KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

Politechnika Wrocławska

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

ROZWÓJ POTENCJAŁU I OFERTY DYDAKTYCZNEJ POLITECHNIKI WROCŁAWSKIEJ

Wrocław University of Technology

Computer Engineering

Zbigniew Huzar

INFORMATION SYSTEMS MODELLING AND ANALYSIS

Wrocław 2011

Projekt współfinansowany ze środków Unii Europejskiej w ramach
Europejskiego Funduszu Społecznego

Wrocław University of Technology

Computer Engineering

Zbigniew Huzar

INFORMATION SYSTEMS MODELLING AND ANALYSIS

Wrocław 2011

Reviewer: Lech Tuzinkiewicz

ISBN 978-83-62098-94-1

Published by PRINTPAP Łódź, www.printpap.pl

Preface

The aim of the text-book is to present the Unified Modeling Language as a contemporary standard in software development process. Modeling is the most important paradigm used by modern software development methodologies [6]. The UML should not be considered as the only modeling language but also as a base for fundamental notions in an object-oriented approach to software development.

In the presentation of the UML, we try to put stress on the relationship between UML models and the modeled domain. The correct and complete definition of the semantics of the UML models is crucial for effective design of an information system which will satisfy the requirements of its future users.

The contents of the text-book is a review of all UML diagrams. Due to limitations of its size, not all diagrams are discussed in detail. The strongest emphasis is placed on class diagrams, not only because of their widespread use, but also because the knowledge and understanding of the concepts related to these diagrams allows for self-understanding of the other diagrams.

Software development modeling is supported by many software tools, which may be used for editing and analysing of UML models. Almost all diagrams presented in the text-book were prepared by the public domain version of astah* community system.

There are many valuable sources for the UML description. Except the basic document specifying UML standard [3], the books [3], and especially [5] may be strongly recommended. The books [1], [2], [7] are related to problems concerning the UML application. For the students interested in how to get a certificate of UML knowledge, book [8] is recommend.

1. Introduction

1.1. Origins of software engineering

The origins of software engineering occured at the end of the 60's, when demands for the software needed to manage banks, airlines, production lines, etc began to rise. During this period, computer science started to leave the research laboratories, creating the prospect of effective support to solve all conceivable problems. The clash of these hopes, with the practice, proved to be very painful. Most large-scale projects ended in failure. Parts of the projects could not even be brought to an end, and a lot of completed projects did not meet users' expectations.

A set of negative phenomena, defined as the software crisis – according to some opinions, as a chronic illness – continues to this day. It seems that this crisis is starting to be slowly overcome, however, some statistics are still unsatisfactory, for example, most projects exceed the time or budget, and about 25% of the projects are not completed.

It is worth it to recall two major causes of the crisis in software design: the difficulty in precise definition of requirements of the system by its users, and mastering the complexity of software projects by the manufacturer.

The complexity of software projects is due to many reasons that lie both on the side of the user and the software developer. For example:

- design of a useful, even middle-scaled software system is related with a large number of inter-related aspects and problems;
- software development process is complex because of the need of different techniques, methodologies, and supporting programming tools;
- the management of the developing process –organization and control of project teams, effective exploitation of human and material resources are sources of separate problems.

In such a complex situation, precise communication between participants in the process of software development – the future software users and developers – is extremely important. Using only the natural language leads to obvious problems. Software requirements formulated at the beginning of a project are often vague, incomplete and sometimes contradictory which results in higher costs and extra realization time.

Attempts to overcome the software crisis were accompanied by a lot of ideas. The first ideas that formed the so-called structural approach came from the 70's. The end of the 80's started so

called object-oriented approaches, which is currently the dominant approach. Over the past two decades, software engineering has become a mature discipline, being able to cope with contemporary challenges. However, software engineering is still very young when compared to e.g. the engineering of machines or building construction. This software is an essential part of any information system, and cost of software is typically much higher than the cost of other system components. It follows from the fact that the software is recognized as one of the most complex products of human intellect.

The term software engineering was introduced by BW Boehm in 1968, but only since the early 80's, software engineering has started to shape up as a separate specialty of computer science. IEEE Computer Society – one of the oldest and most influential computer societies – defines software engineering as the application of a systematic, disciplined, measurable approach to production, operation and maintenance of software.

In the early 90's, IEEE Computer Society and ACM (Association for Computing Machinery), took action to determine the extent of knowledge, issues and techniques related to software engineering. This resulted in an elaboration of the document the Software Engineering Body of Knowledge (SWEBOK). This document defines the following specific software engineering areas:

- software requirements,
- software design,
- software construction,
- software testing,
- software maintenance,
- software configuration management,
- software engineering management,
- software engineering process,
- software engineering tools and methods,
- software quality.

1.2. Software Life Cycle

The problems of software engineering are usually presented on the background of software life cycle, namely the set of activities associated with the development and usage of software – starting from the decision of its development, through design, implementation, exploitation, until its withdrawal from use. Software life cycle is divided into stages and is the base for definition of software development methodologies, in order to obtain high-quality software product.

The group of traditional software life cycle models includes the classic, waterfall model and its variants, e.g. spiral and incremental-evolutionary models. In fact, the elements of the classic model can be found almost in all other models in the software life cycle. This model distinguishes the following phases of software life (see Fig. 1.1):

- business analysis,
- requirement analysis ,
- design,
- implementation,
- acceptance testing,
- installation,
- maintenance,
- withdrawal from use.

These phases – in principle – are executed sequentially. Results (artifacts), e.g. documents, models, programs, etc., obtained in one phase are the basis for the activities in the next phase. In practice, the phases overlap, and very often instead of going to the next phase they return to the previous phase. The reason for the return may be inconsistency or incompleteness in the artifacts that came with the previous phases.

Business analysis is associated with an organization or institution for which the system is to be built. In each place we have a circulation and processing of miscellaneous data. Data takes different forms, e.g. paper or electronic documents, are stored in various forms, e.g. workbooks, paper, archives, etc. Documents are processed and new documents are created. The processing may be performed by humans or by various technical means, including programming tools.

There are two aims of the business analysis. Its first aim is to describe how the interesting fragment of reality is structured and how it behaves. Its second aim is to recognize what the general expectations of the future information system are, and indication of that part of the reality in which the system has to be placed. For example, it is expected that the new system:

- will replace existing, inefficient software system, for example, when the current system cannot keep up with processing a large number of transactions;
- will improve the circulation of information;
- will raise competitiveness in customer service, achieved not only by the clarity and quality of services, but primarily by the prevalence and time of access;

- will modify the existing system taking into account changes in legislation, for example related to the protection of personal data, tax laws, etc.

The analysis usually reveals business organizational changes that have to be carried out within the institution, to be ready for adoption and operation of future information system.

Requirements analysis is to identify all these aspects of reality that may affect functionality and quality of the expected system. This phase is extremely important in view of further consequences. The first consequence deals with financial and legal aspects: well-defined requirements are needed to conclude a contract between the software user and its developer. The second one is related to the acceptance of the final software product – only precisely defined software requirements are the base for trouble-free clearance of the project.

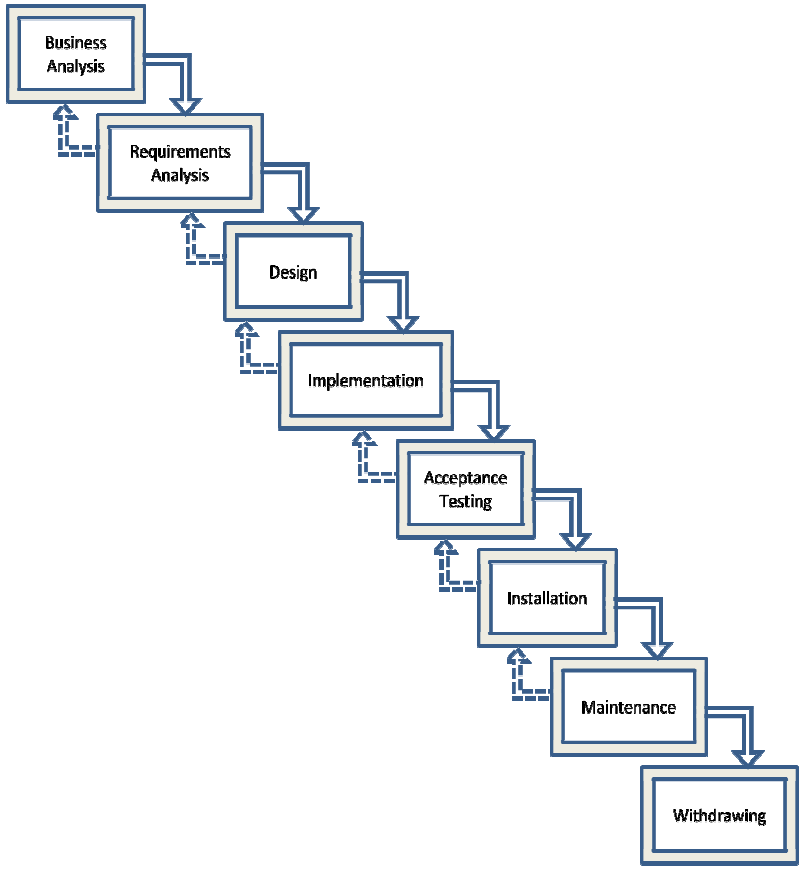


Fig.1.1. The waterfall model of software life cycle

There are functional and nonfunctional requirements. Functional requirements define the scope of functions (services) performed by the system. Non-functional requirements may relate to the quality of the software product or the product's developing process.

The requirements are usually an object analysis done by the institution involved in the future information system. The first step of the analysis is a feasibility study. Such a study should evaluate the feasibility from different perspectives, for example: from technical, economic, law or organization points of view. Positive evaluation of this feasibility study, allows for the approval of the software development project, allocate necessary resources – budget, personnel etc., and next is the establishing of the plan of its implementation.

In the first place the **design** phase concentrates on establishing an architecture of information system. The architecture is defined by a set of the system's components and the relations between them. There are hardware components – nodes, e.g. computers, communication lines, and information components such as programs, databases, software libraries, which are deployed on the nodes. Next, the design phase involves detailed specifications of individual components of architecture. The specification should clearly specify the type of equipment required, determine the functional requirements for software components in such level of abstraction that enables programmers to write the code in the selected programming language. The purpose of the design phase is such a selection and organization of hardware and software components which ensures that the designed system meets user requirements.

The **implementation** phase involves two steps: coding and testing. Coding is to write programs in a selected programming language, while testing is an experimental checking if the system correctly works. There are two types of testing: unit testing, consisting of checking the correctness of the separate software components, and integration testing, consisting of checking the correctness of the cooperation of all system's components. Programmers have to carry out unit testing of the components before forwarding them to integration testing. Both unit and integration testing is performed in a production environment.

Acceptance testing as the next phase is to verify that the built system works correctly in a user environment, whether it actually meets user's expectations. This testing, in contrast to unit and integration testing, which are verification tests, is a validation testing. There are serious consequences when the acceptance testing fails. A return to earlier phases of software development may be required, and respective activities must be performed again, which results in additional, sometimes very high, costs. Hence the need for special care in the early phases of software development, because the cost of an error made in the phase of requirements analysis is several

orders higher than the cost of the programming error in the implementation phase.

Phase of the **installation** involves the implementation of the realized and positively tested system in a target environment. The installation may involve additional activities, for example, user training.

The **maintenance** phase is associated with the time the software is running in a target environment. The aim of the phase is to correct faults, improve performance or other quality characteristics, and adapt the software to a modified environment.

Presented software life cycle was the background for many software development methodologies which were limited to the following phases:

- requirement analysis,
- design,
- implementation, and
- acceptance testing.

The phases are often referred to as a cycle of software development. Software development methodologies concentrate on roles that are responsible for manufacturing processes and producing artifacts to bring a high quality software product.

Similarly, this model was also the background for project management methodologies which sets out a framework for planning, organizing and managing resources to bring about the successful completion of project goals.

1.3. Models in software development

Software crisis of the late 60's led to seeking ways of its solutions. The first attempts, attributable to the 70's have formed the so-called structural approach. The end of the 80's started another approach, called object-oriented approach that may be regarded as a generalization of the structural approach. Currently, object-oriented approach is considered to be the main paradigm of software development. Another is the paradigm, which emerged clearly in the late 90's, called model-driven software development. The modeling paradigm treats the process of software development as a modeling process, i.e. the process of models construction.

The notion of a model is widely used in science. In the most general sense, a model is anything used in any way to represent anything else – a model is an abstraction of something.

In general, models are used to help us know and understand the modeled domain. A model may be an abstraction of a physical reality or an abstraction of an idea. There are two kinds of models – physical and conceptual.

Examples of physical models are models of airplanes or ships. The model of an airplane may be used in a wind tunnel to examine aerodynamic features of the designed airplane. Similarly, the model of a ship may be in used in a stream basin to examine hydrodynamic features of the designed ship.

Conceptual models have different forms; they may be drawn on paper, described in words, or imagined in the mind. In software engineering we use only conceptual models in all these forms. In further presentations we concentrate on graphic models expressed in the UML (Unified Modeling Language) which is a graphic formal language (its models may be presented on paper or screen), and on models described in words both in natural and artificial languages, while the models imagined in mind will be ignored.

The model has notation (syntax) and meaning (semantics). The notation takes the form of pictures and text, the semantics expresses mapping between the model and the modeled domain.

As a model is a simplification of the modeled domain, this simplification depends on the purpose of modeling. The aim of modeling determines the feature of modeled domain that is interesting for the modeler. All the features should be expressed by the model. The features may be divided into two groups – static (structural) and dynamic (behavioral) features.

Modeled domain may correspond to different models. This possibility is of great practical importance, since it allows the gradual examination of selected aspects of reality and thereby facilitates its analysis and understanding. When we have more than one model of the same modeled domain the problem of their consistency arises. Informally, consistency means that the description of the same element in all models may differ at the most detail. For example, if two different models describe personal data then the data relating to the same person is identical or differs by level of detail.

Software development based on modeling paradigm is referred to as Model-Driven Engineering (MDE). A concrete MDE approach was reflected in the standard MDA (Model-Driven Architecture) developed by the OMG. The main idea of the proposed standard is highlighted in the four areas of software development and their modeling perspectives.

The first area is related to the business modeling, understood as in the previous subsection. Models developed in this area are called CIM (Computation Independent Models). A CIM model describes the fragment of reality abstracting from computing means. The model does not show the details of the structure of the created information system, it acts only as a bridge between the domain experts and software designers. The second area is related to the overall design of the system, abstracting from the eventual realization platform, i.e. hardware with appropriate software

runtime environment. In other words: it is involved with the information system design assuming the existence of a virtual environment. Models created here are called PIM (Platform Independent Model). The third area is related to the detailed system design for a given platform. The created PSM (Platform Specific Model) model must take into account the possibilities and limitations of the chosen platform. Finally, the last, the fourth area is related to coding – programming the system in selected languages. Development process is seen as a sequence of transformations between the models:

$$\text{CIM} \rightarrow \text{PIM} \rightarrow \text{PSM} \rightarrow \text{Code}$$

This is a basic modeling scheme of transformations that are referred to as vertical transformations (transformations between different semantic levels). The real software development process is much more complex, because it also allows transformations within each of the model types. Transformation within CIM, PIM etc. are referred to as horizontal transformations (transformations within the same semantic level). The need for horizontal transformation involves pragmatics. Usually, it is not possible to build a complete model for the area from scratch. Instead of that, first some initial model is elaborated, and next, the model is transformed to its extension or refinement. The model has to be extended if the constructed model does not address all the elements of reality (in the case of CIM), or elements of the preceding model (CIM model for PIM, and PIM for the model PSM) are not sufficient to elaborate its successor model. The model refinement results from the pragmatic rule of step-wise discovery of details.

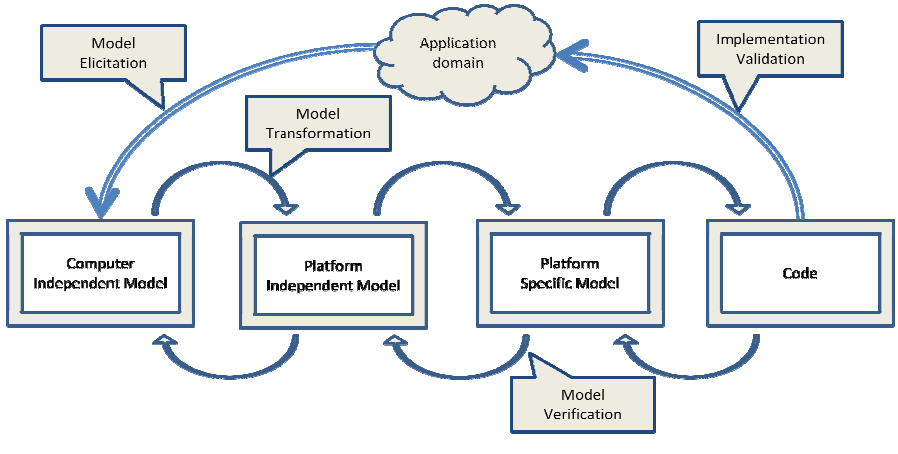


Fig.1.2. Model-based software development cycle in the context of an application domain

The figure 1.2. is an explanation of a model-based approach to software development. An application domain is a starting point for the modeling. The application domain represents a reality in which the future information system will support its function. Construction of a business model describing the application domain is the first, preparatory step in the software development cycle. The business model is prepared by domain analyst and next should be validated by domain experts before it will be a basis for elicitation of information system requirements. On the base of the requirements a model system designer elaborates a design model, which is next transformed into an implementation. After this, the implementation moves to the application domain where it is validated against users' needs.

Further details about MDA approach are given in chapter 12.

2. UML overview

2.1. Pinch of history

The UML (Unified Modeling Language) is the modeling language and it is used commonly in software engineering. The UML is a graphical language designed for specification and designing of software systems. It is not a programming language, but it is the language of documenting the process of software development. The last statement is important because permanent lack of complete documentation was long-term headache for numerous projects previously carried out. The UML models that have been created in software development process are at the same time project's documentation. UML is a language only partially formalized – it has formally defined syntax, but its semantics is expressed in natural language (English prose).

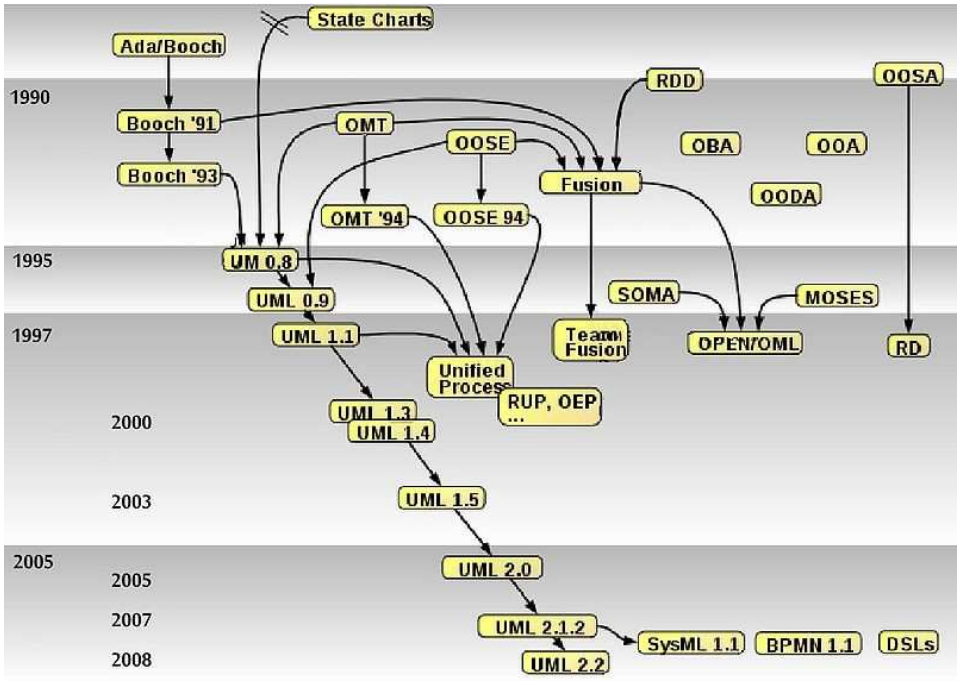


Fig. 1

2.1. UML evolution (Wikipedia: http://en.wikipedia.org/wiki/File:OO-historie.jpg)

The UML has a short but very interesting history. Work on the language began in the mid-nineties, following an initiative by Grady Booch, James Rumbaugh and Ivar Jacobson – creators of the previous object-oriented software development methodologies (Booch Method, OMT, OOSE, Fusion). The work was conducted under the auspices of the OMG (Object Management Group). The OMG is a non-profit organization grouping producers and users of software, and aiming at developing and promoting standards based on object-oriented approach to software development. The language was developed in evolution fashion. The evolution has seen UML through a number of minor new releases (versions 1.0, 1.1, …, 1.5, referenced as UML 1.*x*) into a major release (2) designed to extend UML's growth into new areas. The latest UML 2.2 version was issued in 2008.

The UML development gave rise to elaboration of other modeling languages, e.g. System Modeling Language (SysML), Business Process Modeling Notation (BPMN), and still developed group of so called Domain Specific Languages (DSLs) – see Fig. 2.1.

