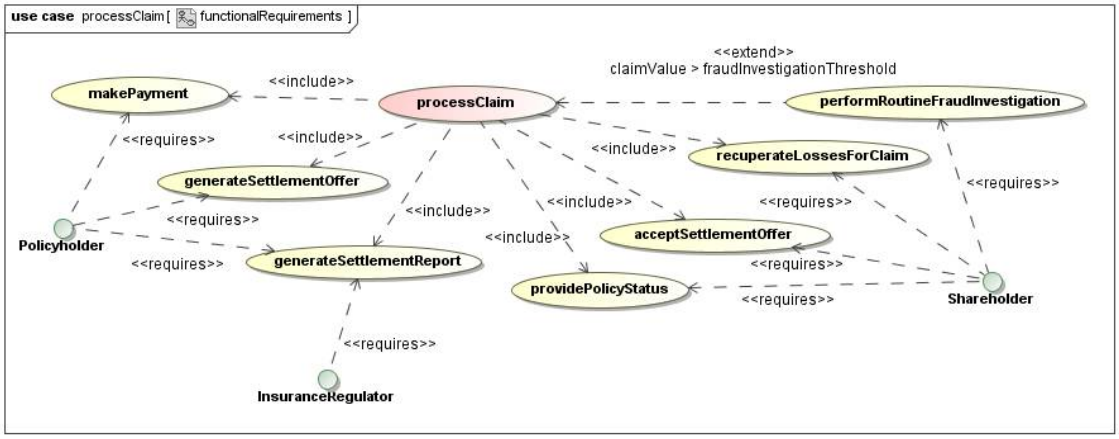

Figure 7.5: Functional requirements for the process claim use case.

Figure **??** shows the high level functional requirements for a process claim use case. Only the first level functional requirements should be included at this level of granularity. The detailed lower level functional requirements around the higher level ones are specified at lower levels of granularity.

For example, the functional requirement of *generating a settlement offer* may include lower level functional requirements like that of determining the value of the claim items and that of assessing to what extend the p0olicy covers the claim.

Note
Often the detailed requirements around the different domains of responsibility are obtained from different role players; i.e. while certain domains of business may be able to provide information around the higher level business process, the details concerning lower level responsibilities are often determined from domain experts in the appropriate domains of responsibility.

7.4.2.2.1 Pre-conditions, post-conditions and quality requirements

In contract-driven development a services contract is specified by the

- service signature with inputs and outputs,
- pre-conditions,
- post-conditions, and
- quality requirements.

The pre-conditions are those conditions under which the service may be refused without breaking the contract.

The post-conditions are those conditions which must hold once the service has been provided. They apply to the success scenarios of the use case. Each functional requirement directly maps onto a post-condition.

Finally, there may be quality requirements which are specific to this use case. Quality requirements are non-functional requirements referring to the realizable quality of service [BCK_2003_SAIP]. They refer to aspects like scaleability, reliability, performance, integrability, ... and are the core drivers behind architecture and infrastructure. While the pre- and post-conditions are part of the functional requirements which are realized through design, the quality requirements are used to assess whether the target architecture for the use case can indeed host the use case or whether architectural adjustments need to be made in order to realize the required quality requirements.

7.4.2.2.1.1 Bibliography

[BCK_2003_SAIP] Len Bass, Paul Clements, and Rick Kazman, *Software Architecture in Practice*, second edition, April 2003, Addison-Wesley Professional.

7.4.2.3 User work flow

The required user work flow is documented via interaction diagrams showing only the messages exchanged between the subject responsible for realizing the use case and the actors.

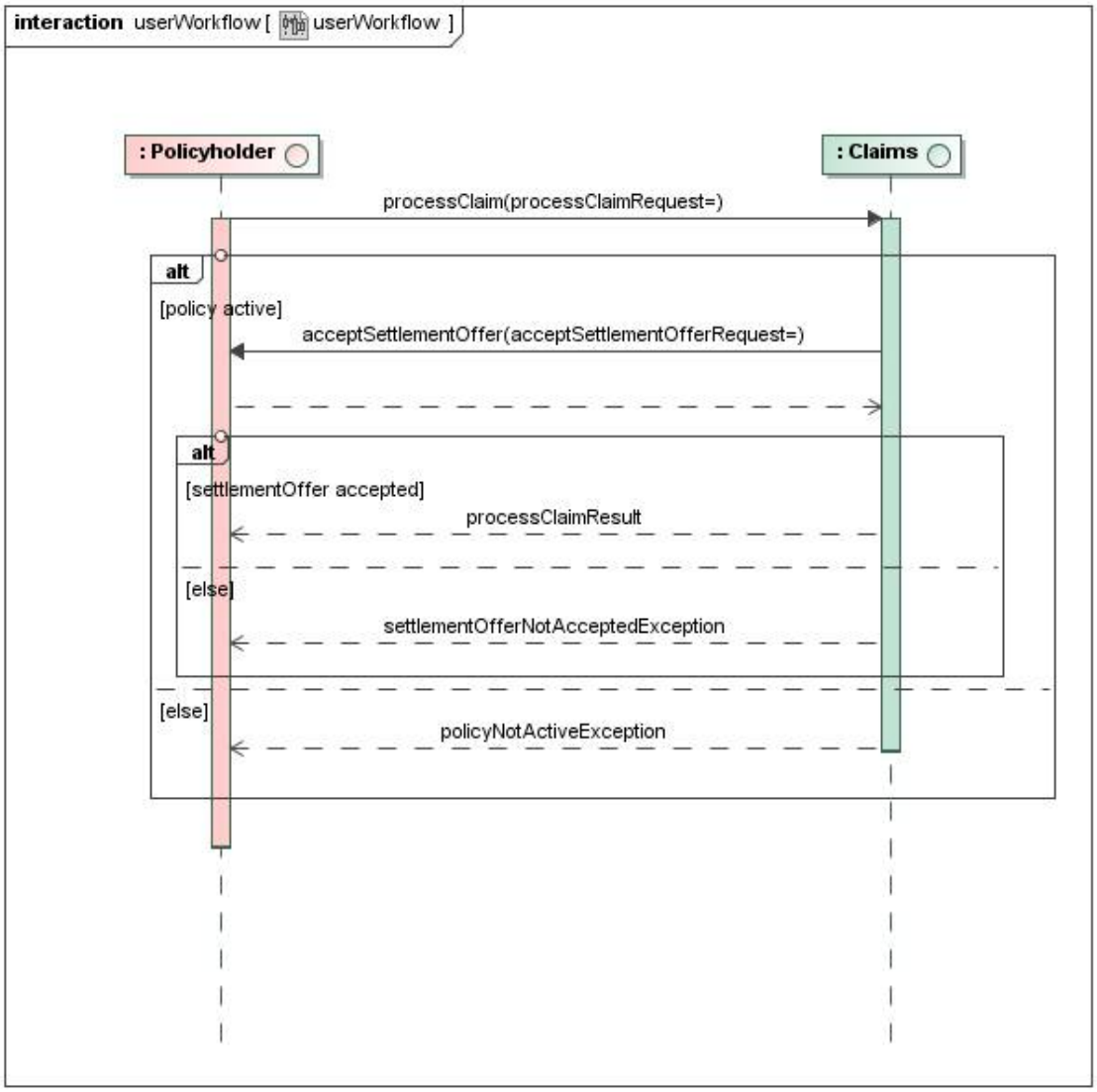

Figure 7.6: The user work flow for a success scenario of the use case.

For example, Figure **??**, shows the interactions of the subject with the actors for a particular scenario and specifies the value objects exchanged between them.

7.4.2.4 The services contract

In contract-driven development a services contract is specified by the

- service signature with inputs and outputs,
- class diagrams specifying the data structure requirements for the inputs and outputs,

- pre-conditions,
- post-conditions, and
- quality requirements.

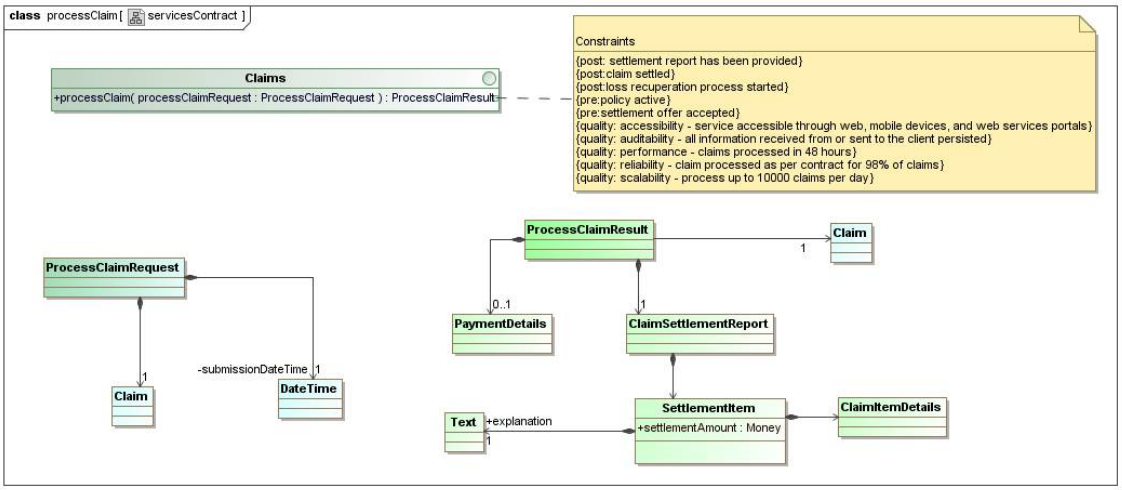

Figure 7.7: The services contract for the process claim service.

7.4.3 The design phase

7.4.3.1 Defining the use case contract

During an URDAD design phase one identifies the responsibilities for the current level of granularity, assigns them to services contracts and specifies the business process the role players realizing the contract need to execute. One then projects out the collaboration context, i.w. the static structure supporting the collaboration which realizes the use case.

The output of the design phase is the technology neutral business process design for that level of granularity. This will include

- the service providers required for the current level of granularity,
- the business process for the current level of granularity,
- the collaboration context which resembles, in a technology neutral way, that subset of the static structure required to support the business process for the current level of granularity, and
- class diagrams for any object exchanged between the role players of the current level of granularity.

7.4.3.2 Responsibility identification and allocation

During the first step of an URDAD design phase one groups functional requirements into responsibility domains and assigns each responsibility domain to a separate services contract. Note that the technology neutral design assigns responsibilities not to concrete implementation classes, but instead to service provider contracts. These contracts can be realized by implementations in different technologies.