2.2. Main features

The UML is used to present in a visual form specifications and constructions of the artifacts that are used or produced under system development. It combines techniques from data modeling (entity relationship diagrams), business modeling (work flows), object modeling, and component modeling. It may be used through the software development life cycle, and across different implementation platforms. The UML being a de facto industry standard, aims to be the modeling language for concurrent and distributed systems. It is not a programming language but it is possible to transform UML models into programming (e.g. Java) representation.

The UML is rather large language containing some redundancies. It is reflected in the phrase derived for Pareto principle that knowledge of 20% of the UML is enough to cope with 80% software projects. Nevertheless, the UML has evolved dialects beyond the reach of official standards for such needs as data modeling, business modeling, and real-time development. The designers of UML intentionally gave UML the power for such evolution: UML has extension mechanisms that allow for creative new approaches. The implementation of UML, and especially UML 2, allows flexible language evolution that can keep up with the pace of change in software development.

The UML is semi-formal language as only its syntax is formally defined while its semantics is described in English prose. More precisely, its abstract syntax (free-context grammar) and well-formed rules are formally defined. Peculiarity of the abstract syntax definition is that it is expressed in terms of classes, associations and generalizations that are also UML elements. One can say that

UML syntax is defined by means of the UML. Well-formed rules are expressed in Object Constraint Language (OCL). The OCL is a formal language being a part of the UML standard.

Practical usage of the UML is inseparably tied with the use of supporting programming tools. The tools enable graphical edition, modification and transformation of UML models. They also allow one to create and use repositories with developed models. For the language, the XMI standard of an internal representation of UML models is defined. The standard enables moving models between different supporting tools.

The UML does not recommend how to use the notation it provides in a model. The UML itself offers no process for developing software. Using UML to enhance project success requires skill, talent, and creativity. However, it was designed to be compatible with the leading object-oriented software development methods. Now, UML is used within many software development methodologies, the best known are IBM Rational Unified Process (RUP), and Unified Software Development Process (USDP).

2.3. Review of diagrams

The first problem in modeling is to set a modeled domain and a purpose of modeling. The purpose of modeling appoints a modeling perspective (viewpoint). Selected modeling perspective points out a set of interesting aspects of the modeled domain, and omits entities that are not relevant to this perspective. Any perspective has a different focus, conceptualization, dedication and visualization of what the model is representing. Structural and behavioral perspectives are traditional ones.

The UML incorporates the following perspectives:

- real-world modeling,
- application analysis,
- high-level design,
- implementation specification.

All the perspectives share a common property, namely they are focused on data and on how they are processed. It is unfortunate that UML itself does not contain good ways to declare the perspectives that a model expresses, but modelers should take care to separate different perspectives into different models and to label each model with the perspective that it expresses. The multiple perspectives sometimes cause confusion because the same concepts may be used in different ways to accomplish different purposes (the inter-model consistency).

The next problem is how to express a model based on the selected perspective. Models in UML are presented by diagrams. A model is a set of diagrams. A diagram may be considered as a graph

with labeled nodes and arcs. As a model may represent static and/or dynamic aspects there are two groups of UML diagrams – structure diagrams and behavior diagrams. In each of the groups there are several diagrams as the diagrams represent different artifacts depending on the stage of software development process. The family of all UML diagrams is presented in Fig. 2.2.

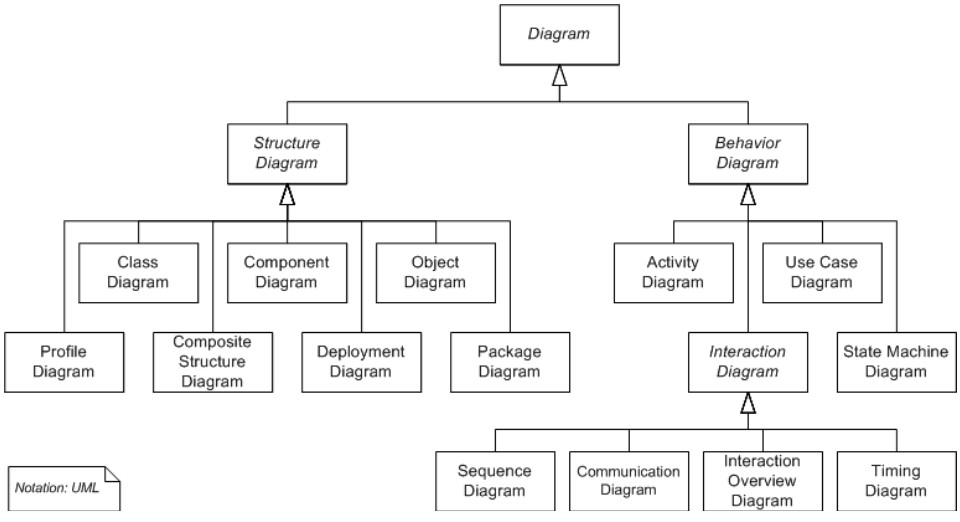


Fig. 2.2. Taxonomy of UML diagrams (Wikipedia: http://en.wikipedia.org/wiki/File:Uml_diagram2.png)

Structural diagrams emphasize the static structure of the system, i.e. which are the basic elements of the system and how they are arranged. They are used extensively in documenting the architecture of software systems.

Behavior diagrams emphasize dynamic behaviors of the system, i.e. how the system's elements are functioning by performing some actions, and how they cooperate with each other via sending and receiving messages.

- Class diagram: describes the structure of a system by showing the system's classes, their attributes, and the relationships among the classes.
- Component diagram: describes how a software system is split up into components and shows the dependencies among these components.
- Composite structure diagram: describes the internal structure of a class and the collaborations that this structure makes possible.

- Deployment diagram: describes the hardware used in system implementations and the execution environments and artifacts deployed on the hardware.
- Object diagram: shows a complete or partial view of the structure of a modeled system at a specific time.
- Package diagram: describes how a system is split up into logical groupings by showing the dependencies among these groupings.
- Profile diagram: operates at the metamodel level to show stereotypes as classes with the <<stereotype>> stereotype, and profiles as packages with the <<profile>> stereotype. The extension relation (solid line with closed, filled arrowhead) indicates what metamodel element a given stereotype is extending.

The last kind of diagrams needs some comment. The metamodel relates to the description of the UML. The notions from the metamodel determine the language (the metalanguage) that is used to describe the UML. Profile diagram is a definition of such a UML extension which is intended to be used for a specific application. Therefore, the extension of the UML has to operate on the metamodel level.

Behavior diagrams emphasize dynamic behavior of the system, i.e. how the system's elements are functioning by performing some actions, and how they cooperate each other via sending and receiving messages.

- Activity diagram: describes the business and operational step-by-step workflows of components in a system. An activity diagram shows the overall flow of control.
- UML state machine diagram: describes the states and state transitions of the system.
- Use case diagram: describes the functionality provided by a system in terms of actors, their goals represented as use cases, and any dependencies among those use cases.

Interaction diagrams, a subset of behavior diagrams, emphasize the flow of control and data among the things in the system being modeled:

- Communication diagram: shows the interactions between objects or parts in terms of sequenced messages. They represent a combination of information taken from class, sequence, and use case diagrams describing both the static structure and dynamic behavior of a system.
- Interaction overview diagram: provides an overview in which the nodes represent communication diagrams.
- Sequence diagram: shows how objects communicate with each other in terms of a

sequence of messages. Also indicates the lifespan of objects relative to those messages.

- Timing diagrams: a specific type of interaction diagram where the focus is on timing constraints.

3. Class and object diagrams

3.1. What is a diagram?

A diagram is a labeled graph. The graph G is defined as a pair G = <V, A>, where V is a set of nodes, and A is a set of arcs, where $A \subseteq V \times V$ is a binary relation on the set of nodes. The nodes and arcs may have labels. Interpretation and notation of the nodes, arcs and labels depend on a type of a diagram.

A class diagram consists of classes and interfaces as nodes, and associations, generalizations and dependency relationships as arcs.

An object diagram consists of objects and interface instances as nodes, and links as arcs.

3.2. Classes and objects

Class diagram describes a modeled system for the static perspective, in terms of classes and relationships among the classes. What is the class and what are the relationships?

In object-oriented programming, the class is a construct that is used as a template to create objects of that class. In modeling, the class is a modeling element that represents a set of entities from the modeled domain, while an object of the class is a modeling element that represents an individual entity from the modeled domain. So, a class represents a concept within the system being modeled. Depending on the kind of model, the concept may be taken from a real-world (for a business model), or it may be taken from a computer platform which is selected to implement information system (for a design model). The class is such a description of all its objects, i.e. its instances, that is focused on attributes, operations, methods and behaviors. Attributes, operations, methods and behaviors are internal classes' features.

This class is a kind of model element called classifiers. Classifiers are model elements that describe behavioral and structural features. Kinds of classifiers include actor, association, behavior, class, collaboration, component, data type, interface, node, signal, subsystem (as a stereotype), and use case. Each classifier has its instances. The class instances are called objects but there is no terminology for instances of other classifiers.

Class syntax in graphical form is presented in Fig. 3.1. The notation is shown as a solid-outlined rectangle with three compartments separated by horizontal lines. The top, obligatory compartment holds the class name and other properties that apply to the entire class. The other two compartments are optional. The middle compartment holds a list of attributes. The bottom compartment contains a list of operations.

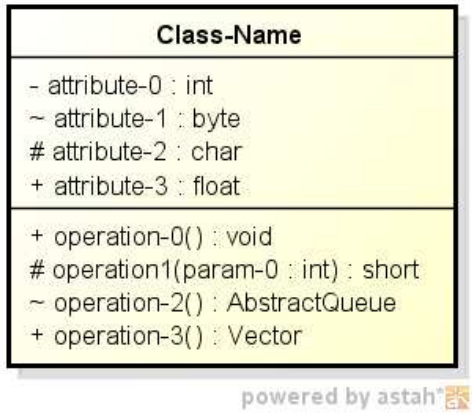


Fig. 2.1. Class notation (graphical syntax)

The attributes and operations have some properties. Part of the properties have visual representation as defined by the default syntax below. The other properties have no graphical representation on the class diagrams but have textual representation in programming tools supporting modeling in UML. The syntax of an attribute declaration is of the form:

[«stereotype»] [visibility] [/] name [: type] [multiplicity] [= initial-value] [{ property-string}]

where an element inside the brackets [] is optional. Similarly, the default syntax of an operation declaration:

[«stereotype»] [visibility] name (parameter-list) [: return-type] [{ property-string }]

Interpretation of the defined syntax elements will be explained by examples in the sequel. Now, we consider a class which represents some reality – see Fig. 3.2.

Fig. 3.2. Example of a class

The class Person has three attributes and one operation. The first two have public visibility (+ preceding their names), the last one has private visibility and has derived value (- / preceding its name). Operation income has public visibility. Public visibility means that any class that can see the

class can also see the attribute. Private visibility means that only the class itself can see the attribute. The value of the attribute age is derived from the value of the attribute dateOfBirth and current date.

The essential question relating to the class Person is its semantics. To define the semantics we have to first define the model to which the class belongs. Let us assume that the class is an element of a business model. It suggests that the class represents some entity from the real world – a set of people. This statement is not sufficient, we have to define the set of people more precisely as it may be a set all people living in the world, a set of people living in Poland, living in one city, working in a given company and so on. So, we have to define precisely the domain of its interpretation. The interpretation of class enables us to interpret the objects of the class. The two instances of the class Person are shown on Fig. 3.3.

A very important demand is to guarantee discrimination of objects belonging to the same class. In business modeling there is a specially important requirement that objects of a class could be distinguished not only by their names, but first of all by valuation of their attributes. The determination of whether the objects of so predefined attributes meet this demand substantially depend on the semantics of the class.

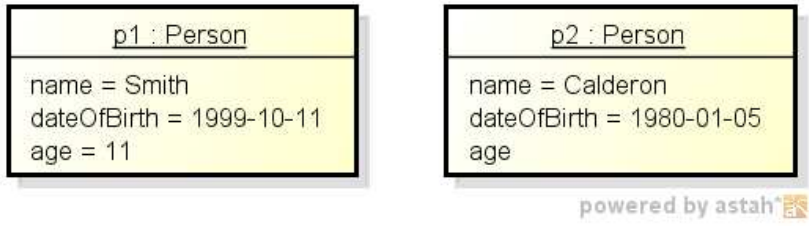


Fig. 3.3. Examples of class instances

Let us notice that the set of all possible instances that might be generated on the base of a class definition is usually infinite. Only some finite subset of this set contains instances that represent entities from the real world. The first set is called the class extent (sometimes called extension), the second set is called the class intent. Fig. 3.4. presents relationships between the class extent, intent and its interpretation domain. Each object belonging to the class intent represents an entity from the interpretation domain.

The attributes and operations are internal properties of a class and its instances. In further, we will use OCL notation for these properties. For example, the expressions:

```
p1.name                -- type String
p2.dateOfBirth         -- type Date
```

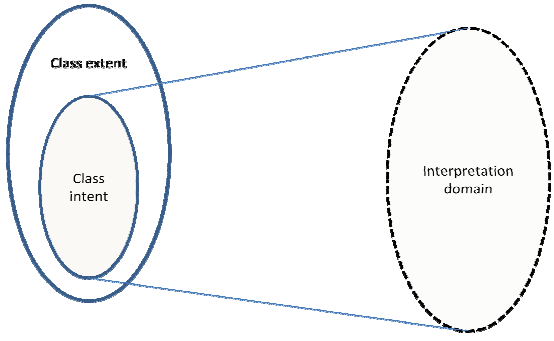


Fig. 3.4. Class intent, extent and its interpretation domain

represent values of respective attributes for the objects p1 and p2 – instances of the class Person.

If C is a name of a class then by Extent(C) and Intent(C) we denote the extent and intent of the class C. If c is an instance of the class C we denote it by c:C, and we say that C is the type of c.

3.3. Binary associations

A binary association is a relationship between two classes. Let us consider Fig. 3.5. presenting an association ab between two classes A and B.

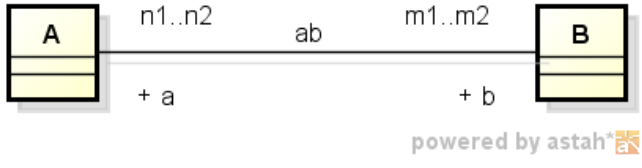


Fig. 3.5. Association notation (syntax)

The association ab has two ends. Each end has a name, a and b, respectively. Each association end has also its multiplicity expressed by subsets of natural numbers defined by intervals n1..n2 and m1..m2, where $n1 \leq n2$, and $m1 \leq m2$. In the case the intervals have not upper constraint, n2 and m2 are represented as *.

The association ab is just a mathematical binary relation of the signature:

$ab \subseteq Extent(A) \times Extent(B)$

It means that the association ab represents a set o pairs of objects of the class A and B. The pairs of such objects are called instances of the associations. If $<a1:A, b1:B> \in ab$, we say that the objects a1:A and b1:B are linked. So, similarly like classes, associations are classifiers.

A particular case we have when the association, named say aa, is defined on the same class, say A. In this case the relation is not reflexive, i.e. $\langle a1{:}A, a2{:}A\rangle \notin aa$.

In further, we assume that we will identify association by the names on its ends. That is justified by the fact that each binary relation may equivalently represented by two functions. The role of such functions will be played by names on association ends. Namely:

$$a : \text{Extent}(B) \to 2^{\text{Extent}(A)}$$

$$b : \text{Extent}(A) \to 2^{\text{Extent}(B)}$$

where the notation 2^X means the powerset of the set X. The function a for a given object $b1 : B$ delivers a subset of such objects of the class A which are linked to b1. Similarly, the function b for a given object $a1 : A$ delivers a subset of such objects of the class B which are linked to a1.

Therefore, the multiplicity constraints that should be satisfied by the relation ab are replaced by the following constraints that should be satisfied by the functions a and b:

$$n1 \le \text{card}(a(b1{:}B)) \le n2$$

$$m1 \le \text{card}(b(a1{:}A)) \le m2$$

where card(X) means the number of elements of a finite set X.

The explanation given above does not describe semantics of an association. As in the case of classes, semantics of an association require references to an explicitly specified interpretation domain. Assuming that an association is an element of a business model, the association has to represent a real world relationship. Fig. 3.6. shows an appropriate example. There are two binary associations. The first one employee-employer represents well known relationship observed in each society and relating people with companies. The names of association ends have clearly defined meaning in plain English. The second binary associations wife-husband represents also well -known relationships relating people of different sex. However, in contrast to employment, the meaning of marriage depends on social and cultural environment.

The class that is associated with other classes, except its internal properties (attributes and operations), has additional external properties. These properties for the classes from Fig. 3.6, and thus objects of the classes, are represented by the following expressions. Let $c : \text{Company}$ then

$$c.\text{emploee}$$

is the new property of the type Set(Person).

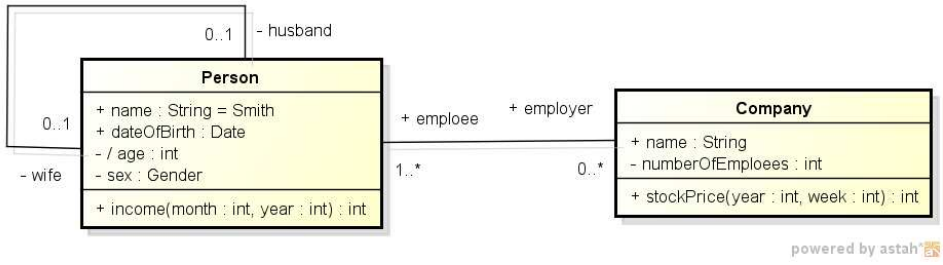


Fig. 3.6. An example of class diagram

The values of this type are all possible objects of the class Person, in other words, the extent of the class. The name of the type is consistent with OCL notation. The property c.employee represents the set of objects from the class Person that are linked with the object c. Similarly, for an object p : Person has the following external properties where each property is of type shown on the left side of a row:

p.employer -- Set(Company)
p.wife -- Set(Person)
p.husband -- Set(Person)

Notice that the only values of the properties p.wife and p.husband are either an empty set or a singleton. This is done because of the multiplicity 0..1 on each ends of the wife-husband association. The association defined on the same class like wife-husband is symmetric. It means that if p1.wife = Set{p2} then p2.husband = Set{p1}, where Set{p1} and Set{p2}, are literals (elements) of the type Set(Person).

Described semantics of binary associations refers to a static aspect. Binary associations have also dynamic interpretation. At run time, the extent of an association is a set of tuples (links), each tuple containing one value corresponding to each end of the association. The linked objects may, at run time, communicate with each other provided that the association ends are declared as navigable. The communication means access to the value or values (depending on the multiplicity of the end) of the class (in general: the type) specified by the end, give one value for each of the other ends. A direction of navigable association is usually shown with an arrowhead on the end of the association path attached to the target class.

Compositions and aggregations are two special cases of binary associations that specify the whole-part relationship. Informally, these associations indicate that objects of one class (composite class) consist of elements that are objects of the second class (part class).

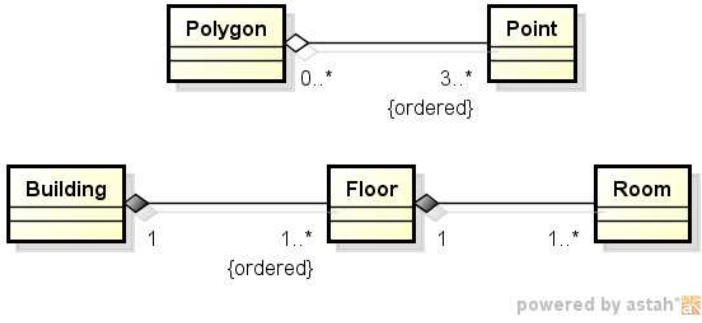


Fig. 3.7. Aggregation (on top), and composition (down) associations

Notation of these associations is presented in Fig. 3.7. On the figure a constraint {ordered} is put on two association ends. It means that if we consider, say, an object p : Polygon then the set of objects of the class Point that linked with p, i.e. p.point (point is a default name of the association end connected to Point) is linearly ordered.

The aggregation and composition relationships are transitive and antisymmetric across all aggregation links, even across those from different aggregation associations. Transitivity means that it makes sense to say, for example, that the Room is part of Building because the Room is a part of the Floor that is part of the Building. Antisymmetry means that there are no cycles in the directed paths of aggregation links. In general, all aggregation and composition associations form graphs without cycles.

Composition is a strong form of aggregation association with strong ownership of parts by the composite and coincident lifetime of parts with the composite. A part may belong to only one composite at a time. Parts may be created after the composite itself. But once created, they live and die with it (that is, they share lifetimes). Such parts can also be explicitly removed before the death of the composite.

Both aggregation and composition may be recursive. It is the case when, for example, these associations are defined from a given class to itself – see Fig. 3.8. The figure contains a class diagram on its left and an object diagram – an instance of the class diagram, on the right.

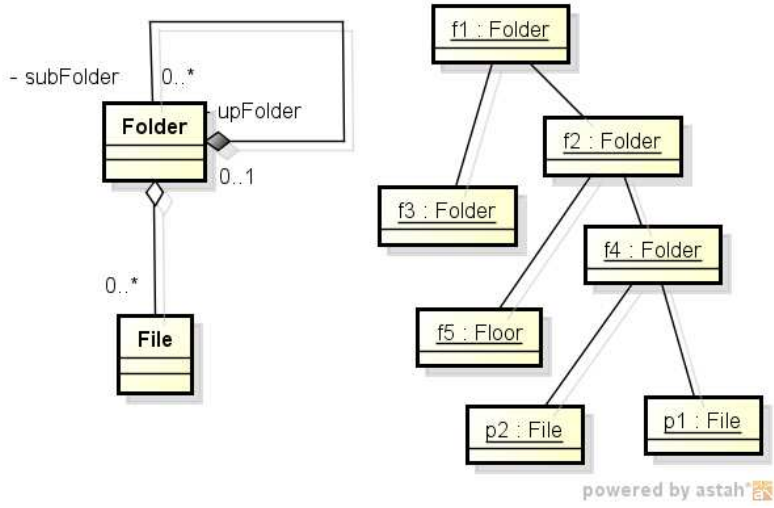


Fig. 3.8. The class diagram and its instance – the object diagram.

3.4. n-ary associations

N-ary association is an association among three or more classes. Each instance of the association is an *n*-tuple of values, one from each of the respective classes. Let's consider an example of ternary association Registration at Fig. 3.9. If not shown, then by default, the names of association ends are the same as the name of the classes at these ends with the difference that begins with a lowercase letter. Instances of the association are triples:

<s:Student, c:Course, sem:Semester>

The set of the triples should satisfy multiplicity constrains that result from the following interpretation of multiplicity on the association ends. The multiplicity on a given association end determines the potential number of objects at the end for when the objects at the other n–1 ends are fixed. So, for a given s:Student and sem:Semester the multiplicity 0..* means that for this pair of object there exists any number of objects of the class Course.

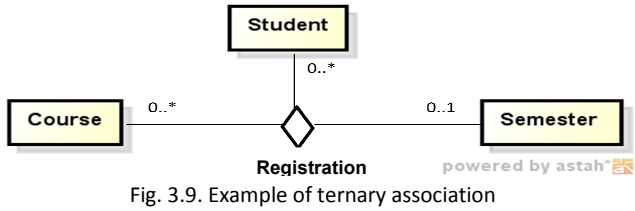


Fig. 3.9. Example of ternary association

3.5. Association classes

An association class is a model element that has both association and class properties. Instances of an association class are links that have attribute values as well as references to other objects. Fig. 3.10 presents an example of the association class. Even though the notation consists of the symbols for both an association and a class, it is really a single model element. The exemplary association class Employment enables to equip each link between Person and Company classes with additional data providing details of the employment of a given person in a given company.

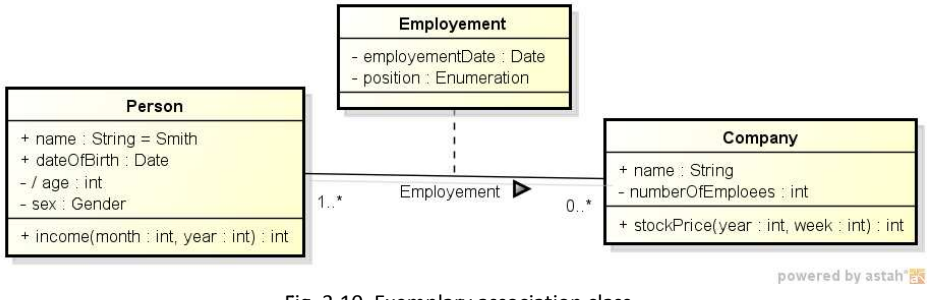


Fig. 3.10. Exemplary association class

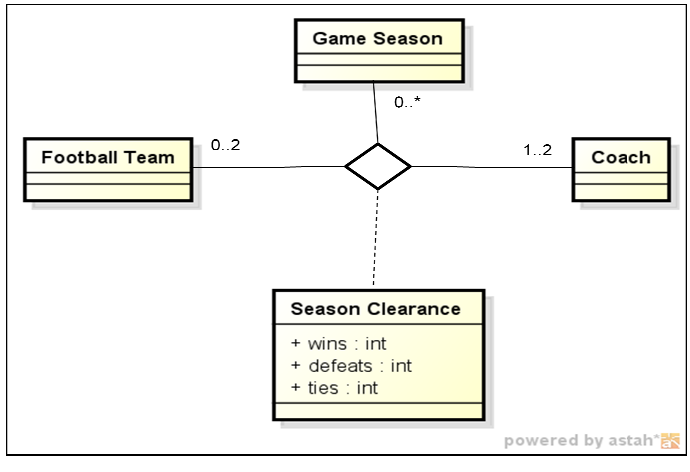


Fig. 3.11. Ternary association class Season Clearance

Because an association class is a class, it may participate in associations itself. However, it may not have itself as one of its participating classes.

An association class may also n-ary – see the example in Fig. 3.11. Instances of the ternary association class are triples <t: Football Team, g: Game Season, c: Coach>. To each triple a

data describing the results achieved by the team t in season g under the guidance of the coach c are attached.

3.6. Bags on association ends

Let us look at Fig. 3.10 again. This class diagram, being a model of some reality, describes employment as one of many possible relationships between people and companies. If you reflect upon this relationship one may ask whether it fully describes the relationship arising from the employment of persons in a given company. After all, it may happen that a given person in a given company can be hired (and exempted) several times. We would expect to have for the same pair <p:Person, c:Company> data regarding each employment act. Unfortunately, it is not possible because the association class (having all properties of an association) allows for the pair for only one link. But an association end may be labeled by the flag {bag} indicating that the object on this end may occur many times in links with object on the opposite association end.

Fig. 3.12 is a modification of the discussed class diagram. Now, the pair <p:Person, c:Company> may occur many times in the association class, where each occurrence regards to separate employment events, i.e. to each occurrence of the pair. So, to each occurrence of this pair detailed data are assigned.

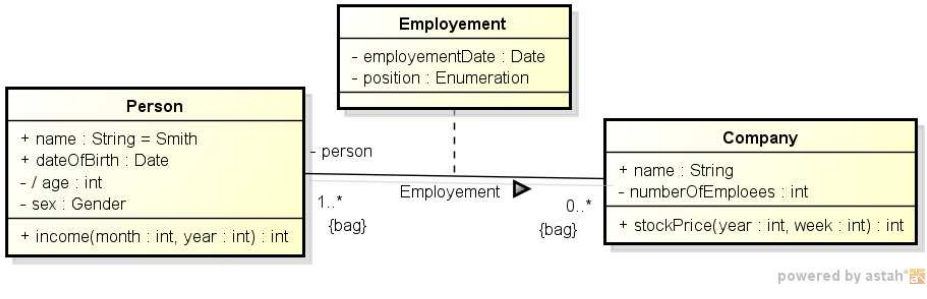


Fig. 3.12. Exemplary association class with {bag} constraints

Let's remember that the notion of a bag is an extension of the notion of a set. Formally, it is defined in the following way. Let X is any set, and Nat the set of natural numbers. A bag over the set X, denoted by Z(X), is defined as a pair <X, no>, where no : X → Nat is a function which for each element of the set X determines the number of instances of the given element.

The definition above, in relation to the association class Employment, should be interpreted in the following way. The multiplicity on the, say, right association end specifies a set of

companies that may be linked to a given person (a counterpart of the set X). The flag {bag} specifies a bag over the set of links (a counterpart of the bag Z(X)).

3.7. Interfaces

At the software design stage classes are assigned to interfaces. An interface is a declaration of services offered or required by objects of a given class. The service includes an access to public properties of a class: attributes, operations, receptions. An interface is a special kind of a class, the class with the stereotype <<interface>>. It is only a declaration of a set of services that should be implemented by other classes. A class may support many interfaces – see Fig. 3.12.

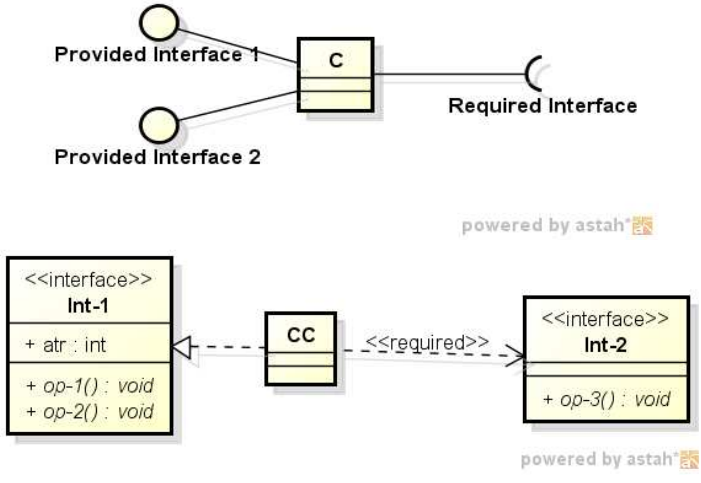


Fig. 3.12. Interface notations: simplified (top) and expanded (bottom) forms.

An interface is a classifier for the externally visible properties. So, the lines between the class C and provided and requested interfaces (Fig. 3.12, top) are binary associations. The dashed closed arrow (Fig. 3.12 bottom) means that the provided interface Int-1 is implemented by the class CC, while the dashed open arrow means that Int-2 is the interface required by the class CC.

The examples above present only typical, the most common form of interfaces. In general, the structure of an interface is more complex as interface may declare not only attributes and operations but also: receptions (signals that anonymous objects can send to an implementing object), constraints (constraints on the invocation of services of an implementing object), and protocols (specification of the order in which services of the implementing object may be invoked).

3.8. Generalizations

Generalization is a taxonomic relationship between a more general element and a more specific element. UML defines generalization as follows: *A taxonomic relationship between a more general classifier and a more specific classifier. Each instance of the specific classifier is also an instance of the general classifier. Thus, the specific classifier indirectly has features of the more general classifier*. Generalization is a relationship between more general (parent, ancestor) and more specialized (child, descendant) classifier of the same kind (classes, interfaces, associations, use case etc.). For classes, the parent is called superclass and the child is called subclass.

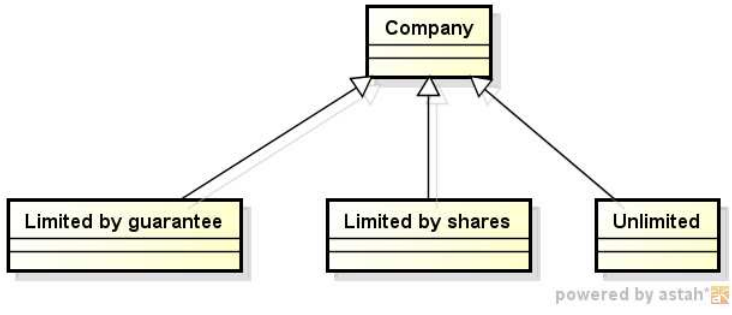


Fig. 3.13. Example of a superclass Company and its subclasses

An example of generalization and of its notation is given in Fig. 3.13. The superclass Company represents *a form of business organization as a collection of individuals and physical assets with a common focus and an aim of gaining profits*. The subclasses represent specialized forms of the organization, namely: Limited by guarantee is a company *which members guarantee the payment of certain (usually nominal) amounts if the company goes into insolvent liquidation, but otherwise they have no economic rights in relation to the company*; Limited by shares is a *company in which the liability of each shareholder is limited to the amount individually invested*; Unlimited is a *company where the liability of members or shareholders for the debts (if any) of the company are not limited*.

Generalization is a transitive, antisymmetric relationship. No directed generalization cycles are allowed. A class may not have itself for an ancestor or descendant. An instance of a classifier is an indirect instance of all of the ancestors of the classifier.

An instance of the subclass is an indirect instance of the superclass and inherits its characteristics. The superclass is a description of a set of (indirect) objects with common properties

over all subclasses. The subclass is a description of a subset of those instances that have the properties of the superclass but that also have additional properties peculiar to the subclass.

The subclass may be defined by putting constraints on its ancestor – see Fig. 3.14. Each instance of the subclass Square is the instance of the superclass Rectangle which satisfies the constraint {width = length}. The constraint is represented in the form of a note attached to the subclass Square.

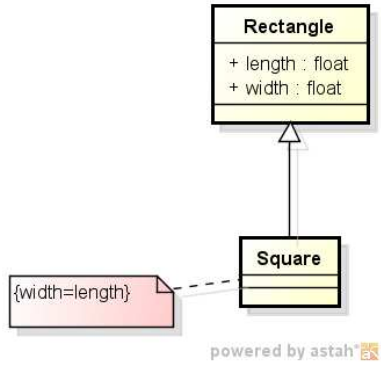


Fig. 3.14. An example of subclass definition that uses the constraints

The instances of a subclass satisfy the substitutability principle attributed to Barbara Liskov that *an instance of a subclass may be substituted for an instance of a superclass*.

In the simplest case, a class has a single parent. In a more complicated situation, a child may have more than one parent. The child inherits structure, behavior, and constraints from all its parents. This is called multiple inheritance – see Fig. 3.15. A child element references its parents and must have visibility to them.

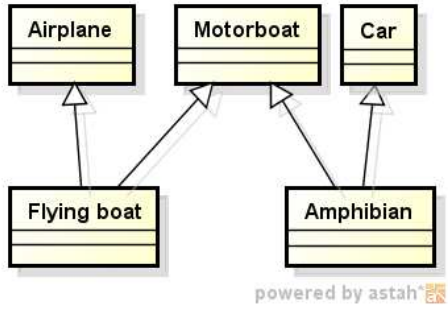


Fig. 3.15. An example of multiple inheritance

The multiple inheritance may bring clash problems if different properties of parent classes have the same name. In such case, additional mechanisms, for example, renaming of properties, should be applied.

Generalization as taxonomic relationship enables classification of a set of entities according to many selection criteria (or dimensions). The set of generalizations that represent a given dimension is called a generalization set. In Fig. 3.16 there is an example of the superclass Person with two generalization sets. Each generalization set has a name (Gender and Profession) and, optionally, may have attributes, in the form of standard constraints, indicating whether the subclasses from the set can overlap and whether they are complete.

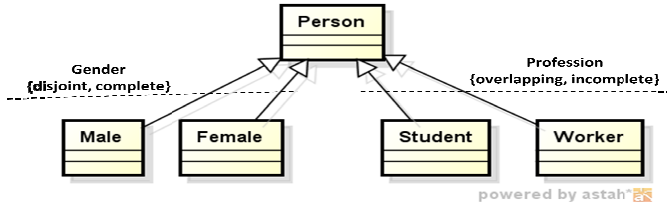


Fig. 3.16. An example of two generalization sets

The complementary pairs of constraints disjoint-overlapping and complete-incomplete have the following meaning:

- disjoint – The subclasses in the set are mutually exclusive. No instance may be direct or indirect instance of more than one of the subclasses.
- overlapping – The subclasses in the set are not mutually exclusive. An object may be an instance of more than one of the subclasses.
- complete – The subclasses in the set completely cover a dimension of specialization. Every instance of the superclass must be an instance of at least one of the subclasses in the set.
- incomplete – The subclasses in the set do not completely cover a dimension of specialization. An instance of the superclass may fail to be an instance of one of the subclasses in the set.

3.9. Structured classes

As you progress in the software development process you need to refine classes to facilitate their implementation. One mechanism of such a refinement is provided by structured classes. A

structured class is the description of the internal implementation structure of a class. The internal structure is expressed in terms of ports, parts, and connectors.

A port is a connection point between a structured class and its environment, or between parts encapsulated by the class. Connections from the outside world are made to ports via provided and required interfaces declared on the ports. A part within a structured class represents an object or set of objects within a contextual relationship. The contextual relationship is expressed by connectors linking parts. A connector is neither a link nor an association. It may be interpreted as a special kind of an association that is applied to a certain context, such as the objects within a classifier.

Fig. 3.17 explains notation: parts are represented in a similar way as objects; ports are squares placed at the edge of parts of the entire class; connectors are links between ports; interfaces are attached to the ports.

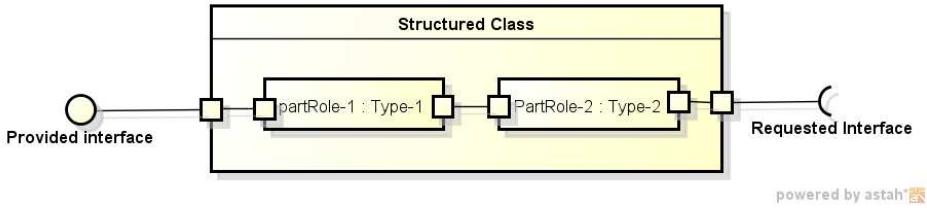


Fig. 3.17. A structured class – notation

The examples of a structured class and an instance of the class are presented in Fig. 3.18 and 3.19, respectively. The structured class Car is a model of the vehicle moving on wheels and driven by the engine. It consists of several parts that are objects of the classes Engine, Axle and Wheel. The part declared by front: Wheel[2.4] represents a family of objects of size from 2 to 4. The parts eng: Engine and drive: Axle[1] are connected by a local link. The instance myCar consists of a set of objects being instance of the declared parts, and the instances of declared connectors.

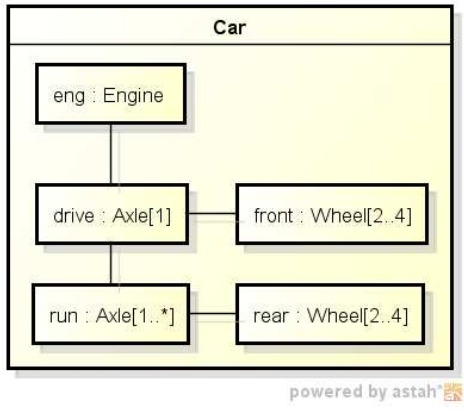


Fig. 3.18. An example of a structured class

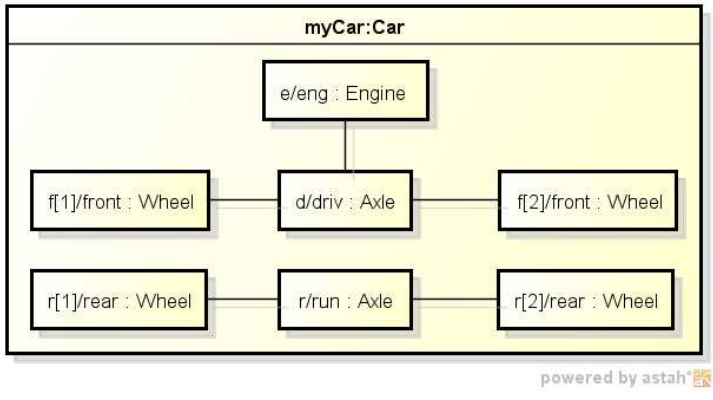


Fig. 3.19. An example of an instance of the structured class

3.10. Constraints – elements of OCL

In practice, to be consistent with reality, class diagrams need constraints. The constraints may be attached to any model element and may be defined informally in a plain language or in a formal language. Below, 'to taste' how constrains may be defined, we consider the class diagram in Fig. 3.20. It is evident that properties, especially the attributes should have consistent values.

Let us consider few examples. The first is concerned with the association child-parent defined for the class Person. The association should express that:

Children are younger than their parents.

This property should be an invariant for the class Person, i.e. it should be satisfied for each object of the class. The invariant may be expressed formally in the Object Constraint Language (OCL) which is a part of the UML standard:

context Person **inv**:

self.child -> notEmpty() **implies** self.child -> forEach(c | c.birthDate < self.birthDate)

The first line of the above constraint specifies the context pointing the class **context** Person. The key word **inv** means that the constraint is an invariant.

The second line represents a formula, in implication form, which should always be true for the model, more precisely, for each intend of the class Person (see p. 3.2).

The line should be interpreted from left to right.

The word self is used to refer to the contextual instance, i.e. to an object of the class Person. It is an expression of the type Person. In OCL all expressions are typed expressions.