In the context of a model driven development process, the choice of a concrete service provider or the technology within which a service provider is o be realized is made during the implementation mapping phase. A services contract can be realized by a system, a system component, an organizational component (e.g. a business unit) or an external service provider to whom the organization has outsourced the responsibility covered by the services contract.

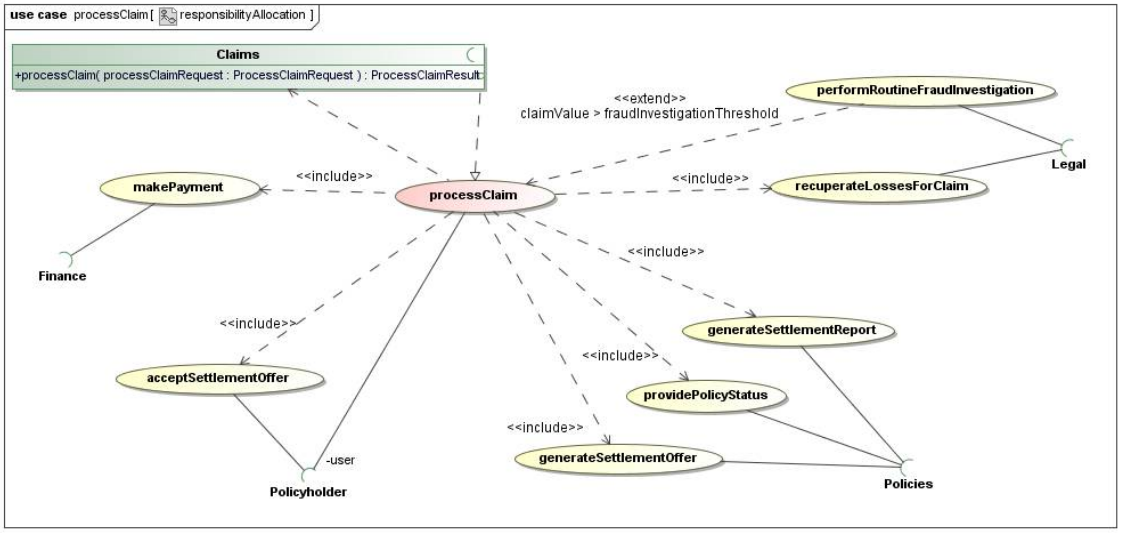

Figure 7.8: Responsibility identification and allocation for the process claim use case

URDAD requires that one adds the responsibility for managing the work flow and assigns the responsibility to a separate services contract. This decouples the service providers from one another, localizes the business process information for the current level of granularity and removes any business process information from the service providers themselves. They are simply there to provide reusable services around a responsibility domain without knowledge of the business processes for which these services are required.

Figure **??** shows the responsibility identification and allocation for the process claim use case.

7.4.3.3 Business process specification

The services contracts are first introduced abstractly without specifying the services which service providers realizing the services contract need to provide. Instead one next designs the business process for the current level of granularity, showing how these abstract service providers need to collaborate in order to realize the use case. The business process design feeds the services required for the business process into the services contracts for the service providers required for the business process.

7.4.3.3.1 The activity diagram specifying the business process for the processClaim service

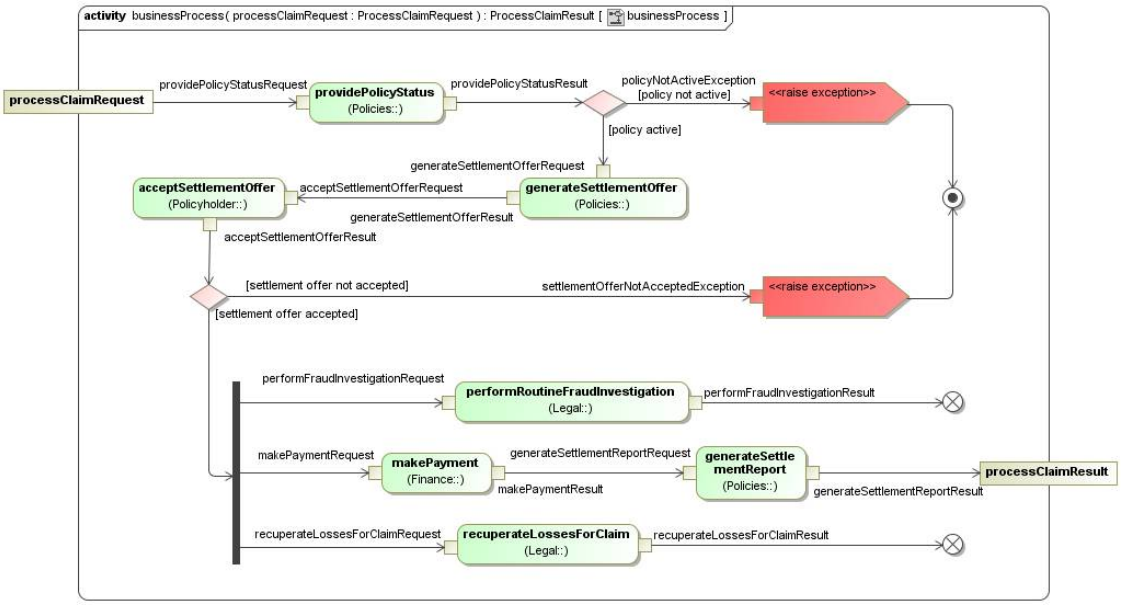

Figure 7.9: Activity diagram showing how the controller assembles the business process across services sourced from service providers.

7.4.3.4 Projecting out the collaboration context

The collaboration context shows the service providers required, at a specific level of granularity, to realize the use case, the services they need to provide for this use case and the message paths we require in order for the service providers to be able to collaborate to realize the use case.

Figure **??** shows the collaboration context for the process claim use case. Note that the dynamics (i.e. the business process specification) will already have fed in the services required for the business process into the contracts for the individual service providers.

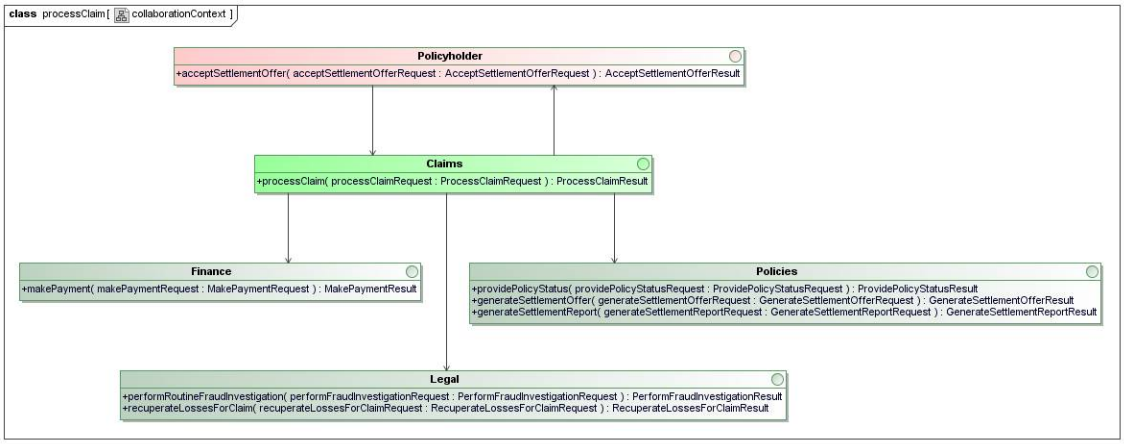

Figure 7.10: The collaboration context for the process claim use case

7.4.4 Transition to next level of granularity

Having completed one analysis/design cycle, one needs to ask oneself whether the business process for the use case has been fully specified or not. If not, one may need to go to lower levels of granularity for some or all of the service providers from the current level of granularity.

Note
Often the lower level granularity design is done by different business analysts who understand that domain of responsibility (e.g. from a different department of the organization) or by the business analysts of other organizations to whom the realization of the services contract is outsourced.

In order to execute the transition to the next lower level of granularity, one selects one of the service providers as the new context. The services from the current level of granularity become the lower level use cases. After all, a use case is defined as a service of value[**?**]. One then selects a particular service or use case and repeats the lower level analysis and design process.

Figure **??** shows an example of stake holders around the lower level use case of providing a settlement offer together with their functional requirements around that use case.

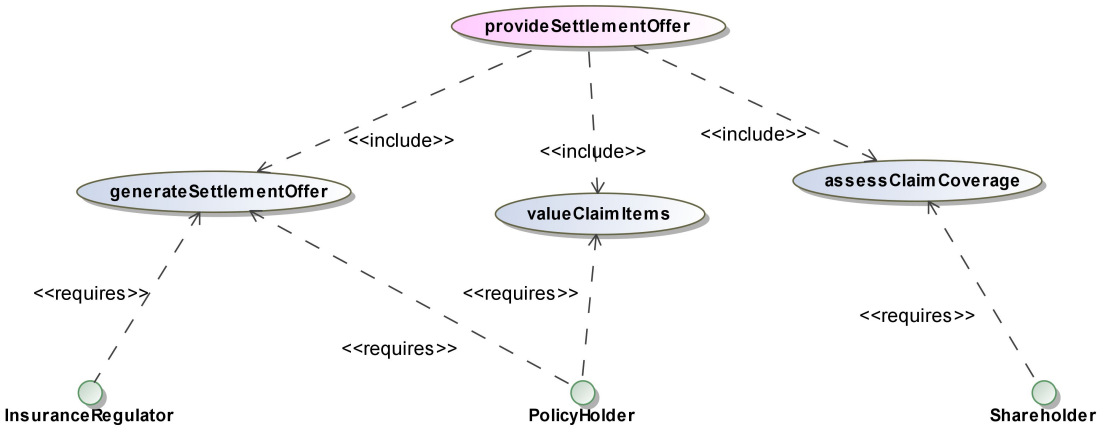

Figure 7.11: Functional requirements for the provide settlement offer service

The lower level design phase is executed in the same way as one was done for the higher level of granularity. It start with the grouping of functional requirements into responsibility domains and the allocation of each responsibility domain to a separate services contract. Figure **??** Figure \ref{fig:assessCoverageResponsibilityAllocation} shows an example of identifying and allocating the lower level responsibilities around assessing the policy coverage.