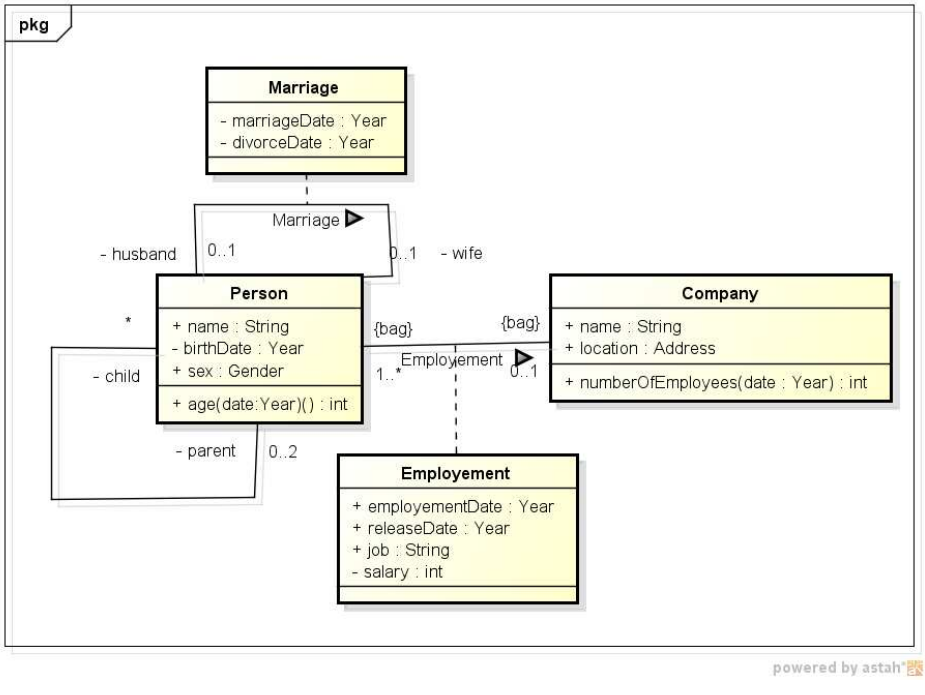


Fig. 3.20. An exemplary class diagram

The expression self.child is expression of the type Set(Person), and represents a set of objects that are linked with the object self (see p. 3.3).

notEmpty() is the OCL standard operation on collections, i.e. on bags, sets and sequences. If applied to a collection yields the true value when the collection is not empty, and false otherwise. Peculiarity of its application is postfix notation, i.e. an argument of the operation precedes the operation. Additionally, the symbol -> is used to separate the argument from the operation name. So, self.child -> notEmpty() is the expression of Boolean type.

The key word **implies** represents one of propositional connectives. There are also other connectives: **and**, **or**, **xor**, **not**, **if-then-else**.

To explain the meaning of the expression on the right side of the implication, i.e.

```
self.child -> forEach(c | c.birthDate < self.birthDate)
```

it is enough to transform the expression to the following traditional mathematical notation:

$$\forall c \in \text{self.child} \bullet (\text{c.birthDate} < \text{self.birthDate})$$

So, going in the opposite direction, the traditional notation:

$$\forall x \in X \bullet P(x)$$

is transformed into the following OCL notation:

```
X -> forEach(x | P(x))
```

Now, let us consider the example of the following constraint:

Marriage shall be entered between grown-up man and woman.

The constraint is an invariant that has in OCL the following form:

```
context Person inv:
    self.wife -> notEmpty()
    implies self.sex = male
        and (self.birthDate – self.marriage.marriageDate) > 18
        and self.wife.sex = female
        and (self.wife.birthDate – self.marriage.marriageDate) > 18
        and self.wife.husband = self
```

The formula specifying the invariant has also an implication form. The left side of the implication has similar meaning to that explained in the previous constraint example. The right side is a conjunction of self explained formulas:

- self.sex = male means that the attribute sex of the object self has the value male.
- (self.birthDate – self.marriage.marriageDate) > 18 means that the difference between birthDate and marriageDate of the object self is greater 18. The navigation

expression self.marriage is the property of the object resulting from the link to association class Marriage. By convention, navigation to an association class has the same name as the class except the first letter of the name is written in lowercase.

- self.wife.sex = female means that the attribute sex of the wife of the object self (self.wife) has the value female.
- (self.wife.birthDate – self.marriage.marriageDate) > 18 is interpreted like the above expression.
- self.wife.husband = self means that navigation from the object self to its wife self.wife and next to the husband of this woman leads back to self.

The next example is concerned with the constraint:

If a given person was employed in a company several times then the periods of working in the company should be disjoint.

Formally, the constraint is expressed by the following OCL expression:

```
context Person inv:
    self.company -> notEmpty()
    implies
        self.employement -> forEach(e1, e2 |
                not(e1=e2)
                and e1.company.name = e2.company.name)
                implies
                Set{e1.employementDate..e1.releaseDate} ->
                intersection(Set{e2.employementDate..e2.releaseDate} -> isEmpty() )
```

As in the previous examples, the formula specifying the invariant has implication form. To understand the implication two explanations are necessary. The first explanation regards the double quantification:

```
self.employement -> forEach(e1, e2 | not(e1=e2) and …).
```

This fragment may be transformed into the traditional notation:

$\forall e1 \in$ self.employement • $\forall e2 \in$ self.employement • (**not**(e1=e2) **and** …)

The second explanation deals with two last lines:

```
Set{e1.employementDate..e1.releaseDate} ->
intersection(Set{e2.employementDate..e2.releaseDate} -> isEmpty()
```

The first of the lines is a literal defining a set of integer values. In general, the definition of a set has a form Set{m..n} where m and n are expressions of the type integer. Set{m..n} defines a set of integers containing values from m to n. Similarly,

Set{e2.employementDate..e2.releaseDate}

is the literal defining a set of integers that is an argument of the operation intersection. The fragment

Set{e1.employementDate..e1.releaseDate} ->
intersection(Set{e2.employementDate..e2.releaseDate}

in traditional mathematical notation has the form

$Set\{e1.employementDate..e1.releaseDate\} \cap Set\{e2.employementDate..e2.releaseDate$

So, the last two lines become a formula which says that the above intersection should be empty.

The last example deals with the definition of the operation numberOfEployees(date:Year) of the class Company. The operation is defined by the following expression:

```
context Company::numberOfEployees(date:Year)
    body self.employement -> select(e |
                        e.employementDate ≤ date and date ≤ e.releaseDate) -> size()
```

Operations in OCL may be defined in several ways, typically by a pair of formulas, called pre- and post-condition. In the example the operation is defined in another way, i.e. by body expression. It is manifested above by the key word **body**. It means that the expression following the key word defines the value which operation returns for a given argument value. The first fragment of this expression self.employement, of the type Set(Employment), defines a set employment for a given object self. The collection operator select, taking the collection self.employement as its argument, defines a subcollection of this collection. All elements e of the collection that satisfy the formula belong to the new subcollection.

e.employementDate ≤ date **and** date ≤ e.releaseDate

i.e. the employments relating to the date being the argument of the defined operation.

Class diagrams represent structural, static view of a modeled domain. The diagrams are universal in the sense that they are used in all stages of software development process starting from business modeling up to implementation. In the stages they play different roles: they model a business domain, next problem domain, and finally, solution domain.

In subsequent phases of software development classes contain more details. For example, an internal structure of classes is considered in general design phase, classes' methods are usually defined within detailed design, however, very often are defined in implementation phase. Therefore, the choice of appropriate level of detail, without going into unnecessary details, is a very important skill of a software developer.

Class diagrams form the background for presentation of behavior of modeled domains.

4. Use case diagrams

4.1. General description

Use case diagrams belong to the category of behavior diagrams, however, they are behavioral-static in nature. Use case diagrams capture the required service – functional requirements – from the users' point of view. In fact, the diagrams present a structure of the required services while a description of the behavioral aspect is delegated first to the textual documentations, and next to other behavioral UML diagrams, usually, activity or sequence diagrams.

The constituent elements of the use case diagrams are: use cases and actors as nodes, and association, generalization, and two stereotyped dependency relationships (<<include>> and <<extend>>).

A use case represents a kind of service. The service is delivered to a user in a result of interaction between the user and the service provider. The interactions is manifested as a sequence of messages exchanged and a sequence of actions performed by the parties participating in realization of the service. There may be many such interactions (scenarios) that end with delivering of expected value. Therefore, in UML use case is described as *the specification of sequences of actions, including variant sequences and error sequences, that a system, subsystem, or class can perform by interacting with outside objects to provide a service of value*. The use case is a classifier. It means that a use case represents a set of scenarios, and its instance is a single scenario. On a diagram a use case is represent by an oval.

An actor represents entities outside a subject (users of an information system) that interact directly with the subject (the information system). The entities outside a system may be a human, a computer system, a device, another subsystem, or other kind of object. An actor participates in a use case or coherent set of use cases to accomplish an overall purpose. An actor characterizes a role played by an outside user or related set of users with respect to a subject. An actor is a classifier. It means that an actor represent a set of users, while its instance – an individual user. On a diagram an actor is represent by a stick man.

Binary associations between actors and use cases may be defined. The associations have the same notation as for classes. Instances of an association are links between individual users and single scenarios – instances of use cases.

Generalizations may be defined only between use cases or between actors. A generalization between use cases means that the child use case inherits the scenarios of the parent and

may add incremental behavior into the parent scenario at arbitrary points. It may also modify some inherited operations and sequences provided the intent of the parent is preserved.

Similarly, a generalization between actors means that child actor inherits the roles and associations to use cases held by the parent actor.

The dependency relationships <<include>> and <<extend>> may be defined between use cases only. The <<include>> relationship from a base use case to an inclusion use case specifies that the behavior defined for the inclusion use case is to be inserted into the behavior defined for the base use case. The base use case can see the inclusion and can depend on the effects of performing the inclusion. In other words, to each scenario belonging to the base use case some scenario from the inclusion use case must be inserted. The insertion point is defined within description of the base use case.

The <<extend>> relationship from an extension use case to a base use case specifies that the behavior defined for the extension use case can be inserted into the behavior defined for the base use case. A description of the base use case defines an extension point and a condition that should be satisfied to enable insertion.

Notation convention on the use case diagrams is explained on Fig. 4.1.

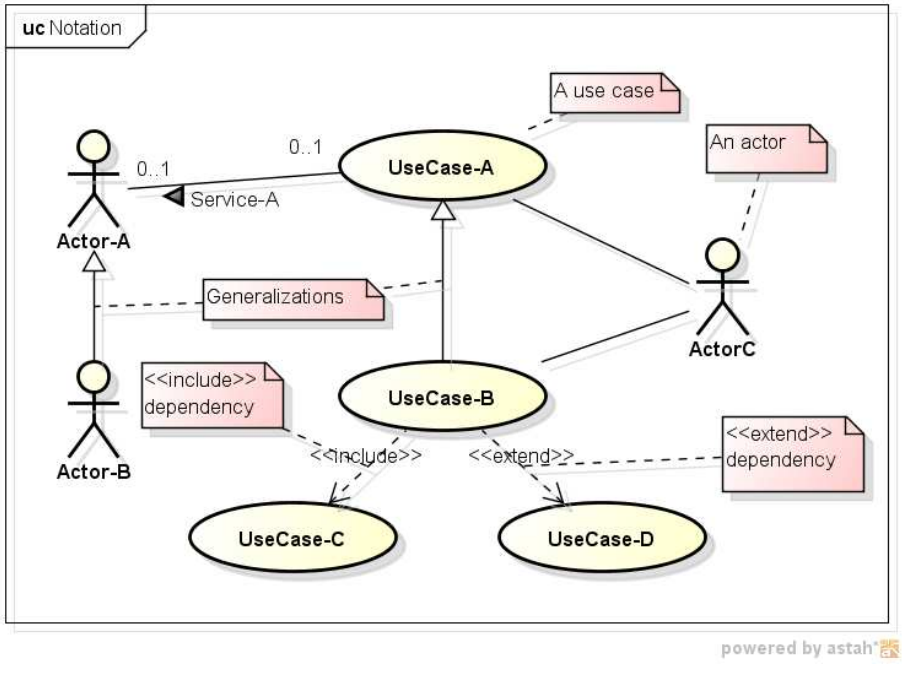


Fig. 4.1. Notation explanation

There are two areas of use case diagrams applications. The first one is concerned with business modeling; use case diagrams are used to model services offered by an institution or company for its clients. The second, the main application area is concerned with requirement analysis; the diagrams are used to define functional requirements for a future information system.

4.2. Use case description

Textual description of a use case is out of UML description. Below there is one of the recommended description schema:

1. Name – naming is a bearing on the way the use case is read and interpreted. Names should be verb-like.
2. Aim – should describe the service offered to a primary actor.
3. Importance (high, medium, low) – this entry is optional but very useful in initial design phase; it enables focus on the most important funcionalities.
4. Complexity (high, medium, low) – this entry is also optional but very useful in initial design phase; it enables estimation of possible efforts during design and implementation phases.
5. Use case initiator – points actor which invokes the use case.
6. Other users – refers to the use cases participating in realization of this use case.
7. Condition of the initiation (preconditions) – indicates if a condition or other use cases need to be completed before the use case begins.
8. Postcondition – is not mandatory; if present it may document effects of the use case realization.
9. Use case relationships – gives references to other use cases that are included, extended and inherited.
10. Scenarios – the most important entry describing a set of scenarios that may be observed during use case realization. The scenarios are divided into three categories:
    - typical scenarios that present the main flow of messages and actions during use case realization that lead to a desired result;
    - alternative scenarios that present the alternative flows but also lead to a desired result;
    - exceptional scenarios that lead to a fail and abortion of the use case realization.

Description of a scenario has a form of a sequence of steps; each step describes an elementary action or a message and, optionally, points to another alternative or exceptional scenarios.

4.3. Use case diagram example

Below a simple example of a use case diagram is outlined. The diagram is a model of functional requirements for e-mail system. The presentation consists of two elements: a concise textual description of e-mail system services, and next of the use case diagram (Fig. 4.1) being a model of the description.

1. E-mail system enables its users to:
   - edit,
   - send, and
   - receive

   messages (letters) in electronic form.
2. Each user has his/her unique name, password, and e-mail box for messages.
3. To begin a work with e-mail system a user has to login entering his/her name and password.
4. A user may check the state of his e-mail box.
5. Before sending e-mail, a user has to define recipient's address and prepare contents.
6. Any received letter may be forwarded to another recipient.
7. An e-mail system checks validity of the address. If the address is not valid the letter is not sent and an exception arises.
8. An e-mail system is managed by an administrator. The administrator may add a new user to the system or remove an old user from the system.

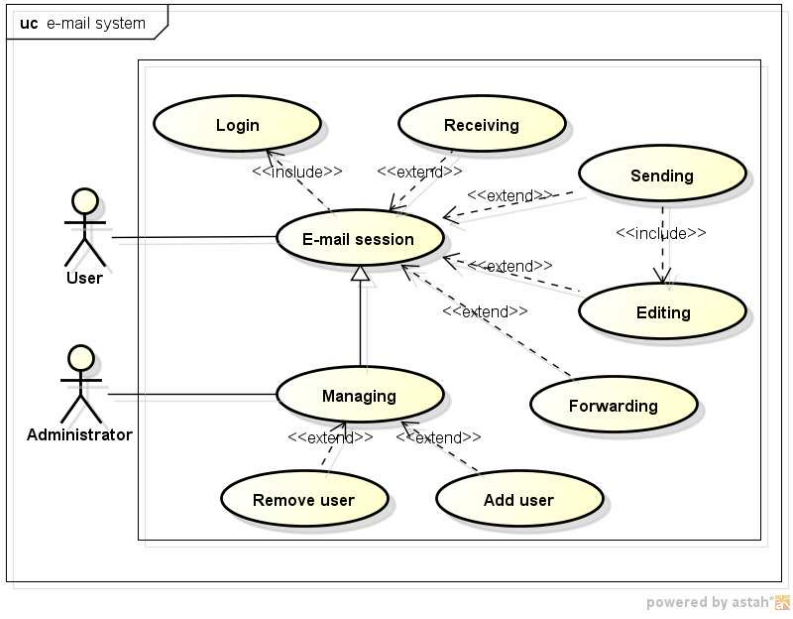


Fig. 4.2. A use case diagram for the e-mail system

5. Sequence diagrams

5.1. Notation and general description

Sequence diagrams are used for pictorial representation of scenarios within a use case. A sequence diagram is a kind of a two-dimensional chart. The vertical dimension is the time axis running from the top to the bottom of the diagram. The horizontal dimension shows the classifier roles that represent cooperating individual objects and actors' instances. The major purpose of sequence diagrams is to show the flow of messages between cooperating entities. They can exist in a descriptor form (describing all possible scenarios) and in an instance form (describing one actual scenario). Fig. 5.1. presented below explains notation of constituent elements of the sequence diagram.

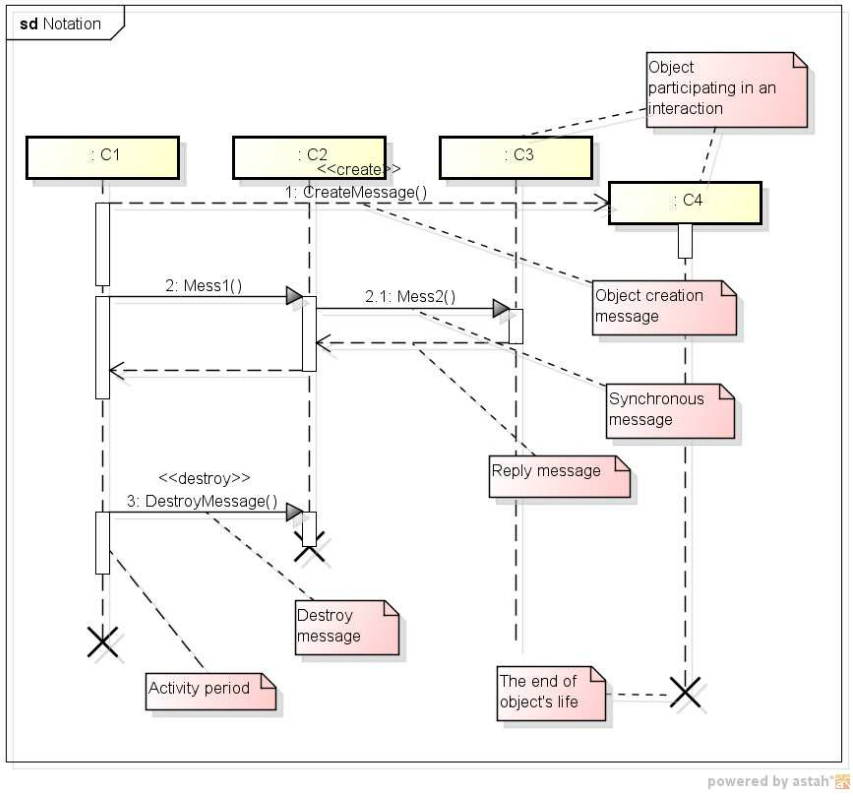


Fig. 5.1. A sequence diagram notation

The main application for sequence diagrams is concerned with defining scenarios of use cases. The set of sequence diagrams is a specification of the future information of system behavior. It should be noted that such a specification is usually not complete, it is focused on the main flow of messages between the system (subject) and its users. However, it should also be noted that, in practice, an attempt to show all possible sequences may be a trap.

5.2. Referenced fragments

Sometimes a given interaction may appear as a fragment of more complex interactions. In such case it is useful to define the fragment and next to refer to this fragment in the more complex interaction. Fig. 5.2 presents a sequence diagram which is intended to be inserted into more complex sequence diagrams. The diagrams may be glued together via input and output gates.

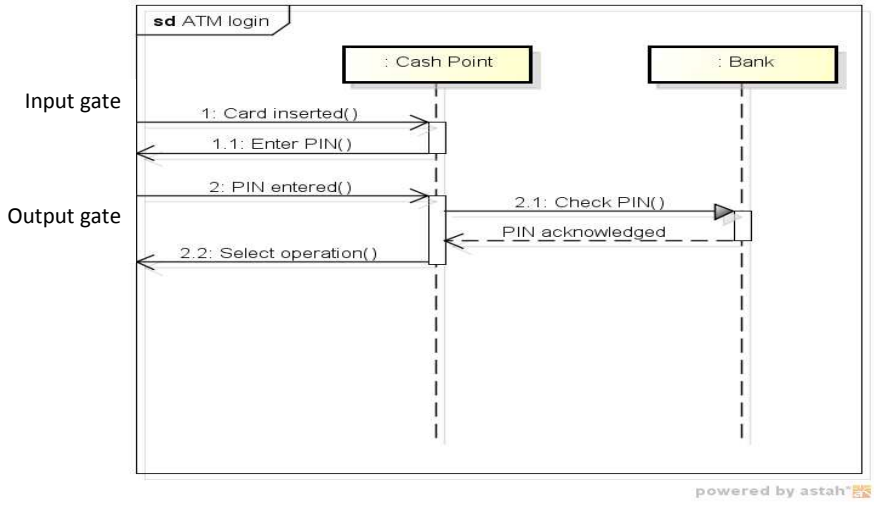


Fig. 5.2. An example of a single scenario

The sequence diagram on Fig. 5.3 presents an interaction which refers to some sequence diagrams defined independently alongside.

To present a set of scenarios on a sequence diagram, the construction of combined fragments is used. A sequence diagram may be defined as a composition of combined fragments. A combined fragment represents a fragment of an interaction. Interactions include a number of constructs for representing contingent behavior, such as conditionals, loops, and so on. Because many of these constructs share structural information, they are grouped together as kinds of

combined fragments. There are several kinds of combined fragments, each comprises an operator keyword and one or more interaction operands representing subfragments of an interaction.