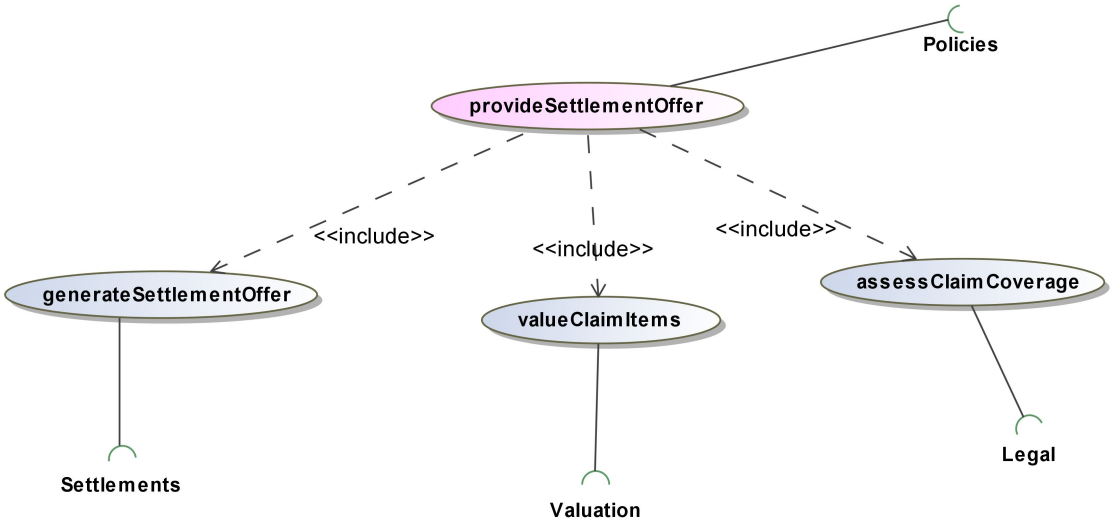

Figure 7.12: Responsibility allocation for the provide settlement offer service

7.4.4.1 Facilitating navigation across levels of granularity

In URDAD a service from one level of granularity is mapped onto a use case at the next lower level of granularity. In order to be able to conveniently navigate across levels of granularity, one needs to maintain the link between the the service and its corresponding use case. This can be done in various UML tools by adding a link with an appropriate stereotype.

7.4.4.2 Bibliography

[Frankel_2003_MDAA] James Rumbaugh, Ivar Jacobson, and Grady Booch, *Unified Modeling Language Reference Manual*, ISBN 978-0471319207, 2004, second edition, Addison-Wesley Professional.

7.5 URDAD views

URDAD defines six standard views (diagrams) which can be used to construct and present an URDAD model. Three views are part of the analysis for the service/use case whilst the other 3 views are part of the The views include

- the *services contract view* specifying the stake holder requirements,
- the *user work flow view* specifying the required interaction between the user and the service provider
- the *functional requirements view* specifying the functionality (lower level services) required to address the stakeholder requirements,
- the *responsibility allocation view* specifying the services contracts to which the functional requirements are assigned to,
- the *process specification view* specifying the (business) process through which the service is realized, and
- the *collaboration context view* specifying the role players (contracts) collaborating in the process realizing the service and the

7.6 How are the design activities realizing the desired design attributes embedded in URDAD?

The single responsibility principle is directly enforced by grouping functional requirements into responsibility domains and requiring the each responsibility domain is assigned to a separate contract.

The levels of granularity are fixed by including only those contracts to which the responsibilities for a particular level of granularity have been assigned. Furthermore, the level of granularity is further fixed be requiring that the lowest level service requests at a particular level of granularity are those which come from the controller for that level of granularity.

The locking into services contracts is enforced by directly assigning responsibility domains to contracts and specifying the work flow across these contracts. The URDAD design process then generates the contract details and requires the specification of pre-and post-conditions as well as quality requirements.

URDAD directly enforces the introduction of a work flow controller for each responsibility domain and each level of granularity, resulting in the localization of the business process information and decoupling of the service providers used in the business process.

The relationship between the layers of granularity are documented through an explicit transition across the layers of granularity, facilitating bidirectional traceability.

Finally, the minimal conceptual (technology neutral) structure supporting the collaboration is projected out from the dynamics of the business process realizing the use case.

7.7 Evaluating an URDAD based design

In order to assess an URDAD based design one will

- validate that each functional requirement is indeed addressed by the business process,
- assess the grouping of functional requirements into responsibility domains in order to verify that there are no overlaps between responsibility domains and that each responsibility domain does indeed comprise a single responsibility,
- verify that the process at any level of granularity is intuitive and simple,
- verify that the service providers are represented by services contracts (UML interfaces) and not by implementation or technology specific classes,
- verify that each services contract has been fully specified including the functional and non-functional requirements,
- verify that the structure of all exchanged value objects is defined using class diagrams.

7.8 The URDAD profile

7.9 URDAD model organization

7.9.1 Why is the model organization important?

If one does not have a well defined strategy for organizing a model, it is very likely that the model will deteriorate very quickly into an unmanageable mess. You need to effectively organize your model such that you are able to

- find elements more rapidly,
- efficiently navigate and work on the model,
- more simply delete and move elements, i.e. moving or deleting elements moves or deletes everything which is only relevant for that element,
- easily modularize the model as it grows.

7.9.2 Guiding principles for model organization

The core guidoing principles behind the URDAD model organization are:

- **Responsibility localization** Group laterally into responsibility domains.
- **Layering** Group horizontally in terms of layers of granularity.
- **Cohesion** Package element specific artifacts within the element.

7.9.3 URDAD rules for model organization

1. Use cases at a particular level of granularity are grouped into packages according to their responsibility domains, i.e. financial services into finance, legal services into legal, ...
2. Use cases across levels of ganularity are packages in sub-packages.
3. All use case artifacts are packaged within the use case. These include
   - the functional requirements specification,
   - the contract specification for the use case,
   - the user work flow specification,
   - the responsibility allocation,
   - the business process specification, and
   - the collaboration context specification for the use case.
4. The higher level request and response objects which are service specific are packaged with the service.
5. Entities and value objects which are not service specific are packaged within the conceptual domain, irrespective of the use cases/services within which they are used.
6. Constraints which are element specific are packages inside the element for which they were created.

7.9.4 Example of the organization of an URDAD model

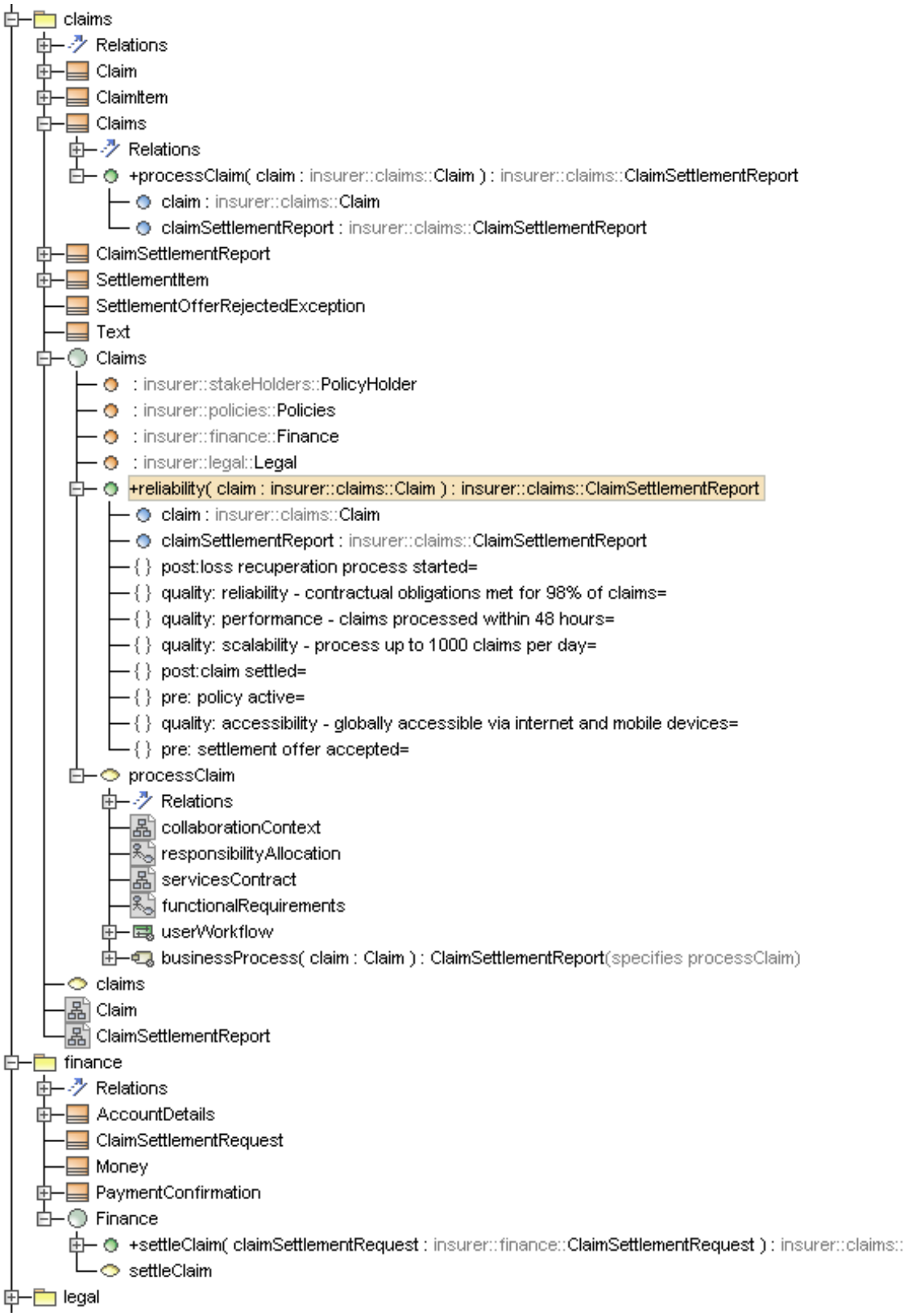

Figure 7.13: Example of the organization of an URDAD model for an insurer.

7.10 URDAD documentation generation

7.10.1 Introduction

To generate useful documentation from a general UML model is very difficult due to

- the richness and complexity of the UML language,
- the purpose and semantics of the different model elements is not necessarily known, and
- as there is no predefined structure for the UML model - different UML models can have very different structures.

However, when following the URDAD analysis and design methodology, both documentation and code generation are considerably simpler because

- URDAD enforces the use of a very small subset of the UML,
- the URDAD components have a well defined semantics and
- an URDAD model has a well defined, simple structure which is the same across levels of granularity.

7.10.2 Why documentation generation?

One usually needs to provide documentation for various role players in order to make the information relevant to them accessible. If one does not use documentation generation to generate the documentation from the model, then the documentation must manually be kept synchronized with the information contained in the model.

Generating the documentation from the model

- ensures that the documentation remains synchronized with that of the evolving model, and
- reduces the probability that there are errors, i.e. that the documentation does not reflect the true semantics and content of the model.

Need for version control
Since the model will typically evolve over time one needs support for version control. Most UML tools either

- provide version control through add-on components, or
- provide adapters which integrate with external version control systems.

In this case the generated documentation would not have independent versioning, but would adopt the versioning of the model from which it is generated.

Further benefits of documentation generation include that one can

- extract information for different role players and different purposes, excluding any information which is not required,
- present the information in different ways depending on the background and needs of the information consumers.

7.10.3 Types of documentation

The types of documentation one may find useful may really be very varied and may evolve continuously. Usually different role players and stake holders may find different types of reports useful. Commonly one may wat to generate documentation for

- business,
- business analysts,
- architects,
- developers,
- quality assurance,
- service providers,
- operations, and for
- users (e.g. clients).

7.10.3.1 Documentation required by business

Business would typically want to see

- the scope of services offered by the organization or a component of the organization,
- the requirements for a service offered by the organization or a component of the organization,
- the business process for a service offered by the organization,
- a list of service providers used by the organization.

7.10.3.2 Documentation required by business analysts

Business/domain analysts are the ones which perform the requirements analysis and the technology neutral (business) process design constructing the platform independent model (PIM) which is often called the business model in URDAD.

They would thus be comfortable working with the model itself. Nevertheless, for them too it is often useful to generate reports rendering specific information extracted from the model. In particular, they would typically require

- documents capturing the requirements specification for a use case including the data structure specification for exchanged value objects, and
- document capturing the technology neutral business process design for a use case.

In particular, they would typically require analysis and design information actoss levels of granularity.

7.10.3.3 Documentation required by architecture

Architecture requires from an URDAD-based technology neutral (business) model reports for

- the scope of services, and
- use case contracts.

7.10.3.4 Documentation required by developers

Developers require sufficient implementation to perform the implementation mapping. In addition they want to generate unit tests for their implementation. The reports they would typically want would include

- the services contracts for the services they need to implement and test, and
- the technology neutral design for these services.

This information is often required across levels of granularity.

7.10.3.5 Documentation required by quality assurance

Quality assurance is responsible for assessing whether the use case contracts are realized. In particular they need to assess whether both, the functional and non-functional requirements are met. For this they require reports which contain the full contract specifications for use cases which include the functional and quality requirements.

7.10.3.6 Documentation required by service providers

Service providers need to know what the services required of them need to provide. As such they require reports which contain the full services contract specifications for use cases which include the functional and quality requirements.

7.10.3.7 Documentation required by operations

Operations is responsible for executing and overseeing the execution of the business processes. They need reports which contain

- the services contracts for the services they need to provide,
- the technology neutral design for these services.

7.10.3.8 Documentation required by users

Users typically need to know what the services will do for them and what their obligations are when making use of the services. To this end they require reports containing

- the services contracts for the services offered,
- the user work flow specification, and
- the services contracts which contain the user obligations.

7.10.4 Documentation generation approaches

There are two main approaches typically taken to documentation generation. One either

- uses the report generation support for a particular UML tool, or
- generating all documents directly off the tool-indepedent object model.

7.10.4.1 Using the report generation facilities of a UML tool

The report generation facilities provided by UML tools will be very tool specific. One will thus have to look at the They typically make simple report generation easy, often providing a set of predefined report types.

7.10.4.1.1 Advantages of UML tool specific report generation

- Usually it is very easy to define a simple report.
- Often there are a number of simple report templates which one can modify according to one's needs.
- One has access to the drawn diagrams for inclusion in the report.

Note
This is actually often not an advantage but a disadvantage. For example, there is no guarantee that the functional requirements diagram drawn by a business analyst contains all functional requirements and other elements defined in the model or that it does not contain elements that should not be shown in that view (e.g. including fucntional requrirement from a lower level of granularity). When following a solid methodology like URDAD with a well defined PIM structure (i.e. a well defined structure for the techology neutral business model), then one can generate the diagrams from the model.

7.10.4.1.2 Disadvantages of UML tool specific report generation

- The report generation is tied to a tool and will have to be redeveloped when changing tools.
- Often the report generation is quite limited with, at times, very limited access to the full object model.
- The tools used often do not scale well with complexity of the report, i.e. the maintenance costs for the report generation templates and tools may become very high.

7.10.4.2 Generating reports directly off the object model

One can generate reports directly from the object model usually stored in XMI. Doing this decouples the report generation from the actual UML tool used.

7.10.4.2.1 Advantages of generating reports directly from the object model

- One can change UML tools without having to redevelop or modify the documentation generation templates and tools.
- One will have to generate the diagrams form the model, but doing this ensures that the diagrams are complete and correct, i.e. that they have all relevant information from the model and that they do not include elements which should not be contained in the diagram.
- One is not limited to the functionality provided by a particular tool.
- One can typically generate a wider variety of reports and render them in a wider variety of rendering technologies.
- The complexity of the documentation generation templates and tools is typically more manageable.

7.10.4.2.2 Using APIs into the object model

There are a number of APIs like Eclipse EMF framework which provide access to the object model. These can be used in conjunction with other text generation frameworks like Apache Velocity to generate reports off the UML model.

7.10.4.2.3 Generating reports using QVT

The QVT is OMG's standard language and tools framework for performing model transformations. Report generation can be seen as a for of model transformation similar to code generation. While the inputs for code generation will be the technology neutral model (the PIM) and the platform description model (the PDM), the inputs for documentation or report generation will be the technology neutral (business) process design (the PIM) and the report specification.

7.10.5 Documentation generation in MagicDraw

MagicDraw has quite powerful report generation facilities which enable one to generate a wide variety of reports in a wide variety of formats. By default the report generation generates reports in RTF, but one can realtively easily define one's own script which generates a report in a different format.

7.10.5.1 Default URDAD service documentation

The current URDAD default report is available in the form of open source apache velocity scripts executed by MagicDraw within the standard MagicDraw report wizards. It makes extensive use of MagicDraw's OpenAPI to interrogate the MagicDraw object model in order to

- generate a services report off an URDAD model, and
- perform some basic URDAD model validation, checking for correctness and completeness like
  - that the user role for each use case has been specified,
  - that each functional requirement is required,
  - that each functional requirement has been assigned to a services contract ,
  - that the data structures for the inputs and outputs of services have been defined, ...

Outlook
It is envisaged that

- that the URDAD documentation generation will move towards directly extracting the information from an XMI based object model,
- the the QVT framework will be used to perform the model transformation for the documentation generation,
- that the diagrams will be generated from the model and no longer requested from an UML tool - this ensures the integrity of the diagrams and also enables the generation of diagrams which were never drawn (e.g. the collaboration context),
- that the model validation will be done via OCL based validation suites which will be used by the documentation generation in order to report model deficiencies.

7.10.5.1.1 Output format

The output format for the current URDAD reports is OASIS docbook, an open specification for XML based documentation. This is an open format which, being XML, is easy to work with. Furthermore, it is easy to validate that the document structure is correct.

From here one can render to a variety of formats like PDF, LaTeX or even ODF, the opend document format. Alternatively one can chain the commands to render directly into PDF.

7.10.5.1.2 Obtaining the default URDAD report templates

The URDAD report templates are open sourced and can be downloaded from `http://sourceforge.net/projects/-urdaddoctempl/`.

7.10.5.1.3 Running the default URDAD report

You will first have to register a new report template with MagicDraw before you can generate reports.

7.10.5.1.3.1 Registering the URDAD default report template

1. Go to *Tools -> Report Wizard* and press on *new* to register a new report.
2. In the dialog box
   - Give the report a name, e.g. `Default URDAD use case report`, and
   - assign it to a category (here you could type in URDAD as a new category for the various URDAD reports).

7.10.5.1.3.2 Generating the report against a use case from your project

To run the URDAD report from within MagicDraw you

1. Go to *Tools -> Report Wizard* and press on *next*.
2. Select default for the report data and press *next*.
3. In the user defined fields add the `targetUseCaseQualifiedName` variable and assign its value to the fully qualified name of the use case for which you would like to run the report, e.g. `insurer::claims::Claims::processClaim` and press *next*.
4. In the *select element scope* dialog, select the entire model via *add all* (the report generator should be able to extract whatever it needs from the model as a whole) and press *next*.
5. In the final dialog, enter the file name you would like to have for the report (e.g. useCaseName.docbook) and press *generate* to generate the report.

7.10.5.1.4 Use case: processClaim

URDAD based Analysis and Design, February 21, 2009

7.10.5.1.4.1 Introduction

This is an URDAD complient document for the requirements analysis and business process design of the processClaim use case. The document shows the requirements and design for this level of granularity. The details of the services used within the business process are shown in the analysis and design of the respective use cases for these lower level services.

This use case is offered to policy holders enabling them to request the processing of a claim against a policy. The fully qualified name of the use case is insurer::claims::Claims::processClaim.

7.10.5.1.4.2 Analysis

The analysis section specifies the stakeholder requirements, the user work flow, the data structures for the exchanged value objects and the service contract with the pre- and post-conditions and quality requirements. It contains the requirements specification for the current level of granularity.

7.10.5.1.4.3 Functional requirements: processClaim

The functional requirements diagram shown in Figure **??** shows the processClaim use case, the stake holders who have an interest in that use case and their functional requirements.

Figure 7.14: The stakeholder requirements diagram for the processClaim use case.

7.10.5.1.4.4 The user role

The user role is the role played by those objects which make use of the use case. It is represented by an interface/contract which accumulates the contractual obligations of the user itself.

- **PolicyHolder** Objects who play the role of a PolicyHolder make use of the processClaim use case. The role played by a party which owns an insurance policy.

7.10.5.1.4.5 Stake holders

The following stake holders have an interest in the processClaim use case:

- **InsuranceRegulator** The role played by a party which regulates the insurance sector.
- **PolicyHolder** The role played by a party which owns an insurance policy.
- **Shareholder** The role played by an investor in the insurance company.

7.10.5.1.4.6 Mandatory functional requirements

The following functional requirements need to be addressed for any success scenario of the use case, i.e. for any scenario where the user obtains the value from the use case.