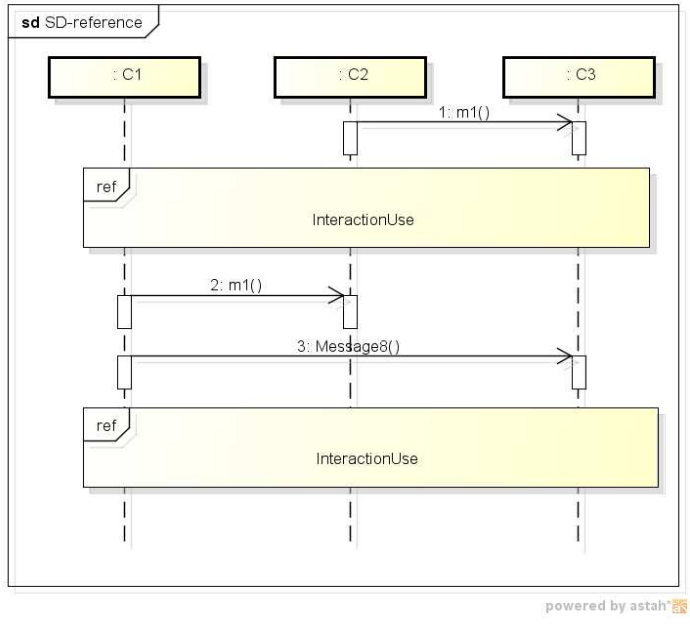


Fig. 5.3. A sequence diagram with references

5.3. Selected combined fragments

Further, only three combined fragments – alt, loop, consider and par – are presented in Fig. 5.4, Fig 5.5, Fig. 5.6 and Fig 5.7, respectively. The fragments have the following interpretation:

- A conditional fragment alt has multiple operands. Each operand has a guard condition. The absence of a guard implies a true condition. The condition else is true if no other guard evaluates true. Exactly one operand whose guard evaluates true is executed, unless no guard is true. If more than one operand evaluates true, the choice may be nondeterministic.
- A loop construct has one subfragment with a guard. The guard may have a minimum and maximum count as well as a Boolean condition. The subfragment is executed as long as the guard condition evaluates true, but it is executed to at least the minimum count and no more than the maximum count. If the guard is absent, it is treated as true and the loop depends on the repetition count.

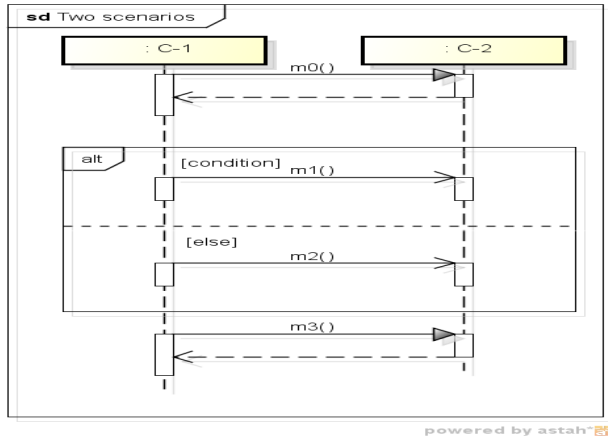


Fig. 5.4. A sequence diagram with the conditional combined fragment

- A consider construct has one subfragment and a list of message types. Only the listed message types are represented within the subfragment. This means that other types can occur in the actual system but the interaction is an abstraction that ignores them.
- A parallel construct has two or more subfragments that are executed concurrently. The execution order of individual elements in parallel subfragments may interleave in any possible order (unless prohibited by a critical construct). The concurrency is logical and need not be physical; the concurrent executions may be interleaved on a single execution path.

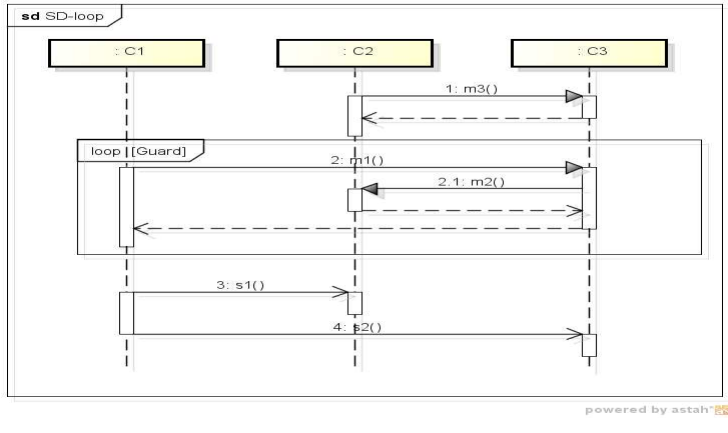


Fig. 5.5. A sequence diagram with the loop combined fragment

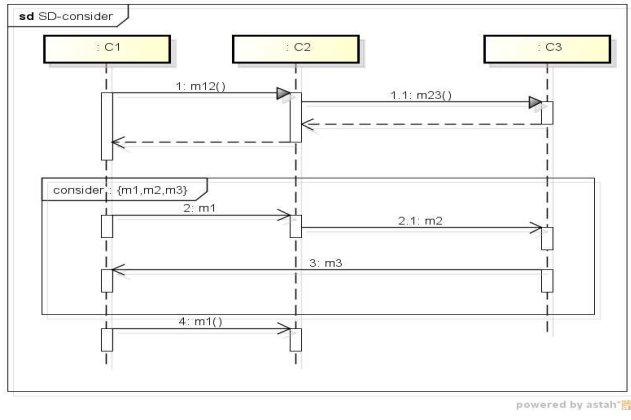


Fig. 5.6. A sequence diagram with the consider combined fragment

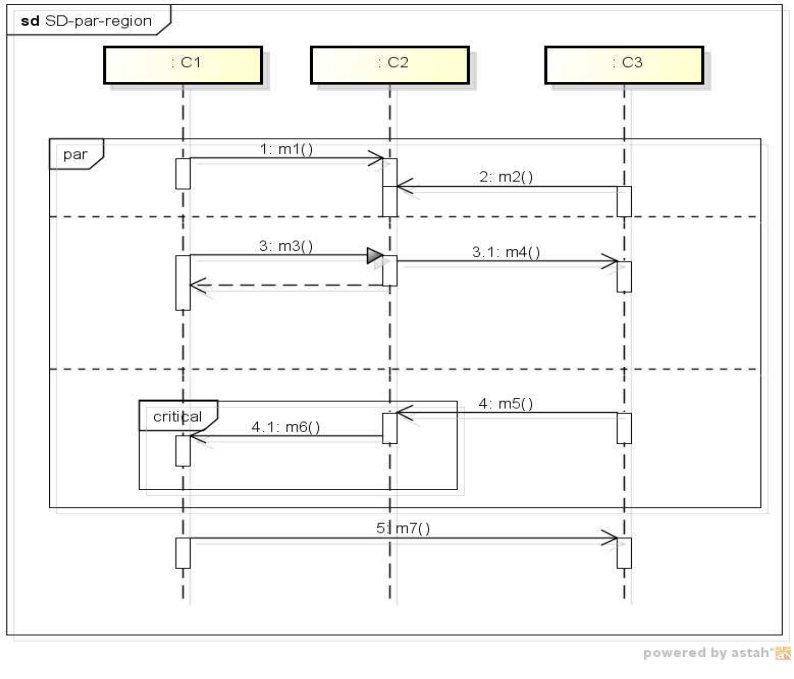


Fig. 5.7. A sequence diagram with the parallel combined fragment

6. Interaction diagrams

Sequence diagrams presented already, together with communication, interaction overview, and timing diagrams form the group of interaction diagrams. Closely related to them are activity diagrams.

Communication diagrams and timing diagrams may be considered as equivalent to sequence diagrams. Therefore, they have the same area of application, i.e. pictorial description of scenarios of use cases, however, timing diagrams are especially suitable for real-time systems. Interaction overview diagrams may be useful in detailed design phase for pictorial description how to cooperate components of a developed system.

6.1. Communication diagrams

A communication diagram shows information similar to that on a sequence diagram but in different manner. The communication diagram consist of two elements:

- an object diagram that shows all participants of an interaction, and
- the messages that are attached to the links of the object diagram.

A communication diagram does not show time as a separate dimension, so the sequence of messages and the concurrent threads must be determined. Messages on the diagram have labels that determine the ordering in time. A simple example of a communication diagram being a counterpart of the sequence diagram shown on Fig. 5.2 is presented in Fig. 6.1.

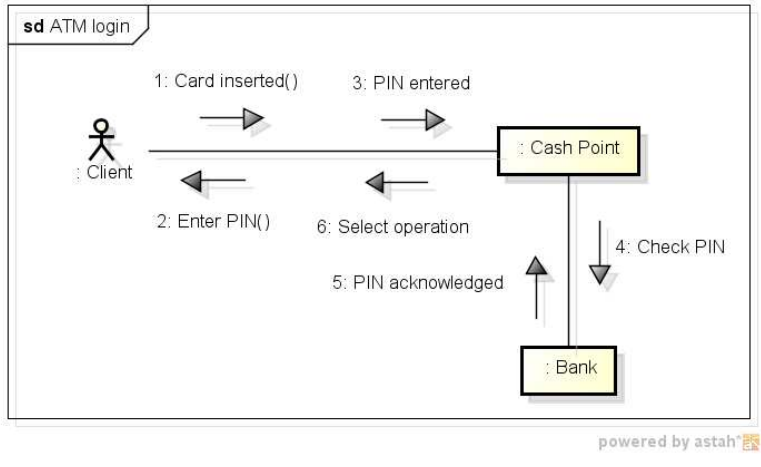


Fig. 6.1. An example of a communication diagram

6.2. Interaction overview diagram

An interaction overview diagrams provide a high-level overview of the interaction within the system. The interactions are best depicted using sequence diagrams, interaction overview diagrams contain references to sequence diagrams with decision and fork notation taken from activity diagrams. A simple example of the diagram is shown on Fig. 6.2.

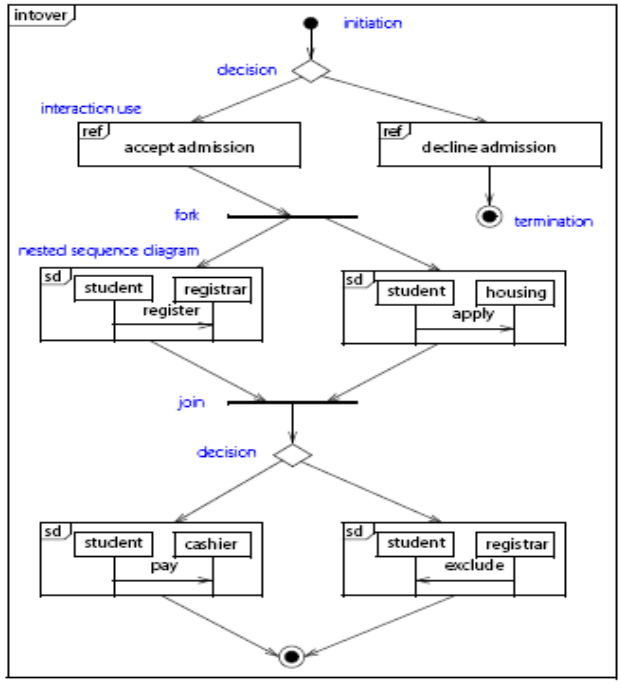


Fig. 6.2. An example of interaction overview diagram (Rumbaugh J., Jacobson I., Booch G., *The Unified Modeling Language – Reference Manual*. Second edition, Addison-Wesley, 2005)

6.3. Timing diagram

Timing diagrams were included only to the UML version 2. They were very popular and useful in real-time applications. A timing diagram is a special-purpose interaction diagram that focuses on the timing of events over the life of an object. Timing diagrams use two forms: a state timeline, and a general value timeline. Timing diagrams may be considered as a special form of sequence diagrams with the following notation convention:

- The axes are usually reversed so that time increases from left to right.
- Lifelines are shown in separate compartments arranged vertically.

- The lifeline jogs up and down to show changes in state. Each vertical position represents a state.
- Alternately, a lifeline can follow a single line with different states or values displayed on the line.
- A metric time axis may be shown. Tick marks indicate time intervals, sometimes discrete times at which changes happen.
- The times on different lifelines are synchronized.
- The value held by an object may be shown.

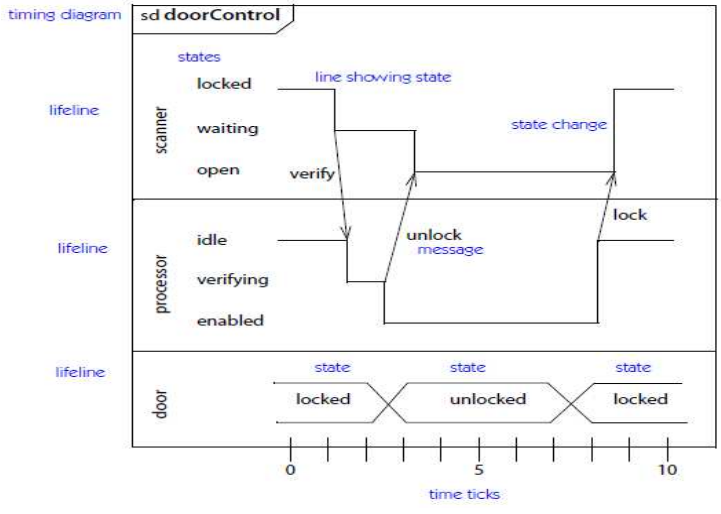


Fig. 6.3. An example of a timing diagram (Rumbaugh J., Jacobson I., Booch G., *The Unified Modeling Language – Reference Manual*. Second edition, Addison-Wesley, 2005)

7. Activity diagrams

Activity diagrams may be considered as generalization of flow charts. They are used to show the flow of actions within processes or systems. Activity diagrams enable to show dependency between activities concentrating basically on the flow of control, and also on the flow of data. This makes them capable of being used to model processes at business process level, within use cases, and occasionally between use cases.

Nodes of an activity diagram include action, object, and control nodes. The nodes may be nested. Arcs of an activity diagram include control flow and data flow arcs.

An activity is a specification of executable behavior as the coordinated sequential and concurrent execution of subordinate units, including nested activities and ultimately individual actions connected by flows from outputs of one node to inputs of another.

An action – a primitive (elementary – indivisible, uninterrupted) activity whose execution results in a change in the state of the system or the return of a value. They include arithmetic and string functions, manipulations of objects and their values, communications among objects, and similar things. Action may have input and output pins with data flow arcs, and input and output arcs with control flow – see Fig. 7.1.

An action may begin execution when input values are available on all its input pins and control tokens are available on all incoming control arcs. When execution begins, the input values and tokens are consumed so that no other action can use them. When the execution of an action is completed, output values are produced on all its output pins and control tokens are placed on all outgoing control arcs.

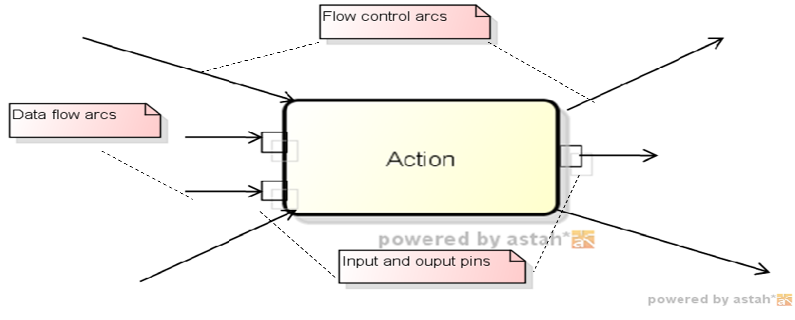


Fig. 7.1. An action notation

Simple example of an action diagram is presented on Fig. 7.2. The diagram consists of actions and control nodes that are related by control flow arcs. Only fork and join control nodes

need an explanation. The fork node copies a single input onto multiple concurrent outputs. The join node synchronizes multiple flows onto a single output.

The next activity diagram from Fig. 7.3 presents special activity nodes representing signals sending and receiving.

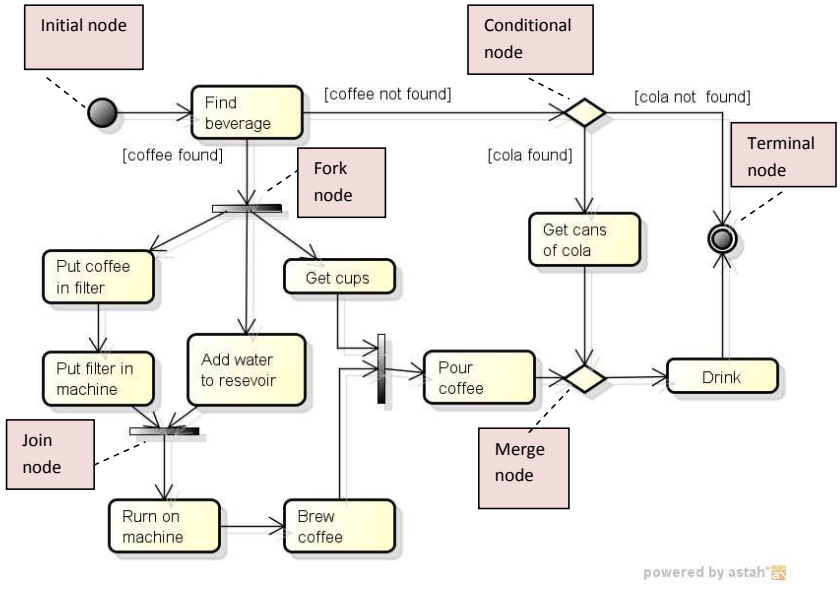


Fig. 7.2. An example of an activity diagram

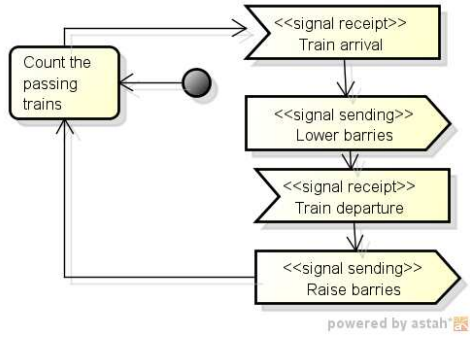


Fig. 7.3. An activity diagram with <<signal sending>> and <<signal receiving>> actions

The last example of an activity diagram is given in Fig. 7.4. The diagram shows control and data flows. The data flow is represented by the object with stereotype <<Document>> and the object Product. The objects pass between activities which change their states; the states are exhibited within brackets. This diagram shows also so called swimlanes that define a partition of the set of activities into subsets with a responsibility assigned to some actors.

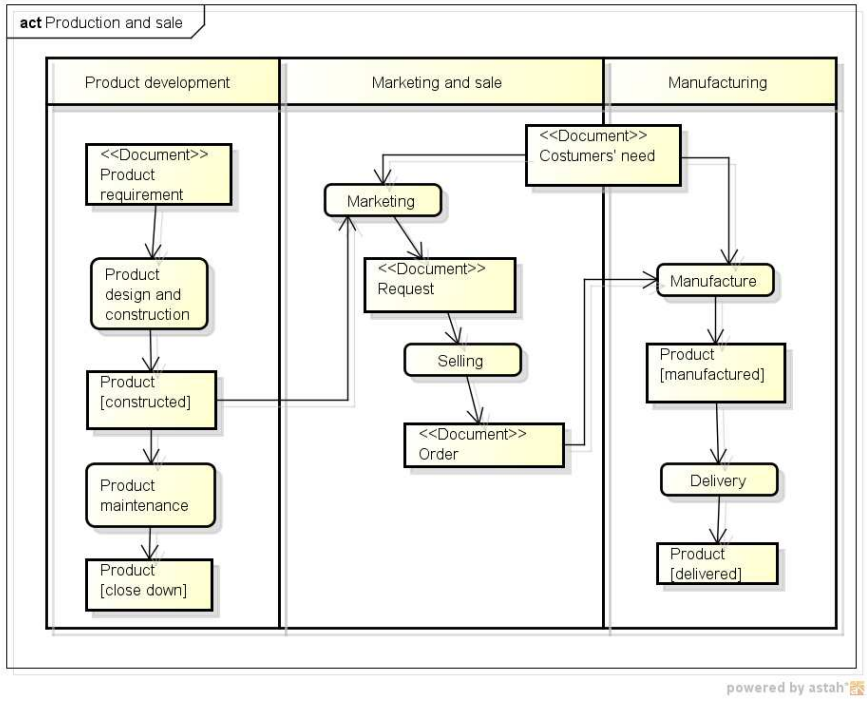


Fig. 7.3. An activity diagram with responsibility lanes

8. State machine diagrams

State machine diagrams (statecharts) are pictorial representation of finite state machines that may be considered as an extension of finite automata with memory. Usually, they are attached to classes but they may be also attached to methods or collaborations. When attached to a class, a state machine diagram is a template of a behavior of each object of the class. A state machine diagram is a specification of a sequence of states that an object goes through in response to events during its life with its responsive effects. The states represent a set of valuations of object's attributes. The events are calls of object's operations or receptions of signals. The responsive effects are actions and activities performed to handle the events.

As a graph a state machine diagram consists of nodes that represent states, and arcs that represent transitions between states. The nodes (just called states) are simple and composite (nested) states, and pseudostates. Arcs are transitions between nodes (states).

In contrast to interaction diagrams which are used as a partial behavioral specification, the state machine diagrams are very convenient for the complete specification. They are also clear and understandable. Their disadvantage is the explosion of states with increasing of system's size.

8.1. Diagrams with simple states

An example of simple state diagram is on Fig. 8.1. Two states Off and On are the main part of the state machine. The pseudostate on the left points Off as an initial state. The state on the right represents the final state which means that an object terminates the execution of its state machine. Transition between states are triggered by the occurrence of events that labels the arcs. So, the event swichOn moves from Off to On state, the event swichOff moves from On to Off state, and the event Finish moves from On to the final state.