- **acceptSettlementOffer** Accept the settlement offer in a legally binding way. This is required by
  - Shareholder
- **provideSettlementOffer** Provide a settlement offer for a claim. This is required by
  - InsuranceRegulator
  - PolicyHolder
- **settleClaim** Perform any financial and other transactions required to settle a claim according to a provided settlement offer. This is required by
  - PolicyHolder
- **providePolicyStatus** Determine and provide the current state of a policy. This is required by
  - Shareholder
- **update claims history** The claims history must be updated with any settled claim. This is required by
  - Shareholder
- **recuperateLosses** Perform the legal processes against accountable parties which recuperate the losses or part thereof incured due to an insurance claim.

ERROR: Missing stakeholder specification
URDAD requires that each functional requirement is linked to the stakeholder(s) who require it via a requires relationship. The recuperateLosses requirement is not linked to a stakeholder via a requires relationship.

7.10.5.1.4.7 Conditional functional requirements

The following functional requirements need to be addressed only under certain conditions:

- **performFraudInvestigation** Perform a routine fraud investigation in order to determine whether there is a reasonable likelyhood of fraud. This functional requirement needs to be addressed if [settlementAmount > fraudInvestigationThreshold]. This is required by
  - Shareholder

7.10.5.1.4.8 User work flow: processClaim

The user work flow diagram shown in Figure **??** specifies how users interact with the service provider in the context of making use of the processClaim use case. It shows the messages exchanged in the various scenarios.

Policy holders request the processing of a claim by submitting the claim to the insurer. If the policy against which the claim is made is inactive, the policy holder is informed of this and the claims processing is aborted. Otherwise the insurer does some pre-processing of the claim resulting in a settlement offer which the policy holder is requested to accept. If the policy holder accepts the settlement offer, the insurer completes the processing of the claim and provides a claim settlement report to the policy holder. Otherwise the insurer confirms with the policy holder that the settlement offer was rejected.

Figure 7.15: The user work flow for the processClaim use case.

7.10.5.1.4.9 Service request specifications

This section specifies the services requested in the user work flow including the data structures for the inputs and outputs (i.e. requests and responses) for each service.

7.10.5.1.4.10 Service: acceptSettlementOffer

Policy holders need to accept settlement offers before claims are settled. This is the service through which a policy holder is requested to accept a settlement offer.

7.10.5.1.4.11 Input (request object): SettlementOffer

The request object for the acceptSettlementOffer service is a SettlementOffer. It contains the information which must be provided with the service request.

ERROR: missing data structure specification
The data structure requirements for each exchanged value object needs to be specified. The data structure of SettlementOffer has not been specified.

7.10.5.1.4.12 Output (response object): SettlementOfferAcceptance

ERROR: missing data structure specification
The data structure requirements for each exchanged value object needs to be specified. The data structure of SettlementOfferAcceptance has not been specified.

7.10.5.1.4.13 Service: processClaim

This is the service for processing a claim against an insurance policy.

7.10.5.1.4.14 Input (request object): Claim

The request object for the processClaim service is a Claim. It contains the information which must be provided with the service request.

The claim contains information required for the processing of a claim including information about the claimant, the policy against which the claims is made, the claim items as well as any supporting information.

Figure 7.16: Data structure (class) diagram for Claim

7.10.5.1.4.15 Output (response object): ClaimSettlementReport

The claim settlement report contains information of how the settlement amounts for each item as well as potentially information pertaining to the calcualtion of the settlement amount and information on how the settlement was made, e.g. information about the financial transaction for the settlement.

Figure 7.17: Data structure (class) diagram for ClaimSettlementReport

7.10.5.1.4.16 Service contract: processClaim

The service contract specification diagram shown in Figure **??** specifies the contract (interface) name, the service with the request and response objects, the pre- and post-conditions and the quality requirements for the service.

The pre-conditions are those conditions under which the service may be refused without being in breach of services contract. The post-conditions are those conditions which must hold true once the service has been rendered. The quality requirements are the non-functional requirements.

The process claim service receives as input a Claim and returns, upon successful completion, a claim settlement report. The claim processing is aborted if either the policy is not active or the policy holder does not accept the settlement offer. Otherwise the claim processing is completed. On completion of the claims processing, the policy holder will have accepted the settlement offer, the claim will have been settled and the loss recuperation process will have been started.

Figure 7.18: The service contract for the processClaim use case.

7.10.5.1.4.17 Technology neutral process design

This section specifies the technology neutral design realizing the use case requirements. In particular, it specifies which functional requirement is assigned to which services contract, how the business process is assembled from these services and the collaboration context containing the services required from the various service providers together with the inputs and outputs for each service and the message paths to the service providers.

7.10.5.1.4.18 Responsibility identification and allocation: processClaim

Figure **??** shows the grouping of functional requirements into responsibility domains. Each responsibility domain is assigned to a separate services contract.

The responsibility of settling the claim is assigned to Finance. The claims process is managed by Policies is responsible for providing a settlement offer for the claim and for providing the status of a policy. The policy holder is required to accept the settlement offer while legal services is responsible for recuperating losses and performing fraud investigations.

Figure 7.19: The responsibility allocation diagram for the processClaim use case.

7.10.5.1.4.19 Controller

The controller takes ownership of the business process and manages the business process for the use case. The controller is responsible for assembling the business process across lower level services sourced from service providers realizing lower level services contracts. All work flow decisions and control logic are executed by the controller. As such the controller also decouples the lower level service providers from one another, i.e. the lower level service providers are have know knowledge of either the business process within which the services are used or any other service providers participating in the business process.

- The controller specifying the business process for the processClaim is class com.nomagic.uml2.impl.magicdraw.classes.mdkernel.Pack
  It realizes the use case via the processClaim service.

7.10.5.1.4.20 Services contracts for the required responsibility domains

The functional requirements for the processClaim have been allocated to the following responsibility domains:

- **Finance** Provide financal services. The following functional requirements have been assigned to the Finance services contract:
  - **settleClaim** Perform any financial and other transactions required to settle a claim according to a provided settlement offer.
- **Policies** Manage policies. The following functional requirements have been assigned to the Policies services contract:
  - **provideSettlementOffer** Provide a settlement offer for a claim.
  - **provideSettlementOffer** Provide a settlement offer for a claim.
  - **providePolicyStatus** Determine and provide the current state of a policy.
  - **update claims history** The claims history must be updated with any settled claim.
- **PolicyHolder** The role played by a party which owns an insurance policy. The following functional requirements have been assigned to the PolicyHolder services contract:
  - **acceptSettlementOffer** Accept the settlement offer in a legally binding way.
- **Legal** Provide legal services. The following functional requirements have been assigned to the Legal services contract:
  - **performFraudInvestigation** Perform a routine fraud investigation in order to determine whether there is a reasonable likelyhood of fraud.
  - **recuperateLosses** Perform the legal processes against accountable parties which recuperate the losses or part thereof incured due to an insurance claim.

7.10.5.1.4.21 Process specification: processClaim

Figure **??** shows how the processClaim service is assembled from services sourced from the service providers to whom the functional requirements have been assigned.

This is the business process for the processClaim service as offered by Claims. Claims receives a claim and first requests policies to provide the policy status for the policy against which the claim is made. If the policy is inactive, claims terminates the claims processing process by throwing a PolicyInactiveException. If the policy is active claims requests policies to provide a settlement offer for the claim. Having received the settlement offer claims checks whether the settlement amount exceeds the fraud investigation threshold or not. If it does, legal services is requested to launch a routine fraud investigation for the claim. In either case the policy holder is requested to accept the settlement offer. If the policy holder rejects the settlement offer, the business process is aborted, notifying the client via a SettlementOfferRejectedException. Otherwise Legal services is requested to start the process of recuperating the losses for the claim while finance is requested to settle the claim and a ClaimSettlementReport is provided to the policy holder.

Figure 7.20: The business process specification diagram for the processClaim use case.

7.10.5.1.4.22 Collaboration context: processClaim

Figure **??** shows the services required for the processClaim use case from the different service providers as well as the required message paths through which the services can be requested.

In order to process a claim, Claims requires from Policies the services to provide a settlement offer for the claim and to provide the policy status. The policy holder needs to accept the settlement offer, Finance needs to provide the claim settlement service and Legal needs to perform fraud investigations and recuperate losses for a claim.

Figure 7.21: The collaboration context diagram for the processClaim use case.

7.10.5.1.5 The template for the default URDAD service report

The current URDAD documentation generation makes use of Apache Velocity to generate a docbook document.

7.10.5.1.5.1 urdadStandardReport.txt

This is the main file which is modularized in order to make the report template more mangeable. It defines

- variables for the URDAD stereotypes,
- variables for UML/MagicDraw concepts, and
- a range of macros which extract information from the UML model and obtains diagrams from MagicDraw. Examples include to
  - obtain all functional requirements for a use case,
  - to obtain the inputs and outputs for a servcice, ...