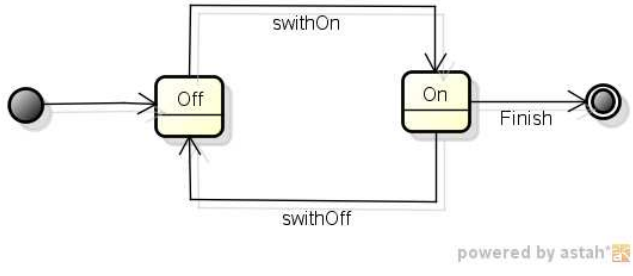


Fig. 8.1. An example of simple state machine diagram

However, simple states are not so “simple”. It is because, there are many actions that may be performed within a given simple state. The internal structure of a state is given on Fig. 8.2. All parts of the structure are optional.

An entry activity is executed immediately when the state is entered. After that an internal do activity starts its execution. An exit action is executed when the state is exited. It may happen after the completion of an internal do activity. Internal transitions are similar to transitions between states except they do not cause the change of state. A deferred event is the event that is recorded in the state but its recognition is postponed to a subsequent state which does not declare the event as deferrable.

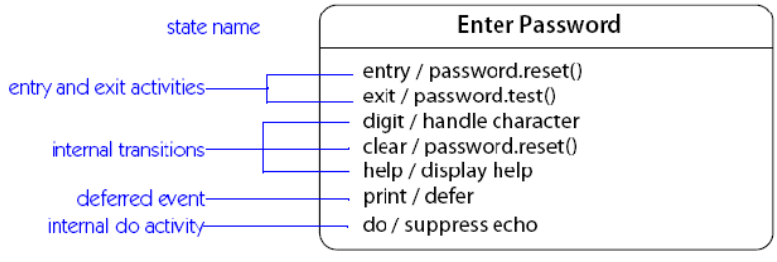


Fig. 8.2. An internal structure of a simple state (Rumbaugh J., Jacobson I., Booch G., *The Unified Modeling Language – Reference Manual*. Second edition, Addison-Wesley, 2005)

Fig. 8.3. presents the state machine diagram which is a modification of the diagram from Fig. 8.1. The states On and Off are equipped now with entry and exit actions and with internal do activity. Additionally, the transition labels are also extended. The label transition may consist of three element: a triggering event (e.g. Click()), condition (e.g. allClosed), and transition action (e.g. bye, welcome). So, the transition from On to Off state is executed in the following way:

1. If the state machine is the state On (in other words, the state On is active) and the internal do activity a-d is completed then an occurrence of the event Click(), provided the condition allClosed is satisfied, causes exiting the state On.
2. The last activity performed before leaving the state On is a-ex action.
3. After a-ex action completion the transition action bye is executed.
4. After bye action completion the state Off is entered and its entry action b-e is executed.
5. After b-e action completion the internal do activity b-d starts its execution.

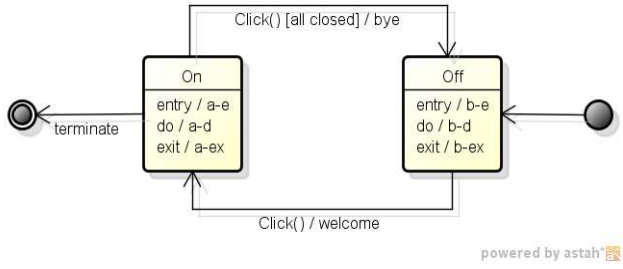


Fig. 8.3. State diagrams with simple states

8.2. Diagrams with sequentially composite states

Composite states may be sequentially or parallelly nested. A sequentially composite or nonorto-gonal state consists of one region which contains a new state machine diagram. The states of the new state machine are considered to be substates of this composite state. In turn, each of the substates may also be a composite, i.e. it may have its substates. Entering a composite state means also simultaneous entering its initial substate.

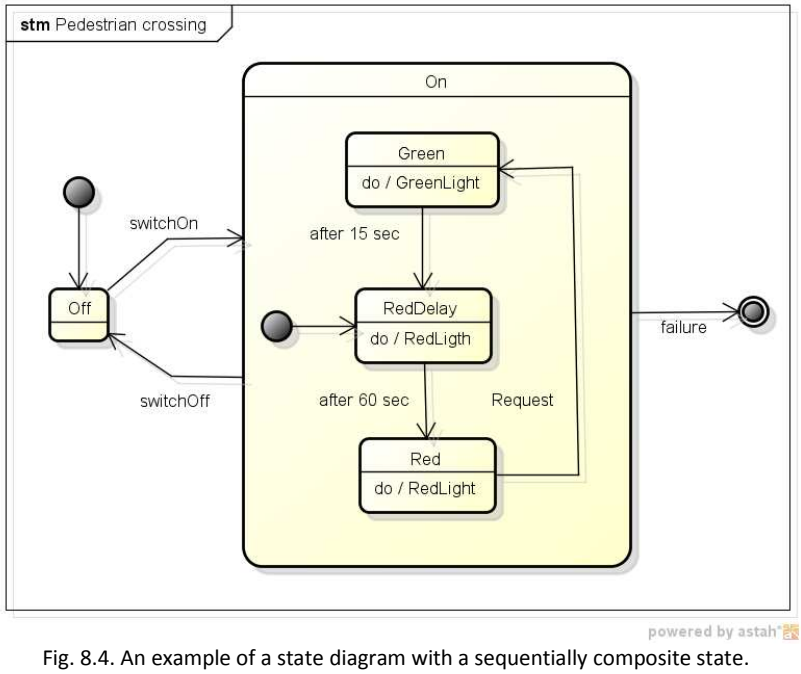


Fig. 8.4. An example of a state diagram with a sequentially composite state.

In the example on Fig. 8.4 transition from Off state to On state means to enter at the same time On and RedDelay states. These active states determine an active configuration of the Pedestrian crossing state machine. From this configuration you can go to the (configuration) states:

- On and Red states, in the result of after 60 sec event,
- Off, in the result of switchOff event, and
- to the final state, in the result of failure event.

The event after 60 sec has specific meaning, namely it is a timed event which appears automatically after 60 seconds from the time the configuration On and Red is entered, provided there were no sooner events triggering from this configuration. It should be noted that each of the events switchOff and failure trigger from the state On regardless of which of its substate is active.

The Pedestrian crossing state machine as well as further presented state machines are structurally constructed. It means that the arcs do not cross states' border. Not crossing states' border is strongly recommend state machine diagram construction principle. It is a counterpart of the principle not using goto statement in program construction.

8.3. Diagrams with parallelly composite states

A parallelly composite or ortogonal state consists of many regions; each of the regions may contain a new state machine diagram. The state machines in the regions are considered to work in parallel. Therefore entering a parallelly composite state means also simultaneous entering all its regions, and consequently, an initial substate in each region.

The state machine HairDryer, presented on Fig. 8.5, has one parallelly composite state On which consists of two regions. Entering the state from the state Off means simultaneous entering its regions and to low and cold as regions' initial substates. Now, the configurations of active states are determined by trees. Below, there are presented the initial configuration after entering the state On, and also other possible configurations with the state On as a root. Transitions between the configurations occur as a result of events PressFlow, down, and up within the state On. The occurrence of the event switchOff will exit from the state On regardless of its active configuration. Re-entrance to the state On activates its initial configuration.

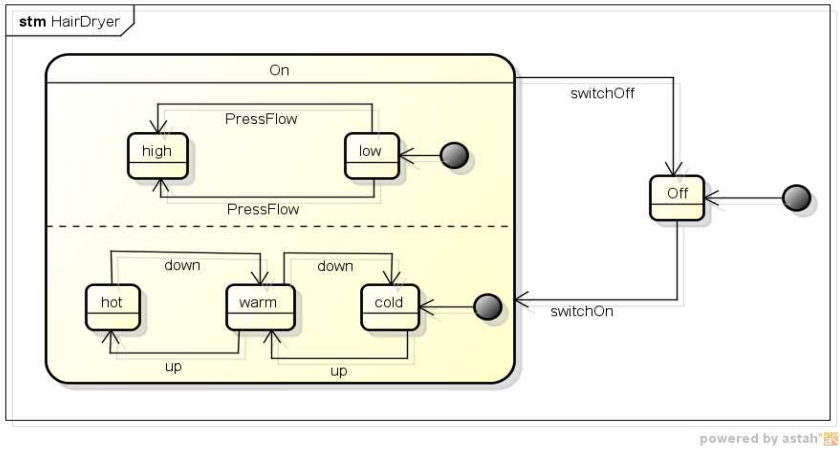


Fig. 8.5. State diagrams with parallelly composite states

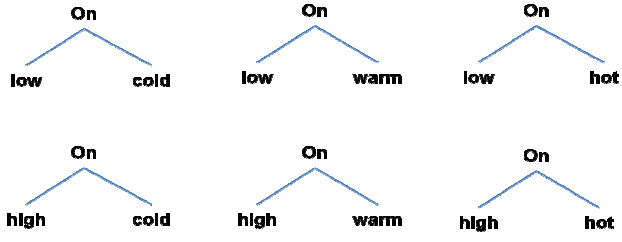


Fig. 8.6. Possible active configurations for the state On

Next two examples of state machine diagrams illustrate the meaning of shallow and deep history states. History states allow a composite state to remember the last substate that was active in it prior to the most recent exit from the composite state. A transition to the history state causes the former active substate to be made active again.

Fig. 8.7 and Fig. 8.8 are a slight modification of the diagram from Fig. 8.5. The only difference is replacing initial states in On regions by shallow and deep history states, respectively. A shallow history state remembers and reactivates a state at the same nesting depth as the history state itself. If a transition from a nested substate directly exited the composite state, the enclosing substate at the top level within the composite state is activated, but not any nested substates. A deep history state remembers a state that may have been nested at some depth within the composite state. To remember a deep state, a transition must take the deep state directly out of the composite state. If a transition from a deep state goes to a shallower state, which then

transitions out of the composite state, then the shallower state is the one that is remembered, because it is the source of the most recent exit. A transition to a deep history state restores the previously active state at any depth. In the process, entry activities are executed if they are present on inner states containing the remembered state.

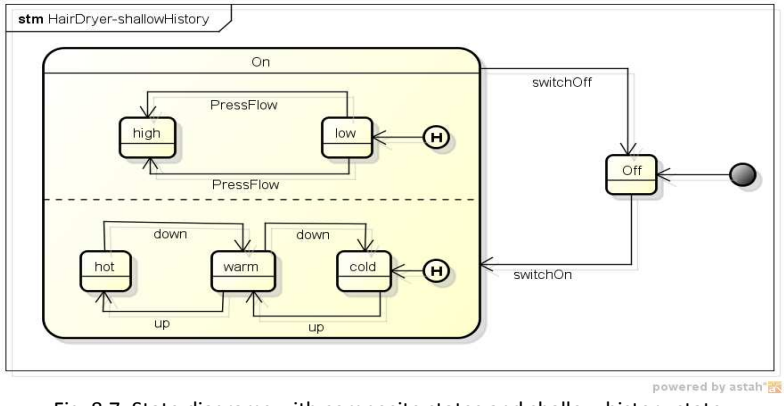


Fig. 8.7. State diagrams with composite states and shallow history state

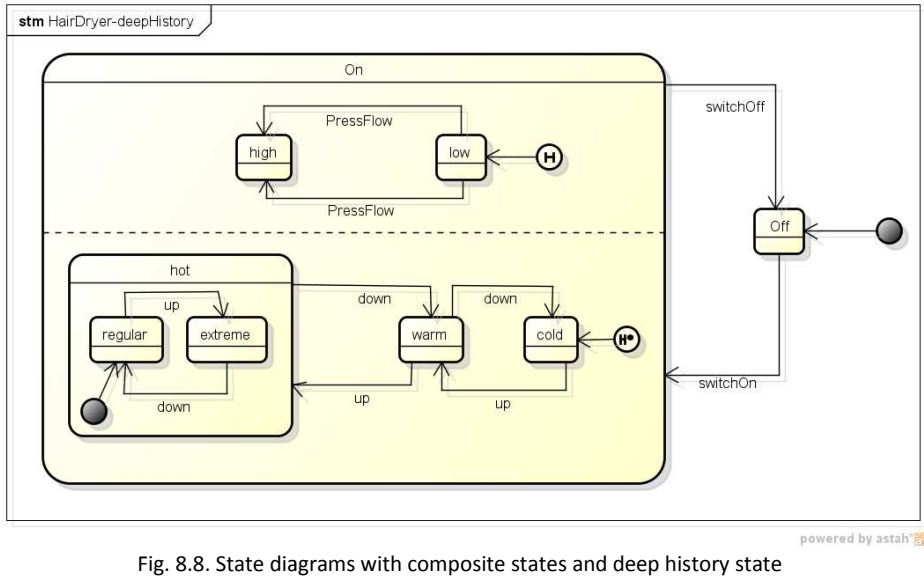


Fig. 8.8. State diagrams with composite states and deep history state

9. Implementation diagrams

Component and deployment diagrams are called sometimes implementation diagrams. Implementation is a realization of a technical specification or algorithm as a program running under an operating system on a specific hardware. The technical specification is a model elaborated within design phase. A programming language and an operating system may be a set of other auxiliary programs, data bases, together with a hardware (computers, peripheral equipment) form so called implementation platform. There are many different platforms offering different facilities, therefore the choice of the implementation platform is an important decision taken at the beginning of design phase, during its subphase, called a general design. The design decisions are related to a project of architecture of the designed information system which bases on selected implementation platform. An architecture is the organizational structure of a system in terms of parts and relationships between them. Components representing software part, and nodes representing hardware part of the designed system, are the basic elements of the structure.

9.1. Component diagrams

The main kind of component diagram nodes are components. Moreover, classes and artifacts may also act as nodes.

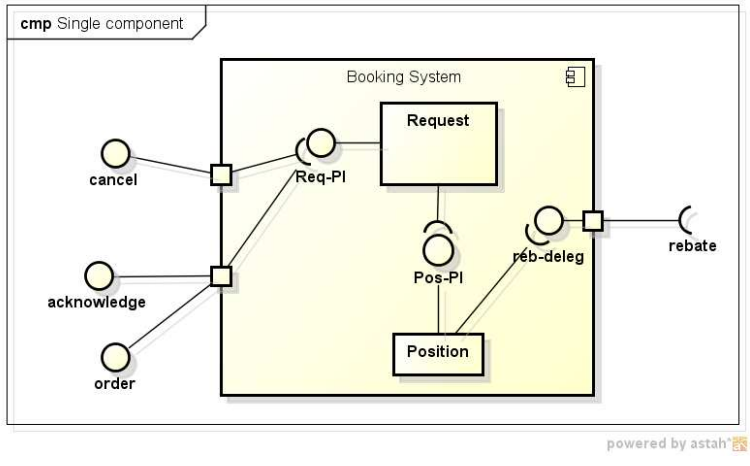


Fig. 9.1. An example of a component

In the UML version 2 a component is a kind of class (in UML 1.x – a classifier) that cannot have attributes, operations and methods on its own. It may be considered as a container of other classes that define its features. The visible component's features are interfaces that specify services

defined by the contained classes. A component hides its implementation behind a set of external interfaces that may be attached to ports. A component can be replaced by another component that supports the proper interfaces, and a component instance within a system configuration can be replaced by an instance of any component that supports the same interfaces. Pictorial representation of a component and its content – two classes – is given on Fig. 9.1. The classes Request and Position belong to the content of the component Booking System. There is also an alternative notation for the elements belonging to a component – see Fig. 9.2. The class Local database belongs to the component Booking System which is expressed by stereotyped dependency relationship <<reside>>.

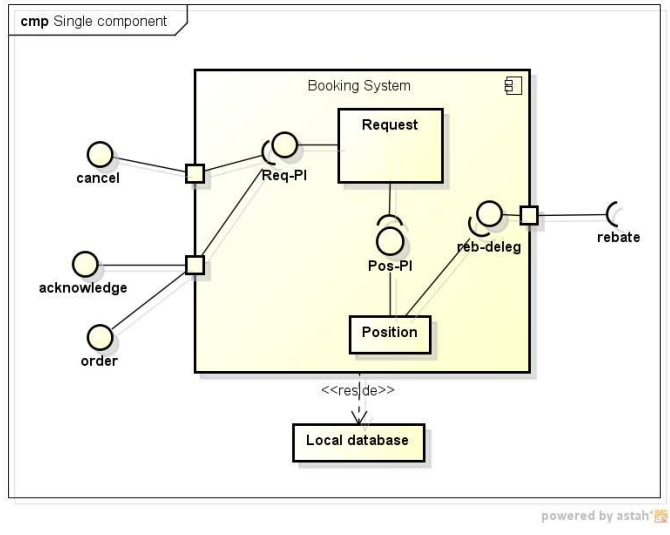


Fig. 9.2. Illustration of <<reside>>dependency use

A component can be a source code component, a binary component, or an executable component. Therefore, there is a number of components' stereotypes, for example:

- «executable» – a component that runs on a processor,
- «library» – a set of resources referenced by an executable during runtime,
- «table» – a database component accessed by an executable,
- «file» – typically represents data or source code,
- «document» – a document such as a Web page.

In addition to this stereotypes two other are in use. These are «specification» and «implementation» stereotypes. A component with «specification» stereotype describes its domain without providing physical implementation, typically it describes interfaces or constraints without listing

their features. A component with «specification» stereotype provides the implementation for a separate specification on which it depends.

Dependencies between components are limited to the dependencies between their interfaces – see Fig. 9.3.

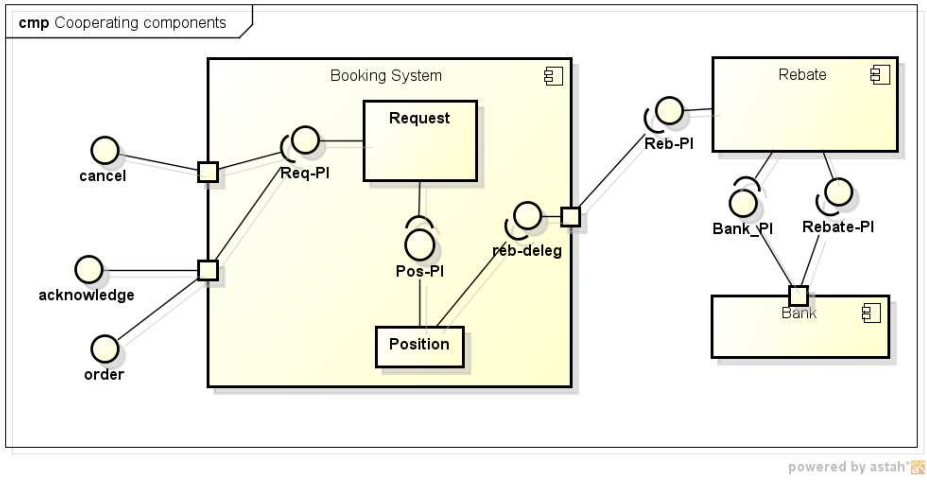


Fig. 9.3. Dependencies between components

Components may be related not only with classes but also with artifacts. In general, an artifact is defined as the specification of a physical piece of information that is used or produced by a software development process. In the context of the transition from design to implementation, every logical construct of the design must be mapped to an artifact as an implementation element. Artifacts are classifiers. They can be a text document, a model, different kinds of files etc. Artifacts may be related with a component via manifestation dependency – see Fig. 9.4. The dependency <<manifest>> means that the artifact, in its final implementation, is represented by a set of artifacts. In general, a manifestation is the relationship between a model element and the artifact that implements it. This is a many-to many relationship. One model element may be implemented by multiple artifacts, and one artifact may implement multiple elements.

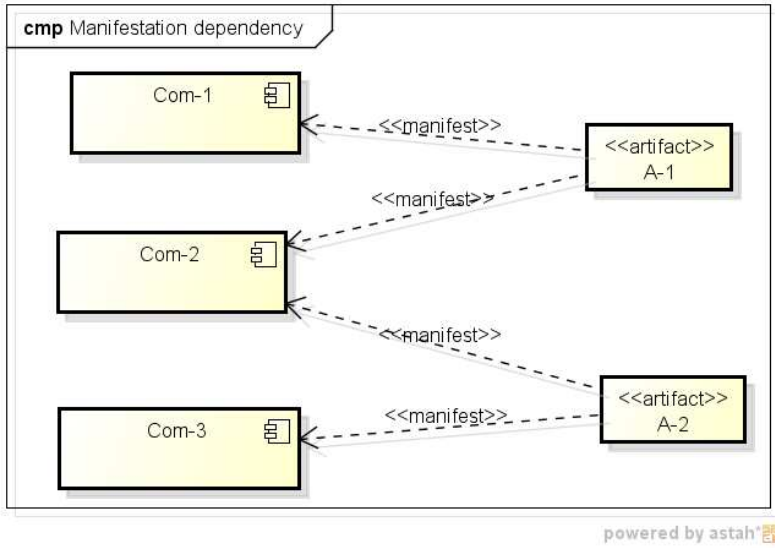


Fig. 9.4. An example of a dependency between components and an artifact.

Primary aim of component diagrams is to map components to classes, and next transform the classes into executable. Component diagrams do this by providing interfaces for a good, reusable design. Summarizing, a component diagram is a pictorial representation of the final executables or linked libraries that contain the code of the system. Note, that component diagrams can occur at two levels: a type level and instance level.