Finally it imports the modules for the templates definging the introduction, the analysis section and the design section of the generated document.

```
                <?xml version="1.0" encoding="utf-8"?>
<?oxygen RNGSchema="http://www.oasis-open.org/docbook/xml/5.0b5/rng/docbookxi.rng" type=" ↩
    xml"?><!--Authored by mailto:fritz@solms.co.za on 2007-06-01 -->
#*
This is the Open Document Standard template for the default docbook based URDAD report.

Notes:
-----
  - the $targetUseCaseQualifiedName must be provided as a user-defined variable
  - the $documentType variable which specifies the type of document to be generated.  ↩
      Currently this will be one of
       * standard
    * minimal
       * contract

TODOs:
------
  - actors of use case diagrams
*#

##-------------------------------------------------
## URDAD stereotypes
##-------------------------------------------------
#set($urdad_functionalRequirementsDiagram = "functionalRequirementsDiagram")
#set($urdad_userWorkflowDiagram = "userWorkflowDiagram")
#set($urdad_serviceContractDiagram = "serviceContractDiagram")
#set($urdad_responsibilityAllocationDiagram = "responsibilityAllocationDiagram")
#set($urdad_businessProcessDiagram = "businessProcessDiagram")
#set($urdad_collaborationContextDiagram = "collaborationContextDiagram")
#set($urdad_requires = "requires")
#set($urdad_valueObject = "valueObject")
#set($urdad_raiseException = "raiseException")
##-------------------------------------------------

##-------------------------------------------------
## UML concepts
##-------------------------------------------------
##STRUCTURE
#set ($umlInterfaceClass = "com.nomagic.uml2.impl.magicdraw.classes.mdinterfaces. ↩
    InterfaceImpl")
```

```
#set($umlOperation = "com.nomagic.uml2.impl.magicdraw.classes.mdkernel.OperationImpl")
##RELATIONSHIPS
#set($umlAssociationRelationship = "com.nomagic.uml2.impl.magicdraw.classes.mdkernel. ↩
    AssociationImpl")
#set($umlUsageRelationship = "com.nomagic.uml2.impl.magicdraw.classes.mddependencies. ↩
    UsageImpl")
#set($umlDependencyRelationship = "com.nomagic.uml2.impl.magicdraw.classes.mddependencies. ↩
    DependencyImpl")
#set($umlRealizationRelationship = "com.nomagic.uml2.impl.magicdraw.classes.mddependencies. ↩
    RealizationImpl")
#set($umlGeneralizationRelationship = "com.nomagic.uml2.impl.magicdraw.classes. ↩
    mddependencies.GeneralizationImpl")
##ACTIVITY DIAGRAMS
#set($umlActivityParameterNode = "com.nomagic.uml2.impl.magicdraw.activities. ↩
    mdbasicactivities.ActivityParameterNodeImpl")
#set($umlCallOperation = "com.nomagic.uml2.impl.magicdraw.actions.mdbasicactions. ↩
    CallOperationActionImpl")
#set($umlForkNode = "com.nomagic.uml2.impl.magicdraw.activities.mdintermediateactivities. ↩
    ForkNodeImpl")
#set($umlDecisionNode = "com.nomagic.uml2.impl.magicdraw.activities. ↩
    mdintermediateactivities.DecisionNodeImpl")
#set($umlAcivityFinalNode = "com.nomagic.uml2.impl.magicdraw.activities.mdbasicactivities. ↩
    ActivityFinalNodeImpl")
#set($umlFlowFinalNode = "com.nomagic.uml2.impl.magicdraw.activities. ↩
    mdintermediateactivities.FlowFinalNodeImpl")
#set($umlAcceptEventAction = "com.nomagic.uml2.impl.magicdraw.actions.mdcompleteactions. ↩
    AcceptEventActionImpl")
#set($umlSendSignalAction = "com.nomagic.uml2.impl.magicdraw.actions.mdbasicactions. ↩
    SendSignalActionImpl")
##-------------------------------------------------
## MACRO definitions
##-------------------------------------------------
##
#macro (getUseCase $qualifiedUseCaseName)
#set($result = '')
#foreach ($useCase in $UseCase)
#if ($useCase.qualifiedName == $qualifiedUseCaseName)
#set ($result = $useCase)
#end
#end
#end
##-------------------------------------------------
##
#macro ( getDiagramForUseCase $useCase  $stereotypeName)
#set($result = '')
#foreach  ($diagram in $Diagram)
#if (($diagram.owner == $useCase) || ($diagram.owner.owner == $useCase))
##   for sequence and activity diagrams owned by an interaction which is owned by the use  ↩
    case
#if ($report.containsStereotype($diagram,$stereotypeName))
#set ($result = $diagram)
#end
#end
#end
#end
##-------------------------------------------------
##
#macro ( getMandatoryFunctionalRequirements $useCase)
#set ($result = $array.createHashSet())
#foreach($includeRelationship in $useCase.include)
#set ($dummy = $result.add($report.getSupplierElement($includeRelationship)))
#end
```

```
#end
##-------------------------------------------------
##
#macro ( getConditionalFunctionalRequirements $useCase)
#set ($result = $array.createHashSet())
#foreach ($uc in $UseCase)
#foreach($extendRelationship in $uc.extend)
#if ($report.getSupplierElement($extendRelationship) == $useCase)
#set ($dummy = $result.add($report.getClientElement($extendRelationship)))
#end
#end
#end
#end
##-------------------------------------------------
##
#macro ( getFunctionalRequirements $useCase)
#set ($functionalRequirements = $array.createHashSet())
#getMandatoryFunctionalRequirements($targetUseCase)
#set ($dummy = $functionalRequirements.addAll($result))
#getConditionalFunctionalRequirements($targetUseCase)
#set ($dummy = $functionalRequirements.addAll($result))
#set ($result = $functionalRequirements)
#end
##-------------------------------------------------
##
#macro ( getStakeHolders $functionalRequirement)
#set ($result = $array.createHashSet())
#foreach ($relationship in $functionalRequirement.get_relationshipOfRelatedElement())
#if ($report.containsStereotype($relationship,$urdad_requires))
#set($dummy = $result.add($report.getClientElement($relationship)))
#end
#end
#end
##-------------------------------------------------
##
#macro( getStakeholdersForUseCase $useCase)
#getMandatoryFunctionalRequirements($useCase)
#set($mandatoryFunctionalRequirements = $result)
#getConditionalFunctionalRequirements($useCase)
#set($conditionalFunctionalRequirements = $result)
#set ($result = $array.createHashSet())
#foreach ($functionalRequirement in $mandatoryFunctionalRequirements)
#foreach ($relationship in $functionalRequirement.get_relationshipOfRelatedElement())
#if ($report.containsStereotype($relationship,$urdad_requires))
#set($dummy = $result.add($report.getClientElement($relationship)))
#end
#end
#end
#foreach ($functionalRequirement in $conditionalFunctionalRequirements)
#foreach ($relationship in $functionalRequirement.get_relationshipOfRelatedElement())
#if ($report.containsStereotype($relationship,$urdad_requires))
#set($dummy = $result.add($report.getClientElement($relationship)))
#end
#end
#end
#end
##-------------------------------------------------
##
#macro ( getRequestedServices $interaction)
#set ($result = $array.createHashSet())
#set ($exchMsgs = $interaction.message)
#foreach ($msg in $exchMsgs)
```

```
#set($op = $msg.getReceiveEvent().getEvent().getOperation())
#if ($op)
#set($dummy = $result.add($op))
#end
#end
#end
##-------------------------------------------------
##
#macro (getInputParameters $service)
#set ($result = $array.createHashSet())
#foreach ($par in $service.getOwnedParameter())
#if ($par.getDirection() == 'in')
#set($dummy = $result.add($par))
#end
#end
#end
##-------------------------------------------------
##
#macro (getDataStructureDiagram $class)
#set($result = '')
#foreach  ($diagram in $Diagram)
#if($diagram.getQualifiedName() == $class.getQualifiedName())
#set ($result = $diagram)
#end
#end
#end
##-------------------------------------------------
##
#macro (getRelatedElements $element, $relationshipClass)
#set ($result = $array.createHashSet())
#foreach ($relationship in $element.get_relationshipOfRelatedElement())
#if($relationship.getClass().getName() == $relationshipClass)
#set($dummy = $result.add($report.getClientElement($relationship)))
#end
#end
#end
##-------------------------------------------------
##
#macro (getRelationshipsFor $element, $relationshipClass)
#set ($result = $array.createHashSet())
#foreach ($relationship in $element.get_relationshipOfRelatedElement())
#if($relationship.getClass().getName() == $relationshipClass)
#set($dummy = $result.add($relationship))
#end
#end
#end
##-------------------------------------------------
##
#macro (getRelationshipsFor $element, $relationshipClass)
#set ($result = $array.createHashSet())
#foreach ($relationship in $element.get_relationshipOfRelatedElement())
#if($relationship.getClass().getName() == $relationshipClass)
#set($dummy = $result.add($relationship))
#end
#end
#end
##-------------------------------------------------
##
#macro (getRealizingServices, $useCase)
#getRelationshipsFor($targetUseCase, $umlRealizationRelationship)
#set($umlRealizationRelationships = $result)
#set ($result = $array.createHashSet())
```

```
#foreach($umlRealizationRelationship in $umlRealizationRelationships)
#foreach($realizationSource in $umlRealizationRelationship.source)
#if ($realizationSource.class.name == $umlOperation)
#set ($dummy = $result.add($realizationSource))
#end
#end
#end
#end
##-------------------------------------------------
##
#macro (getNodes, $activity, $nodeType)
#set ($nodes = $activity.getNode())
#set($result = $array.createHashSet())
#foreach ($node in $nodes)
#if($node.getClass().getName() == $nodeType)
#set ($dummy = $result.add($node))
#end
#end
#end
##-------------------------------------------------
##
#macro (getInputParameterNodes, $activity)
#getNodes ($activity, $umlActivityParameterNode)
#set($parameterNodes = $result)
#set($result = $array.createHashSet())
#foreach ($parameterNode in $parameterNodes)
#if ($parameterNode.getParameter().getDirection() == "in")
#set($dummy = $result.add($node))
#end
#end
#end

#getUseCase($targetUseCaseQualifiedName)
#set($targetUseCase = $result)

#if ($documentType == 'contract')
#parse("analysis/servicesContract/servicesContract.txt")
#elseif ($documentType == 'minimal')
#parse ("minimal.txt")
#else
#parse("standard.txt")
#end
```

7.10.5.1.5.2 introduction.txt

```
                    <section>
    <title>Introduction</title>
    <para>
        This is an URDAD compliant document for the requirements analysis and business ↩
            process design
        of the $targetUseCase.name use case. The document shows the requirements and design ↩
             for this
        level of granularity. The details of the services used within the business process ↩
            are shown in
        the analysis and design of the respective use cases for these lower level services.
    </para>
    <para>
          $report.getComment($targetUseCase).body
          The fully qualified name of the use case is $targetUseCase.qualifiedName.
```

```
    </para>
</section>
```

7.10.5.1.5.3 analysis.txt

```
                    <section>
    <title>Analysis</title>
    <para>
        The analysis section specifies the stakeholder requirements, the user work flow,
        the data structures for the exchanged value objects and the service contract with
        the pre- and post-conditions
        and quality requirements. It contains the requirements specification for the  ↩
            current level
        of granularity.
    </para>

#parse("analysis/functionalRequirements/functionalRequirements.txt")
#parse("analysis/userWorkflow/userWorkflow.txt")
#parse("analysis/servicesContract/servicesContract.txt")

</section>
```