9.2. Deployment diagrams

The deployment diagram is the only diagram in UML that shows the physical architecture of the hardware and software in the system. The main elements of deployment diagram are nodes. Nodes are classifiers that represent computational resources, which have at least memory and often processing capability. The computational resources may not only be electronic computers but also mechanical devices or human resources. Inside the nodes, executable components and objects are allocated to show which software units are executed on which nodes. Nodes are connected by associations that represent communication paths.

Between components deployed on nodes above discussed dependencies between the components may also be shown.

Similarly as component diagrams, deployment diagrams can occur on type and instance level. An example of such diagrams is presented on Fig. 9.5. The example from Fig. 9.6 shows a

deployment diagram at instance level which is an implementation model of a control system positioning a telescope. The nodes like X-actuator or X-sensor represent specialized mechanical devices that measure and adjust the telescope position with respect to X dimension.

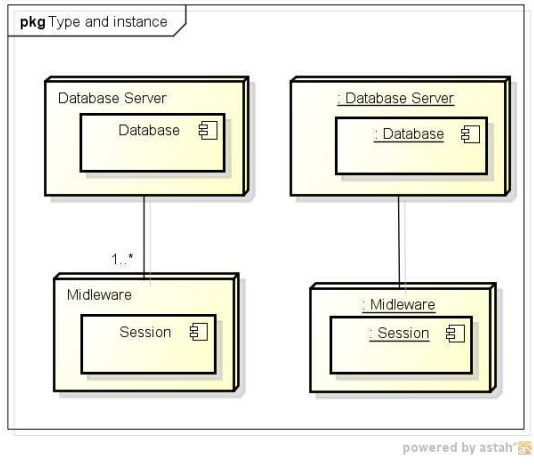


Fig. 9. 5. An example of type level and instance level deployment diagrams

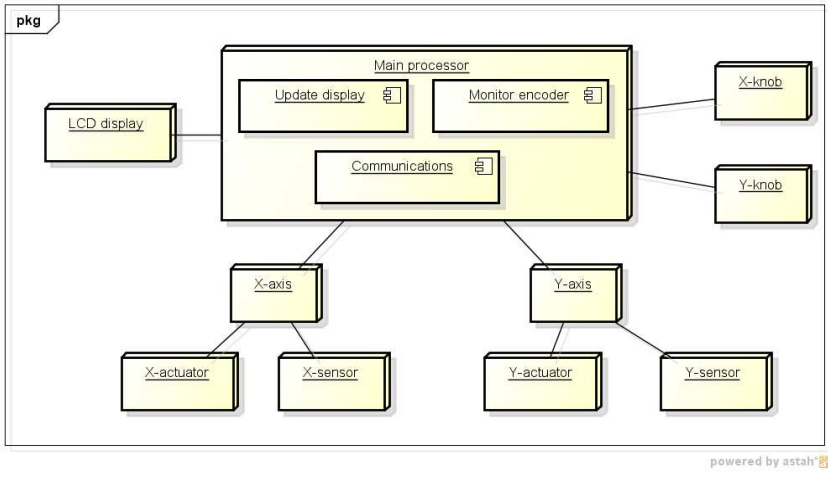


Fig. 9.6. An example of a deployment diagram at the instance level.

10. Package diagrams

To explain package diagrams, we have to describe first what a package is. The package is the main construct in UML for grouping model elements. It enables to group the elements and place them into a container. A package provides exactly the same functionality as a folder, i.e. it provides a namespace for the elements placed in the package. It means that an element of a given type placed in a package becomes a unique name for an element of that type in that package. An element of the same name in a different package has a different definition.

Packages can contain other packages, so it is possible to create a hierarchy of packages.

Fig. 10.1 presents a hierarchy of packages. A package is shown as a large rectangle with a small rectangle attached to the left top corner of the large rectangle. Hierarchy (nesting) of packages is presented in one of two notations. The first notation convention relates to the package P-1 which contains two packages P-2 and P4. The second convention relates to the package P-2 which contains also two packages P-3 and P5.

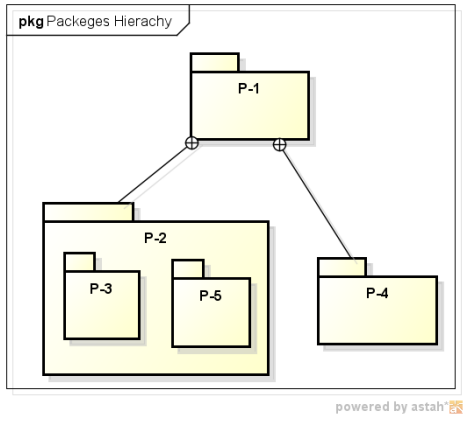


Fig. 10.1. Notation conventions

The visibility of a package element outside the package may be indicated by preceding the name of the element by a visibility symbol ('+' for public, '–' for private). Except nesting there are several dependency relationships between packages. The dependencies are represented by dashed arrows between package symbols. Fig. 10.2 presents a dependency «access». The dependency establishes the fact that all classifiers in the source package can legally reference public classifiers in the target package. In the example it enables the package PP to use elements from the package PR. The class B from the package PR is referenced with qualified name PR::B in the package PP. Therefore, the association between A and PR::B is justified.

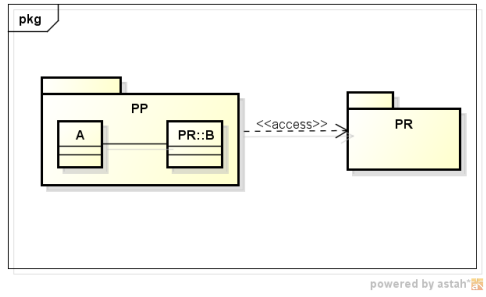


Fig. 10.2. <<access>> dependency between packages

In UML 1.x an <<import>> relationship between packages was defined, meaning that a copy of a classifier from one package is brought into another package. The new copy of the imported classifier belongs to the package that you imported it into, while its original version still exists in its original package.

In UML 2.0, the <<import>> dependency relationship between packages is subsumed in the <<merge>> relationship. The relationship should take into account the need to resolve name conflicts when importing some classifiers. Simple example, without name conflict, illustrating a result of package merging is given on Fig. 10.3. The upper diagram presents diagrams before merging, and the bottom diagram – after merging.

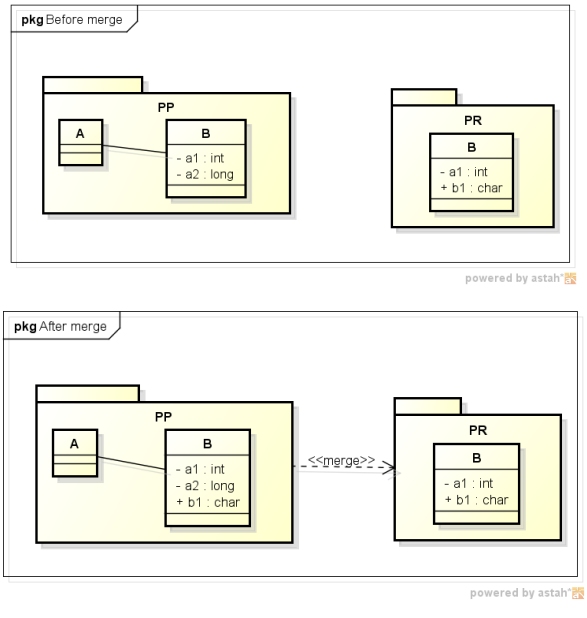


Fig. 10. 3. <<merge>> dependency between packages

A nested package has access to any elements directly contained in outer packages (to any degree of nesting), without the need to import them. However, a package must import its contained packages to add them to its direct namespace. A contained package is, in general, an encapsulation boundary.

A package defines the visibility of its contained elements as private or public. Public elements are available to other elements of the owning package or one of its nested packages and to packages importing the package. Private elements are not available at all outside the owning package. Contents of other element types (such as attributes and operations) may also have the visibility protected or package. Protected elements are available only to descendants of the classifier owning the feature. Package elements are available to all elements within the same package as the classifier owning the feature.

Package diagrams are very important means to organize work during software development process. To support management of the process, package diagrams are customized to illustrate different aspects of systems. For example, a package diagram can model the structure of systems and subsystems. Also models are presented as a kind of package.

11. UML profiles

The UML, despite being very large and having many redundant mechanisms turns out to be not well suited for certain applications. One solution to this difficulty is the use of UML profiles. A profile is defined by a set of stereotypes that are added to the set of model elements. The stereotypes are defined using the extension mechanisms built into the language. Apart from stereotypes, these mechanisms comprise tagged values, and constraints.

A stereotype is a type that may be applied as a property value to a UML element to extend its properties and slightly alter its semantics. Stereotypes may be applied to all UML element types, such as classes, use cases, components, associations, dependency relationships, and so forth. Applying a stereotype to a model element, additional properties, or tagged values may be added. The stereotype of a UML element is provided in guillemets above the element's name. Some stereotypes are associated with icons.

A tagged value is an extra property that can be added to a UML element, in this way enabling specification of additional information. A tagged value is defined as a pair: <name, value>. An example of a tagged value might be the business analyst that created a use case, or the author of a class, or the date and time when a class was created or modified. Having some tagged values in mind, we may define the stereotype <<authorships>> in the following way: each instance of a classifier with this stereotype should contain three tagged values:

- author's name = a, where a is a value of the type string;
- creation date = cd, where cd is a value of the type Date;
- last modification date = lmd, where lmd is a value of the type Date.

Constraints enable specification of rules and restrictions on model elements in UML. For example, a profile might specify a constraint that you cannot draw an association between two classes of a given stereotype.

There are many UML profiles that have been adopted by the OMG or are undergoing the adoption process. For example, to the set of profiles belong:

- UML Profile for Business Modeling,
- UML Profile for Software Development Processes,
- Profile for Java,
- Component Profile Examples, for J2EE/EJB, COM, .NET, and CCM.

The UML Profile for Software Development Processes is provided as an example profile in the UML 1.4 specification itself. The profile is defined only through stereotypes and constraints; no tagged values are specified. Below, selected three stereotypes, all based on classes, are presented:

- A Boundary Class sits at the boundary to a system, and is an interface between actors outside the system and the classes within the system, such as entity, control, and other boundary classes. In Jacobson's early work, it was called an interface class, but the name was changed so that it wasn't confused with a component interface.
- The objects of a Control Class control the interaction between other class objects. The behavior of a Control Class is usually specific to the Use Case for which it is modeled.
- An Entity Class is a passive class - objects of other classes work with it or on it. The Entity Class does not initiate interactions on its own. It is usually persistent, in that it stores information and exists after any use case realization that it is in finishes.

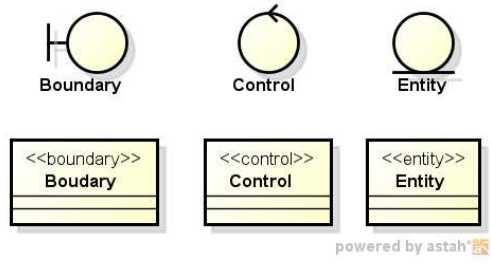


Fig. 11.1. Stereotypes notation

Fig.11.1 presents icons being representation of the stereotypes. Upper part contains specific icons for the profile, while lower part of the figure presents universal notation – the pictorial representation of the stereotyped model element, here a class, and the name of the stereotype in guillemets placed above the element's name.

12. Model Driven Architecture

To conclude the presentation of the UML it is worth to refer to its place in software development process. The MDA (Model Development Architecture) approach, mentioned in the first chapter, will be a background for the further consideration. The MDA approach is quite general, and still in progress, but elements of this approach can be found in almost all modern software development methodologies.

MDA stems from the belief that models expressed in a well-defined notation are a cornerstone to understanding both problem domain and solution domain. Moreover, the process of system development can be organized around a set of models by imposing a series of transformations between models – see Fig. 1.2.

The belief on usefulness of modeling is based on experiments that show that models help people understand and communicate complex ideas. Models have wide application areas and may, for example, to:

- capture and precisely state requirements and domain knowledge so that all stakeholders may understand and agree on them;
- think about the design of a system;
- capture design decisions in a mutable form separate from the requirements;
- generate usable artifacts;
- explore multiple solutions economically.

The contents of particular CIM, PIM, and PSM models is not defined within the MDA specification. So, Fig. 12.1 below presents only an exemplary package diagram that comprises all these models. Each model is represented as a separate package containing several sub-packages. The sub-packages forming the entire model express both structural and behavioral aspects of the modeled domain.

The modeled domain for the CIM model is a fragment of reality, e.g. institution or organization, which is the place where future information system is to be exploited.

The modeled domain for the PIM model is the CIM model, and more precisely, only this fragment of the CIM model which the future information system will support.

For the PSM model, the modeled domain is the PIM model. Informally, the PIM model may be considered as a transformation of PSM model. The PSM model should define the same data processing processes as PIM does, but PSM defines the processes in terms of the mechanisms and means taken from a chosen computation platform.

Finally, the code model is just the PSM model rewritten into a program.

So, the CIM, PIM and PSM models represent different semantic levels, while code is on the same semantic level as PSM.

Now, we will give short comments on each of the exemplary models.

CIM model consists of four packages. Business Vocabulary package contains a text consisting of a list of pairs: notion – its meaning explained in plain language. The list should contain all the notions used in business analysis. If some notion is not present in the list, it is assumed that its meaning is the same as in dictionary of a given language. The package Business Class Diagram is a model of structural aspect of the modeled domain.

The package Business Use Case Diagram with the package Business Activity Diagram define behavioral aspect of the modeled domain.

The PIM models consists of three packages. The package Analysis Class Diagram reflects structural aspect, and the two others Analysis Use Case Diagram and Analysis Sequence Diagram, similarly to the previous model, define behavioral aspect. In contrast to CIM model, activity diagrams are replaced by sequence diagrams.

PSM model consist of four packages. Two of them, Design Class Diagram and Design State Diagram represent structural and behavioral aspects, while Design Component Diagram and Design Deployment Diagram represent the software and hardware architecture of the designed system.

The model of code is only symbolically defined. The code may consists of many programs – Implementation Elements – written in different languages.

There are dependency relationships with different stereotypes between the MDA models. All dependencies have stereotype <<trace>> which indicates a historical development process: first CIM model was elaborated, next PIM etc. Between PIM and PSM models there is the additional stereotype <<refine>>. This refinement relationship indicates that PSM is a fuller specification of an information system than PIM model is. Finally, the stereotype <<derive>> between PSM and code indicates that the code may be computed from PSM. The statement is confirmed practically by programs that automatically transform the PSM into a skeleton of the code.

A practical and theoretical problems are how to guarantee intra-model consistency, and to guarantee inter-models consistency between subsequent models.

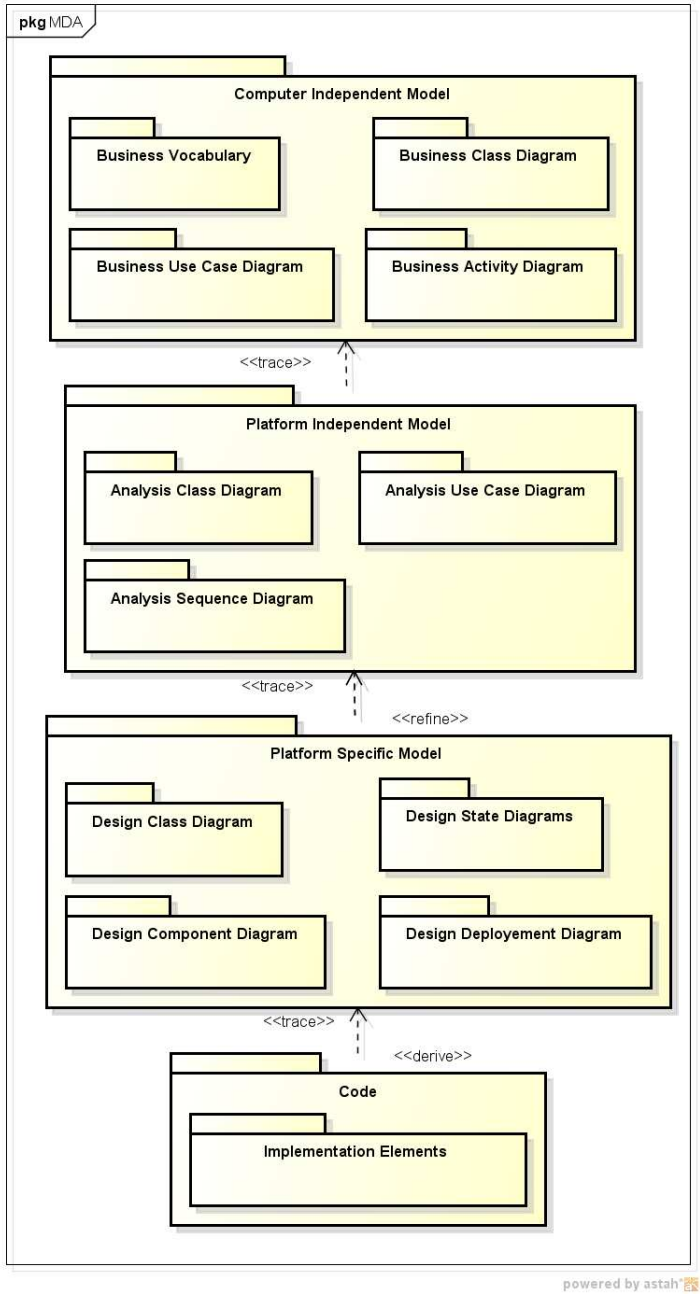


Fig. 12.1. An exemplary package diagram representing the family of MDA models

The intra-model consistency means that model is syntactically correct and there are no semantic contradictions within the model. Syntactical correctness of UML models is relatively easy to satisfy as existing programming tools supporting modeling in UML check the syntactical correctness in the fly. Semantic correctness is much more difficult to be checked. Theoretically, it may be checked by automatic theorem provers, but, in practice, the checking is done manually what resembles program testing.

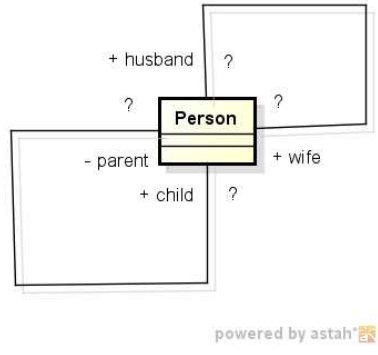
Informally, the inter-model consistency means that a later model should not lose information contained in the earlier model. Checking inter-models consistency is much more difficult as intra-model consistency checking.

Exercises

1. Let us consider a building like the one where you are now. Prepare a class diagram being a model of this kind of building. Select your perspective for the model by choosing respective level abstraction, for example, you may identify levels (stages) of the building, different kinds of rooms (private, laboratory, lecture theatre, corridors, toilets etc.). Select also some interesting associations among these notions, for example, neighborhood of rooms. Derive exemplary object diagram being an instance of the elaborated class diagram.

2. Let us consider a simple file system. The file system consists of a hierarchy of folders. Each folder may contain a set of files. Prepare a class diagram being the model of the system, and give few examples of object diagrams that are instances of the class diagram.

3. Complete the class diagram below by inserting proper multiplicities at association ends. Next, generate few examples of object diagrams which are instances of the class diagram.



4. Design a class diagram modeling a family of geometric figures: polygon, rectangle, triangle, square, ellipse, and circle. For each figure define attributes that values enable to represent the figure on the plain. Propose some common operations for the figures.

5. There is the following grammar presented in BNF notation. Terminal symbols are written in Roman letters, non-terminal symbols are written in italics. The initial symbol of the grammar is *term*.

   term ::= *variable* | (*term binarnyOperator term*) | *unarnyOperator term*

   variable ::= *identifier*

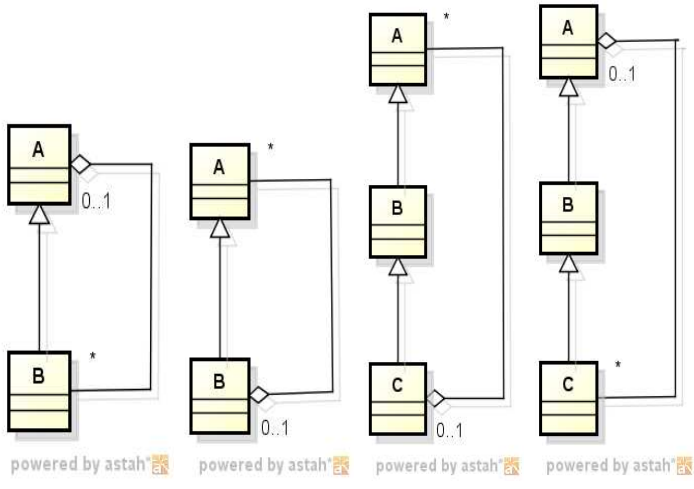
   identifier ::= a | b | … | z

   binarnyOperator ::= + | *

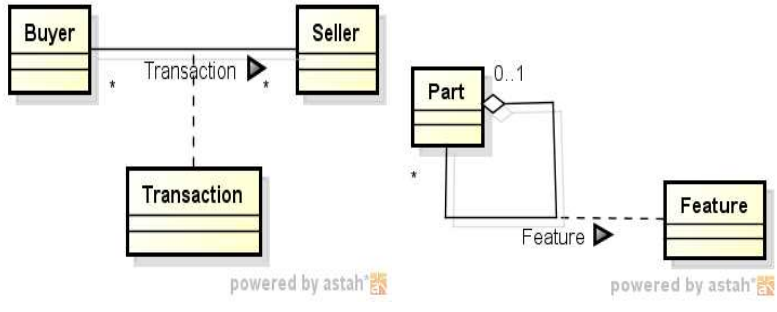
unarnyOperator ::= – | @

Define the class diagram which is a model of the grammar. How to interpret the object diagrams being instances of the designed class diagram?

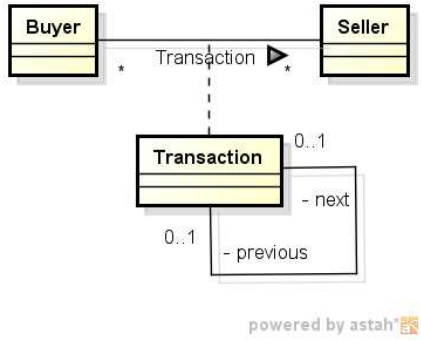
6. Below, there are four class diagrams presented containing generalization and association relationships. Transform each of the diagrams into two separate diagrams – the first containing only generalizations, and the second containing only associations. The two target diagrams should represent the same information as the source diagram.



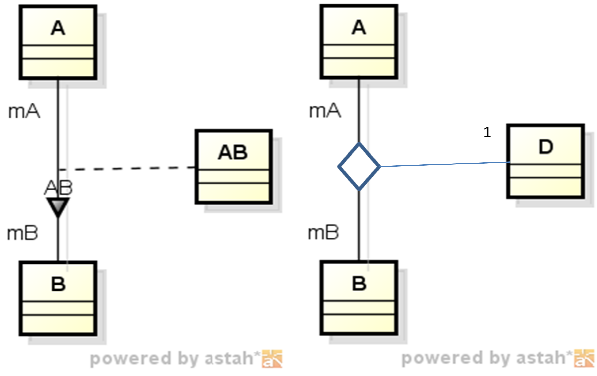
7. There are two class diagrams containing association classes. First, give a reasonable interpretation of the diagrams. Next, try to eliminate the association classes and propose for each of the diagrams equivalent class diagram not having association class. Present exemplary object diagrams that are instances of both class diagrams.



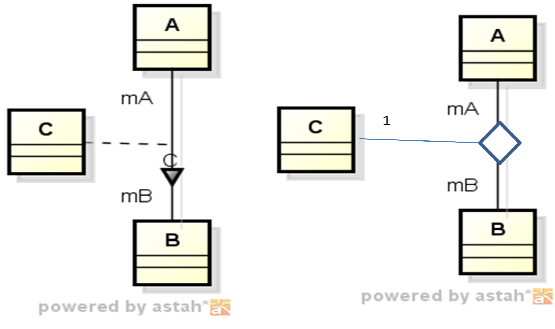
8. Give an interpretation of the class diagram below, and next present exemplary object diagrams.



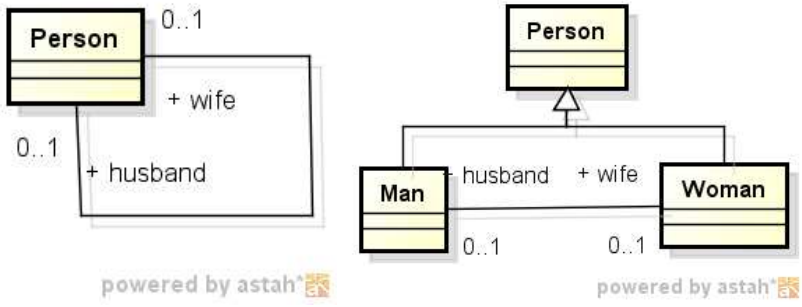
9. Present exemplary object diagrams that are instances of the following class diagram.



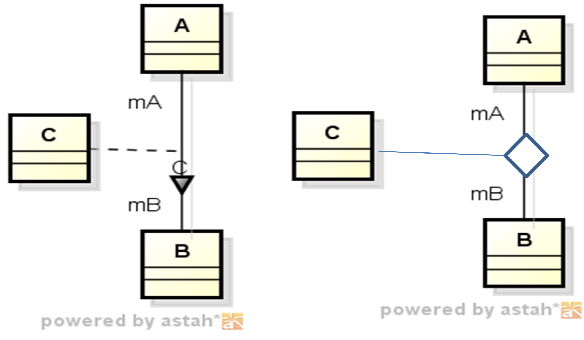
10. Analyze the following two class diagrams. May we consider the diagrams to be equivalent for each multiplicities mA and mB?



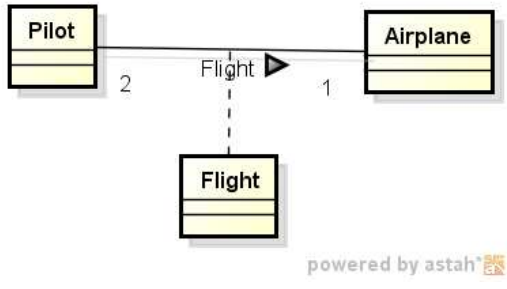
11. Which of the following two class diagrams can be considered as the better one? Explain why.



12. Compare the following two class diagrams, and explain whether they have the same meaning.



13. An airplane has two pilots during each flight. We can model a flight as an association class between Airplane and Person. Discuss whether this is a good way to model the three classes. If not, propose a better design with a better way to use association class.



14. Prepare a class diagram modeling the domain described informally in the following way:
    a) The domain of our interest is a kindergarten.
    b) The kindergarten takes care of five groups of kids.
    c) Each kid belongs to only one group.

d) The minimal number of kids belonging to a group is 10 and maximal – 15.
e) Each group is supervised by two teachers (caretakers).
f) The kindergarten is managed by a person, which is superior to all teachers but is not involved in childcare.
g) Sometimes the kindergarten organizes optional activities for groups of keen kids; the group must have at least 5 kids.
h) The optional activities may have different forms, for example: foreign language learning, horse riding, dancing, music etc., and are taught by additional teacher.

15. Develop a class diagram which is the representation of the following description:
    a) Many people can take part in the conference as participants.
    b) Many papers are presented at the conference.
    b) A conference has two committees: the program and the organizing committee.
    c) Conference committees consist of many members.
    d) Sessions take place in many rooms; in one room many subsequent sessions can take place.
    e) A person who is interested in participating in a conference has to send the registration form.
    f) Papers are reviewed by many reviewers.
    g) Every session has a chairperson.

16. Draw a class diagram modeling the structure of a typical menu applied in programs like, for example, MS Word, MS PowerPoint etc.

17. Draw a class diagram modeling the structure of selected multimedia player.

18. Prepare description of e-mail discussion group. Identify and describe actors and use cases. Next define relationships among them. Describe textually selected actors and use cases.

19. Automatic Teller Machine (ATM) system consists of a number of cash points distributed over a country. Each cash point is linked to a central server located in a bank. The bank co-operates with other banks. Cash point users must have an account at least in one of the co-operating banks. Each user has an identification card that enables to login, authorization to a responsible bank, and next realization of the requested transaction. The transactions should enable checking the account, money collection, and fund transfer among co-operating banks. Consider all transaction recordings as an additional functionality of the system.

First prepare a class diagram modeling the ATM system, and next respective use case diagram. Describe textually selected use cases and present respective scenarios in the form of sequence diagrams.

20. ATM (continuation)

    All ATMs communicate with a central server computer, which serves as clearinghouse for all transactions at individual bank computers.

    Transaction processing at bank computers is handled by individual banks and is out of project scope. Software that runs inside each ATM is not part of the project. However, the interfaces with the bank computers and the ATM are included.

    Bank customers with cash cards may access the ATM system via an ATM. When a cash card is inserted into an ATM, the ATM will read the information on the card, prompt for PIN, validate the PIN, interact with the customer through the ATM, and communicate with the central server to process transactions. The ATM may also dispense cash and print record on request.

    The requirements for the ATM system include the following:

    a) It must keep adequate records of all ATM usage and transactions. It should have the capacity to generate daily reports from these records.
    b) It must include security measures.
    c) It must be able to handle concurrent access to the same account.
    d) It should be open to allow future inclusion of cashier stations, Internet clients, and so forth.

    The constraints include the following:

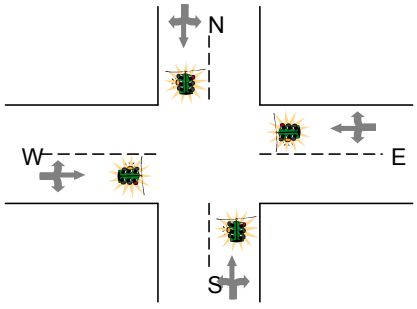
    a) The ATM system is owned by a consortium of banks.
    b) The ATM system software runs on the central server.
    c) Cash cards are issued and relation information is maintained by individual banks. All cards from the same bank share the same six-digit prefix numbers.
    d) All ATMs have the same programming interface.
    e) Each bank has its own programming interface, which may be different from the interfaces of other banks.

21. Elaborate an activity diagram describing selected use cases of ATM system.

22. Elaborate a state diagram modeling operation of car's gear box. The diagram is intended to present recommended behavior of car's drivers. Before preparing diagram construction, prepare the text, which describes recommend rules for changing of gears.

23. Elaborate a state diagram modeling traffic lights control system. We consider a crossing of two roads. Each road has two lanes which enables cars to move in two opposite directions – see fig. below. Assume "usual" traffic regulation rules. The diagram is intended to be the specification for a computer system controlling the traffic at the road crossing.

    Before diagram construction prepare the text, which describes the traffic rules and possible additional requirements for the light control system.



24. Elaborate a state diagram modeling behavior of a vending machine that is used at railway station. The machine is used to sell train tickets. First, a user of the machine, using numeric keyboard, declares the value of requested ticket. Next, the user inserts coins into the machine. When the amount of coins inserted is equal or greater to the declared value, the machine prints the ticket and returns change. The user may withdraw his/her transaction at any moment before ticket printing.

25. Provide a model of an elevator system operating in a high building. First, present a class diagram modeling the static aspect of the system, next elaborate a state machine diagram modeling behavior of elevator control system.

    The modeling is based on the following (partial) description of the services of the lift system. At each level of the n-storey building, except the lowest and highest levels, there are two buttons calling the elevator "up" and "down". In the elevator cab n the buttons are located. Pressing a selected button means the selection of the level of travel. There are also two other buttons for opening and closing the cab's door. Elevator is moving in one direction - up or down - until all

orders for a given direction are realized. After that, the elevator starts to move in the opposite direction until all orders for a new direction are realized. On reaching the floor the elevator stops and automatically unlocks the door. The elevator doors close automatically after 5 seconds from the moment the photocell does not register the movement of passengers at the door. If in a period of 30 seconds no requests arrive, the elevator moves down to the lowest level.

26. Prepare textual description of a car wash machine, and next elaborate a state diagram which could be considered as a specification of system controlling the machine. The description of the domain below is an excerpt from Wikipedia.

    The first conveyorized automatic car washes appeared in the late 1930s. Conveyorized automatic car washes consist of tunnel-like buildings into which customers (or attendants) drive. Some car washes have their customers pay through a computerized point of sale unit, also known as an "automatic cashier", which may take the place of a human greeter. The mechanism inputs the wash prise lock-up code into a master computer or a tunnel controller automatically. When the sale is automated, after paying the car is put into a line-up often called the stack or queue. The stack moves sequentially, so the wash knows what each car purchased. After pulling up to the tunnel entrance, an attendant usually guides the customer onto the track or conveyor. At some washes, both tires will pass over a tire sensor, and the system will send several rollers. The tire sensor lets the wash know where the wheels are and how far apart they are. On other systems the employee may guide the customer on and hit a 'Send Car' button on the tunnel controller, to manually send the rollers which push the car through.

    When the customer is on the conveyor, the attendant (or signage) will instruct the customer to put the vehicle into neutral, release all brakes, and refrain from steering. Failure to do so can cause an accident on the conveyor. The rollers come up behind the tires, pushing the car through a detector, which measures vehicle length, allowing the controller to tailor the wash to each individual vehicle. The equipment frames vary in number and type. A good car wash makes use of many different pieces of equipment and stages of chemical application to thoroughly clean the vehicle.

    The carwash will generally start cleaning with pre-soaks applied through special arches. They may apply a lower pH (mild acid) followed by a higher pH (mild alkali), or the order may be reversed depending on chemical suppliers and formula used. Chemical formulas and concentrations will also vary based upon seasonal dirt and film on vehicles, as well as exterior temperature, and other factors. Chemical dilution and application works in combination with removal systems based on either high pressure water, friction, or a combination of both. Chemical substances, while they are industrial strength, are not used in harmful concentrations since car washes are designed not to harm a vehicle's components or finish.

    The customer next encounters tire and wheel nozzles, which the industry calls CTAs (Chemical Tire Applicators). These will apply specialized formulations, which remove brake dust and build up from the surface of the wheels and tires. The next arch will often be wraparounds, usually made of a soft cloth, or closed cell foam material. These wraparounds should rub the front bumper and, after washing the sides, will follow across the rear of the vehicle cleaning the rear including the license plate area. Past the first wraps or entrance wraps may be a tire brush that will

scrub the tires and wheels. This low piece is often located beneath a mitter (the hanging ribbon-like curtains of cloth that move front to back or side to side) or top wheels. There may also be rocker panel washers which are shorter in size (ranging in size from 18 inches [45 cm] up to 63 inches [160 cm] tall) that clean the lower parts of the vehicle. Most rocker brushes house the motor below the brush hub so they don't inhibit cloth movement and allow the brush to be mounted under a support frame or below a mitter. Some car washes have multiple mitters, or a combination of mitters *and* top brushes.

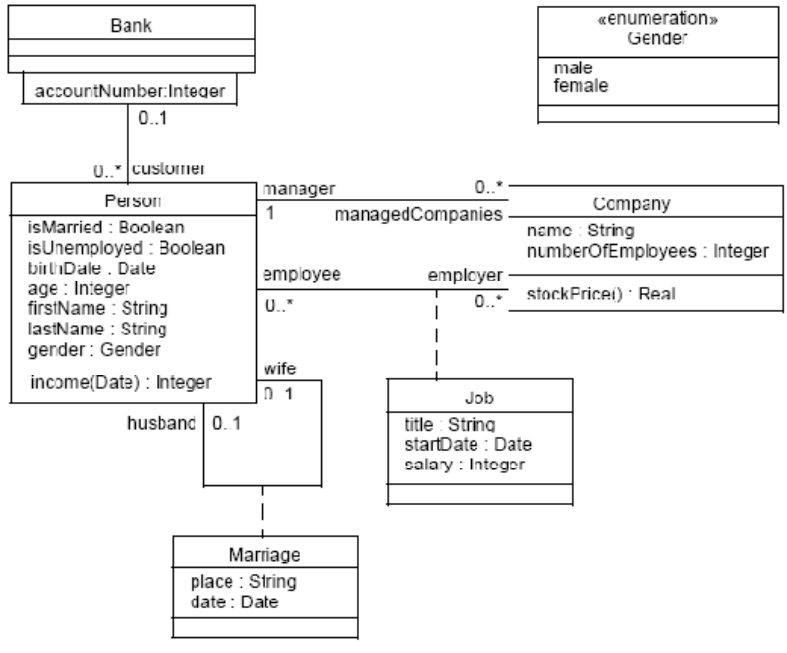
After the mitter or top brush(es) the car may pass through a second set of wraparounds. This may also be where high pressure water streams are used to clean difficult to reach parts of the vehicle. The car generally passes over an under carriage wash and/or has high pressure nozzles pointed at it from various positions. Next may be a tire spinner, high pressure nozzles angled specifically to clean wheels and tires. After the several wash stations the vehicle may go through triple foamers, usually red, blue, and yellow, although colors can be customized with higher end chemical suppliers. The triple foam process includes special cleaners as well as some protective paint sealant.

Some washes have multiple rinse stages, usually offering a protectant as an option.

Protectants vary by manufacturer. Near the rinse is where a tire shining machine is often installed, which is designed to apply silicone tire dressing to the tires. This application makes the tires look good (new, and glossy) and preserves the rubber. Next the vehicle is treated with a drying agent and a final rinse. Many carwashes utilize a "spot free" rinse of soft water that has been filtered of chlorine and sent through semi permeable membranes to produce highly purified water that will not leave spots. After using spot free water, the vehicle is finished with forced air drying, in some cases utilizing heat to produce a very dry car.

In order to avoid paint marking issues, "touchless" (aka "touchfree" or "no-touch") car washes were developed. A touchless car wash uses high water pressure to clean the vehicle instead of brushes, minimizing the chance of surface damage to the vehicle. There are five primary factors to cleaning a vehicle successfully using a touchless system. These five factors are water quality, water temperature, chemistry, time, and water pressure generated by the equipment. If these factors are all set properly, vehicles will come out clean and shiny without the chance of vehicle damage caused by brushes.

27. For the given class diagram express in the OCL the following functions:



a) The function computing the sum of salaries gained monthly by a given person.

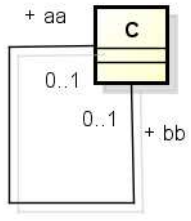
b) The function computing the sum of salaries gained monthly by a married couple.

c) The function computing the sum of salaries gained by a given person in the period of his/her employment in a given company.

d) The function computing the sum of salaries gained by a given person in the period of his/her employment in all companies.

e) The function computing the sum of salaries gained by a married couple since the time of their marriage.

28. For the given class diagram express in the OCL the following constraints:



a) There is exactly one object that has no predecessor neither in the role *a* nor in the role *b*.

b) Each object diagram being an instance of this class diagram has a structure of a tree.

c) Maximal length of the path between any two objects is less than given number *n*.

Miniproject

Project description

The purpose of the project is to build the ATM system software. The overall system consists of multiple ATMs, a central server, the ATM system software, and related interface software. The following is an overview of the project and its concept of operation.

The ATN system software handles transactions among a network of ATMs and bank computers. Each bank has its own computer to maintain its own accounts and to process transactions. All ATMs communicate with a central server computer, which serves as a clearinghouse for all transactions at individual bank computers.

Transaction processing at bank computers is handled by individual bank computers and is out of the project scope. Software that runs inside each ATM is not part of this project either. However, the interfaces with bank computers and the ATMs are included.

Bank customers with cash cards may access the ATM system via an ATM. When a cash card is inserted into a ATM, the ATM will read the information on the card, prompt for personal identification number, validate the identity of the customer, interact with the customer through the ATM, and communicate with the central server to process transactions. The ATM may also dispense cash and print records on requests.

The requirements for the ATM system software include the following:

1. It must keep adequate records of all ATM usage and transactions. It should have the capability to generate daily reports from these records.
2. It must include security measures.
3. It must be able to handle concurrent access to the same account.
4. It should be open to allow future inclusion of cashier stations, Internet clients, and so forth.

The constraints include the following:

1. The ATM system is owned by a consortium of banks.
2. The ATM system software runs over the central server.
3. Cash cards are issued and relevant information is maintained by individual banks. All cards from the same bank share the same six-digit prefix numbers.
4. All ATMs have the same programming interface.
5. Each bank computer has its own programming interface, which may be different from the interfaces of other banks.

Static analysis and design

a) Analyze the description and identify the object classes and their static relationships (associations and generalizations).

b) Describe in natural language the meaning (semantics) of the identified classes and their relationships.

c) Propose attributes for each of these classes.

d) Which of the classes should be declared as persistent?
Explanation: A class is considered to be persistent if each object of the class exists after the thread that created it has ceased to exist.

e) Present a class diagram representing the classes and relationships.
Suggestion: First start from the core class diagram, and next extend it by adding new elements.

Dynamic analysis and design

a) Identify the actors and use cases of the ATM system software.

b) Present a use case diagram representing services of the ATM system software.

c) Describe textually the most important use case and next prepare a sequence diagram representing scenarios of this use case.

d) Propose operations for the previously identified classes.

References

[1] Eriksson H.-E., Penker M., Lyons B., Fado D., *UML™2 Toolkit*, Wiley Publishing, Inc., 2004.

[2] Maciaszek L. A., *Requirements Analysis and System Design*, Second edition, Pearson, Addison-Wesley, 2005.

[3] OMG, Unified Modeling Language Specification, version 2.0. Object Management Group, 2004 (www.omg.org).

[4] Pender T., *UML Bible*, John Wiley & Sons, 2003.

[5] Rumbaugh J., Jacobson I., Booch G., *The Unified Modeling Language – Reference Manual*. Second edition, Addison-Wesley, 2005.

[6] SWEBOK — *Guide to the Software Engineering Body of Knowledge*, ISO/IEC TR 19759:2005(E), First edition 2005-09-15.

[7] Unhelkar B., *Verification and Validation of UML 2.0 Models*, John Wiley & Sons, 2005.

[8] Weilkiens T., Oestereich B., *UML 2 Certification Guide. Fundamental and Intermediate Exams*, Elsevier 2007.