7.10.5.1.5.4 functionalRequirements.txt

```

<!--  FUNCTIONAL REQUIREMENTS SPECIFICATION -->
<!-- ****************************************** -->

<section>
    <title>Functional requirements: $targetUseCase.name</title>
    #getDiagramForUseCase($targetUseCase,$urdad_functionalRequirementsDiagram)
    #set($diagram = $result)
    #if ($diagram && $diagram != '')
        #set($diagramId = $diagram.qualifiedName.replaceAll('::','.').replaceAll(' ','_'))
        <para>
                The functional requirements diagram shown below
##                in <xref linkend="$diagramId"/>
                shows the $targetUseCase.name use case, the stake holders who have an  ↩
                    interest in that
                use case and their functional requirements.
        </para>
        #set ($diagramComment = $report.getComment($diagram).body)
        #if ($diagramComment && $diagramComment != '')
                <para>
                    $diagramComment
                </para>
        #end

        <figure xml:id="$diagramId">
            <title>The stakeholder requirements diagram for the $targetUseCase.name use  ↩
                case.</title>
            <mediaobject><imageobject>
                <imagedata contentwidth="150mm" scalefit="1" fileref="$diagram.image"/>
            </imageobject></mediaobject>
        </figure>

        #parse("analysis/functionalRequirements/user.txt")
```

```
        #parse("analysis/functionalRequirements/stakeholders.txt")

       #getMandatoryFunctionalRequirements($targetUseCase)
       #set($mandatoryFunctionalRequirements = $result)
       #getConditionalFunctionalRequirements($targetUseCase)
       #set($conditionalFunctionalRequirements = $result)
       #if (($mandatoryFunctionalRequirements.size() == 0) && ( ↩
           $conditionalFunctionalRequirements.size() == 0))
         <note>
             <title>ERROR: No functional requirements specified for use case</title>
             <para>
                 URDAD requires that there must be at least one functional requirement for  ↩
                     each use case. No functional requirements
                 have been specified for the $targetUseCase.name use case.
             </para>
         </note>
       #else
           #if ($mandatoryFunctionalRequirements.size() > 0)
               #parse("analysis/functionalRequirements/mandatoryFunctionalRequirements.txt ↩
                   ")
            #end

            #if ($conditionalFunctionalRequirements.size() > 0)
               #parse("analysis/functionalRequirements/conditionalFunctionalRequirements. ↩
                   txt")
            #end
        #end
    #else
        <note>
            <title>ERROR: Missing functional requirements diagram</title>
            <para>
                The URDAD process requires that the stake holders and their functional  ↩
                    requirements are specified
            </para>
        </note>
    #end
</section>
```

7.10.5.1.5.5 user.txt

```
                            #getRelationshipsFor($targetUseCase,  ↩
                                $umlAssociationRelationship)
#set($umlAssociationRelationships = $result)

<section>
    <title>The user role</title>
    <para>
        The user role is the role played by those objects which make use of the use case.  ↩
            It is represented by an interface/contract
        which accumulates the contractual obligations of the user itself.
    </para>
    <para>
      #if ($umlAssociationRelationships.size() == 0)
        <note>
          <title>ERROR: No user role specified for $targetUseCase.name use case</title>
          <para>
            URDAD requires that a single user role is assigned to each use case. The user  ↩
                role needs to be represented by
            an interface and there should be an association relationship between the use  ↩
                case and interface/contract
```

```
            representing the user role.
          </para>
        </note>
      #else
          <itemizedlist>
          #if ($umlAssociationRelationships.size() > 1)
            <note>
              <title>WARNING: Multiple user roles specified for $targetUseCase.name use  ↩
                  case</title>
              <para>
                URDAD requires that a single user role is assigned to each use case. The  ↩
                    user role needs to be represented by
                an interface and there should be an association relationship between the  ↩
                    use case and interface/contract
                representing the user role.
              </para>
            </note>
          #end
          #set ($userRole = '')
          #foreach ($umlAssociationRelationship in $umlAssociationRelationships)
              #if ($report.getClientElement($umlAssociationRelationship).getClass().getName ↩
                  ().equals($umlInterfaceClass))
                  #set ($userRole =$report.getClientElement($umlAssociationRelationship))
              #else
                  #if ($report.getSupplierElement($umlAssociationRelationship).getClass(). ↩
                      getName().equals($umlInterfaceClass))
                      #set ($userRole =$report.getSupplierElement( ↩
                          $umlAssociationRelationship))
                  #else
                    <note>
                        <title>ERROR: user not represented by an interface</title>
                        <para>
                            URDAD requires that the user role is represented by an  ↩
                                interface which will accumulate the user responsibilities
                            for the use case.
                        </para>
                    </note>
                  #end
              #end

              #if ($userRole != '')
                <listitem><formalpara>
                  <title>$userRole.name</title>
                  <para>
                      Objects who play the role of a $userRole.name make use of the  ↩
                          $targetUseCase.name use case.
                      #set ($userRoleComment = $report.getComment($userRole).body)
                      #if ($userRoleComment && $userRoleComment != '')
                          $userRoleComment
                      #end
                  </para>
                </formalpara></listitem>
              #end
          #end
        </itemizedlist>
      #end
    </para>
</section>
```

7.10.5.1.5.6 stakeholders.txt

```
                                        #getStakeholdersForUseCase($targetUseCase)
            #set($stakeholders = $result)
            #if ($stakeHolders.size() > 0)
<section>
    <title>Stake holders</title>
    <para>
        The following stake holders have an interest in the $targetUseCase.name use case:
            <itemizedlist>
                #foreach($stakeholder in $stakeholders)

                ## Enforce that stakeholder represented by interface
                #if ($stakeholder.getClass().getName() != $umlInterfaceClass)
                <note>
                    <title>ERROR: Stakeholder not represented by contract/interface</title>
                    <para>
                        URDAD requires that each stakeholder be represented by a contract ( ↩
                            interface). $stakeholder.name
                        is not represented by an interface.
                    </para>
                </note>
                #end
                <listitem><formalpara>
                    <title>$stakeholder.name</title>
                    <para>
                        $report.getComment($stakeholder).body
                    </para>
                </formalpara></listitem>
                #end
            </itemizedlist>
    </para>
</section>
         #end
```

7.10.5.1.5.7 conditionalFunctionalRequirements.txt

```
                            <section>
    <title>Conditional functional requirements</title>
    <para>
        The following functional requirements need to be addressed only under certain  ↩
           conditions:
        <itemizedlist>
            #foreach($functionalRequirement in $conditionalFunctionalRequirements)
            <listitem><formalpara>
                <title>$functionalRequirement.name</title>
                <para>
                    $report.getComment($functionalRequirement).body
                    This functional requirement needs to be addressed if
                    #foreach ($extendRelationship in $functionalRequirement.extend)
                    #if($report.getSupplierElement($extendRelationship) == $targetUseCase)
                    <literal><![CDATA[$extendRelationship.condition.specification.body]]></ ↩
                        literal>.
                    #end
                    #end
                    #getStakeHolders($functionalRequirement)
                    #set($stakeHolders = $result)
                    #if ($stakeHolders.size() > 0)
                    This is required by
```

```
                    <itemizedlist>
                        #foreach ($sh in $stakeHolders)
                        <listitem><para>$sh.name</para></listitem>
                        #end
                    </itemizedlist>
                    #else
                    <note>
                        <title>ERROR: Missing stakeholder specification</title>
                        <para>
                            URDAD requires that each functional requirement is linked to ↩
                                the stakeholder(s) who require it via a requires
                            relationship. The $functionalRequirement.name requirement is ↩
                                not linked to a stakeholder via a requires relationship.
                        </para>
                    </note>
                    #end
                </para>
            </formalpara></listitem>
            #end
        </itemizedlist>
    </para>
</section>
```

7.10.5.1.5.8 mandatoryFunctionalRequirements.txt

```
                        <section>
    <title>Mandatory functional requirements</title>
    <para>
        The following functional requirements need to be addressed for any success scenario ↩
             of the use case,
        <abbrev>i.e.</abbrev> for any scenario where the user obtains the value from the ↩
            use case.
        <itemizedlist>
            #foreach($functionalRequirement in $mandatoryFunctionalRequirements)
            <listitem><formalpara>
                <title>$functionalRequirement.name</title>
                <para>
                    $report.getComment($functionalRequirement).body
                    #getStakeHolders($functionalRequirement)
                    #set($stakeHolders = $result)
                    #if ($stakeHolders.size() > 0)
                    This is required by
                    <itemizedlist>
                        #foreach ($sh in $stakeHolders)
                        <listitem><para>$sh.name</para></listitem>
                        #end
                    </itemizedlist>
                    #else
                    <note>
                        <title>ERROR: Missing stakeholder specification</title>
                        <para>
                            URDAD requires that each functional requirement is linked to ↩
                                the stakeholder(s) who require it via a requires
                            relationship. The $functionalRequirement.name requirement is ↩
                                not linked to a stakeholder via a requires relationship.
                        </para>
                    </note>
                    #end
                </para>
```

```
            </formalpara></listitem>
            #end
        </itemizedlist>
    </para>
</section>
```

7.10.5.1.5.9 userWorkflow.txt

```
<!--  USER WORKFLOW SPECIFICATION -->
<!-- ******************************** -->
#getDiagramForUseCase($targetUseCase,$urdad_userWorkflowDiagram)
#set($diagram = $result)
#if($diagram && ($diagram != ""))
#set($diagramId = $diagram.qualifiedName.replaceAll('::','.').replaceAll(' ','_'))

    <section>
        <title>User work flow: $targetUseCase.name</title>
        <para>
            The user work flow diagram shown in
            <xref linkend="$diagramId"/>
            specifies how users interact with the service provider in the context of
            making use of the $targetUseCase.name use case. It shows the messages exchanged
            in the various scenarios.
        </para>
        #set($diagramComment = $report.getComment($diagram).body)
        #if ($diagramComment && ($diagramComment != ""))
            <para>
                $diagramComment
            </para>
        #end

        <figure xml:id="$diagramId">
            <title>The user work flow for the $targetUseCase.name use case.</title>
            <mediaobject><imageobject>
                <imagedata contentwidth="150mm" scalefit="1" fileref="$diagram.image"/>
            </imageobject></mediaobject>
        </figure>

        #parse("analysis/userWorkflow/services/services.txt")

    </section> ## end of user workflow section
    #*
#else
    <note>
        <para>
            The user work flow was not specified.
        </para>
    </note>
    *#
#end
```

7.10.5.1.5.10 services.txt

```
                            #getDiagramForUseCase($targetUseCase,$urdad_userWorkflowDiagram ↩
                                )
```

```
#set($userWorkflowDiagram = $result)
#getRequestedServices($userWorkflowDiagram.owner)
#set($services = $result)
<section>
    <title>Service request specifications</title>
    <para>
        This section specifies the services requested in the user work flow including
        the data structures for the inputs and outputs (<abbrev>i.e.</abbrev>
        requests and responses) for each service.
    </para>

    #foreach($service in $services)

        #parse("analysis/userWorkflow/services/service/serviceNoParameters.txt")

    #end  ##foreach(service)

</section> ## end of services section
```

7.10.5.1.5.11 servicesContract.txt

```
#getDiagramForUseCase($targetUseCase,$urdad_serviceContractDiagram)
#set($diagram = $result)
#if($diagram && $diagram != '')
#set($diagramId = $diagram.qualifiedName.replaceAll('::','.').replaceAll(' ','_'))

    <section xmlns="http://docbook.org/ns/docbook" xmlns:xi="http://www.w3.org/2001/ ↩
        XInclude">
        <title>Service contract: $targetUseCase.name</title>
        <para>
            The service contract specification diagram shown below
##            in <xref linkend="$diagramId"/>
            specifies the contract (interface) name, the service with the request and  ↩
                response objects, the pre- and post-conditions
            and the quality requirements for the service.
        </para>
        <para>
            The pre-conditions are those conditions under which the service may be refused  ↩
                without being in breach of services contract.
            The post-conditions are those conditions which must hold true once the service  ↩
                has been rendered.
            The quality requirements are the non-functional requirements.
        </para>

        #set($diagramComment = $report.getComment($diagram).body)
        #if ($diagramComment && ($diagramComment != ''))
            <para>
                $diagramComment
            </para>
        #end

        <figure xml:id="$diagramId">
            <title>The service contract for the $targetUseCase.name use case.</title>
            <mediaobject><imageobject>
                <imagedata contentwidth="150mm" scalefit="1" fileref="$diagram.image"/>
            </imageobject></mediaobject>
        </figure>
```

```
#getRealizingServices($targetUseCase)
#set($services = $result)
#foreach($service in $services)
    <para>
        The service realizing the services contract is $service.getName().
        #set($serviceComment = $report.getComment($service).body)
        #if ($serviceComment && $serviceComment != '')
        $serviceComment
        #end
    </para>

#parse("analysis/userWorkflow/services/service/parameters/parameters.txt")

#parse("analysis/userWorkflow/services/service/returnValue/returnValue.txt")

#end

    </section>
#else
  <section>
      <title>ERROR: Missing service contract specification</title>
      <para>
          URDAD requires the specification of a services contract showing the contract ( ↩
              interface) with the services, their inputs
          and outputs and their quality requirements.
      </para>
  </section>
#end
```

7.10.5.1.5.12 design.txt

```
                    <section>
    <title>Technology neutral process design</title>
    <para>
        This section specifies the technology neutral design realizing the use case  ↩
            requirements. In particular, it specifies which functional requirement
        is assigned to which services contract, how the business process is assembled from  ↩
            these services and the collaboration context containing
        the services required from the various service providers together with the inputs  ↩
            and outputs for each service and the message paths to the
        service providers.
    </para>

    #parse("design/responsibilityAllocation/responsibilityAllocation.txt")

    #parse("design/businessProcessSpecification/businessProcessSpecification.txt")

    #parse("design/collaborationContext/collaborationContext.txt")

</section>
```

7.10.5.1.5.13 responsibilityAllocation.txt

```
                        <!-- RESPONSIBILITY ALLOCATION -->
<!-- ***************************** -->

#getDiagramForUseCase($targetUseCase,$urdad_responsibilityAllocationDiagram)
```

```
#set($diagram = $result)

#if ($diagram && $diagram != '')
#set($diagramId = $diagram.qualifiedName.replaceAll('::','.').replaceAll(' ','_'))
    <section>
        <title>Responsibility identification and allocation: $targetUseCase.name</title>

        <para>
            <xref linkend="$diagramId"/> shows
            the grouping of functional requirements into responsibility domains. Each ↩
                responsibility domain
            is assigned to a separate services contract.
        </para>
        #set ($diagramComment = $report.getComment($diagram).body)
        #if ($diagramComment && $diagramComment != '')
        <para>
            $diagramComment
        </para>
        #end

        <figure xml:id="$diagramId">
            <title>The responsibility allocation diagram for the $targetUseCase.name use ↩
                case.</title>
            <mediaobject><imageobject>
                <imagedata contentwidth="150mm" scalefit="1" fileref="$diagram.image"/>
            </imageobject></mediaobject>
        </figure>

        #parse("design/responsibilityAllocation/controller/controller.txt")
        #parse("design/responsibilityAllocation/serviceProviderContracts/ ↩
            serviceProviderContracts.txt")
    </section>
#else
    <section>
        <title>ERROR: missing responsibility allocation diagram</title>
        <para>
            URDAD requires that a responsibility allocation diagram is provided which shows ↩
                how functional
            requirements are grouped into responsibility domains and the services contract ↩
                to which each
            responsibility domain is assigned to.
        </para>

    </section>
#end
```

7.10.5.1.5.14 businessProcessSpecification.txt

```
                        <!-- BUSINESS PROCESS SPECIFICATION -->
<!-- ********************************* -->

## TODO:
## - Actually should get behaviour diagram assigned to service realizing use case
## - Lots of validation and information extraction


#getDiagramForUseCase($targetUseCase,$urdad_businessProcessDiagram)
#set($diagram = $result)

#if ($diagram && $diagram != '')
```

```
#set($diagramId = $diagram.qualifiedName.replaceAll('::','.').replaceAll(' ','_'))

    <section>
        <title>Process specification: $targetUseCase.name</title>
        <para>
            <xref linkend="$diagramId"/> shows
            how the $targetUseCase.name service is assembled from services sourced from the ↩
                 service providers to whom the
            functional requirements have been assigned.
        </para>
        #set ($diagramComment = $report.getComment($diagram).body)
        #if ($diagramComment && $diagramComment != '')
            <para>
                $diagramComment
            </para>
        #end

        <figure xml:id="$diagramId">
            <title>The business process specification diagram for the $targetUseCase.name  ↩
                use case.</title>
            <mediaobject><imageobject>
                <imagedata contentwidth="150mm" scalefit="1" fileref="$diagram.image"/>
            </imageobject></mediaobject>
        </figure>
    </section>

 ## %%%%%%%%%%   #parse("design/businessProcessSpecification/englishMapping/englishMapping. ↩
     txt")

#else
    <section>
        <title>ERROR: Missing process specification</title>
        <para>
            URDAD requires that a (business) process is specified for the use case and that ↩
                 the process is assigned as the behaviour for
            the service realizing the use case.
        </para>
    </section>
#end
```

7.10.5.1.5.15 responsibilityAllocation.txt

```
<!-- COLLABORATION CONTEXT -->
<!-- ************************** -->

#getDiagramForUseCase($targetUseCase,$urdad_collaborationContextDiagram)
#set($diagram = $result)
#if ($diagram && $diagram != '')
    #set($diagramId = $diagram.qualifiedName.replaceAll('::','.').replaceAll(' ','_'))

    <section>
        <title>Collaboration context: $targetUseCase.name</title>

        <para>
            <xref linkend="$diagramId"/> shows
            the services required for the $targetUseCase.name use case from the different  ↩
                service providers
            as well as the required message paths through which the services can be  ↩
                requested.
```

```
        </para>
        #set ($diagramComment = $report.getComment($diagram).body)
        #if ($diagramComment && $diagramComment != '')
        <para>
            $diagramComment
        </para>
        #end

        <figure xml:id="$diagramId">
            <title>The collaboration context diagram for the $targetUseCase.name use case ↩
                .</title>
            <mediaobject><imageobject>
                <imagedata contentwidth="150mm" scalefit="1" fileref="$diagram.image"/>
            </imageobject></mediaobject>
        </figure>
    </section>
#else
    <section>
        <title>WARNING: Missing collaboration context</title>
        <para>
            URDAD recommends that you show the services required from the various  ↩
                responsibility domains (services contracts)
            in a collaboration context diagram (this is a class diagram).
        </para>
    </section>
#end
```

7.11 Implementation mappings

The implementation mappings to a particular SOA technology suite (such as a JBI-based EJB) would be quite technology-specific. Thus, while the technology neutral business process design is usually done by business analysts, the implementation mappings are usually done by the technical team, with guidance from the architect.

Often a business process is realised across a combination of manual (human) work flow steps, services provided by external service providers and automated processing steps executed within systems. The services contracts coming out of the technology neutral business process design can be used as a basis for the service provider contracts which are either realised by external service providers or by business units hosted within the organisation. The implementation mapping of such work flow steps may require training certain staff members to execute them.

Often, however, the technology mapping may result in mapping the technology neutral design onto

- a realisation using current systems, with perhaps some additional development,
- buying technology components and customising them, or
- developing an entire system hosting the various services from scratch.

MDA (Model-Driven Architecture) tools aim to automate this process fully in the near future.

7.11.1 Notes on mapping onto a Service-Oriented architecture (SOA)

7.11.1.1 Services Contracts

Services contracts are mapped to WSDL contracts which are typically abstract, i.e. without protocol binding definitions. The structure of the exchanged value objects are mapped to XML Schema definitions.

7.11.1.2 Workflow controllers

The service which plays the role of workflow controller at a certain level of granularity typically needs to invoke other, lower-level services. For this, we typically map it to a technology which could be used for service orchestration, such as *WS-BPEL* or a *Java-based* service.

The design for a workflow controller often involves dynamically routing requests to lower-level service providers based on certain criteria (message contents, context, environmental state). This implies a content-based router, often implemented in a rules or workflow technology such as Apache Camel or Drools.

Note
WS-BPEL has practical shortcomings in terms of data manipulation / querying when dealing with object-oriented XML structures, which render it practical only for relatively simple, course-grained and high-level activities, forcing the developer to outsource logic to a greater number of low-level services (implemented in, say, Java or XSLT) than what may have been otherwise necessary).

7.11.1.3 Service Adaptors

An URDAD-based design often includes, or implies, adaptors between services. These adaptors often primarily involve message transformation, to which a technology such as XSLT is very suited. If an adaptor contains a large amount of logic, or requires multiple helper services, it could be implemented as a special form of workflow controller.

7.11.1.4 Low-level business services

Individual services that perform atomic tasks (such as computation, database persistence, etc) are often implemented in a technology which has access to the necessary support technologies, such as Java (and often in a container such as EJB).

7.12 Summary

The set of accepted design principles which are seen as required characteristics of a good design can be supported by a set of design activities through which these design principles are realized. URDAD defines an algorithmic design process which incorporates these design activities. It generates a technology neutral business process design in the form of services contracts for each level of granularity together with the business process for that level of granularity. URDAD can be embedded within a model driven development process where the technology neutral business process design is ultimately mapped onto one's choice of implementation architecture and technologies.

Chapter 8

Index