CHAPTER 1

Unified Modeling Language: A Standard for Designing a Software

INFORMATION IN THIS CHAPTER:

- UML diagrams
- Formalism of UML
- Benefits and disadvantages of applying UML
- UML improvement options

1.1 INTRODUCTION

Unified Modeling Language—abbreviated as UML—is a graphical language officially defined by Object Management Group (OMG) for visualizing, specifying, constructing, and documenting the artifacts of a software system [106]. An artifact in software development is an item created or collected during the development process (example of artifacts includes use cases, requirements, design, code, executable files, etc.). UML offers a standard way to write system's blueprints, including conceptual things such as business processes and system functions as well as concrete things such as programming language statements, database schemas, and reusable software components [37]. Despite that UML is designed for specifying, visualizing, constructing, and documenting software systems, it is not restricted only for software modeling. UML has been used for modeling hardware, and is used for business process modeling, systems engineering modeling and representing organizational structure, among many other domains [125].

The first UML specification (version 1.1) was published by OMG at 1997. Since then there has been continuously ongoing work to improve both the language and its corresponding specification. Additionally, we should admit that UML versions 1.4.2 and 2.4.1 have been published under International Organization for Standardization (ISO) [44] and International Electronical Commission (IEC) [43] as a

Topological UML Modeling. DOI: http://dx.doi.org/10.1016/B978-0-12-805476-5.00001-0

standard. In year 2005, the version 1.4.2 was published as ISO/IEC 19501:2005 [46]. Following in year 2012, the version 2.4.1 was published as ISO/IEC 19505-1 [47] and ISO/IEC 19505-2 [48]. You should ask—why there are two separate ISO/IEC standards for single UML version? The answer hides in fact that beginning with UML version 2.0 its specification was divided in two parts (i.e., two separate documents)—so-called *Infrastructure* and *Superstructure*. Accordingly, the ISO/IEC standard is based on this separation. But what a surprise—UML version 2.5 specification [79] again is a single document.

During the two major and a number of revision versions of UML, the definition of UML is evolving. UML version 2.4.1 specification [77,78] defines the language as follows: "*UML is a visual language for specifying, constructing, and documenting the artifacts of systems. It is a general-purpose modeling language that can be used with all major object and component methods, and that can be applied to all application domains (e.g., health, finance, telecom, aerospace) and implementation platforms (e.g., J2EE, .NET).*"

The UML originally was developed in middle of 1990s as a combination of previously competing object-oriented analysis and design approaches:

- *Booch method* by Booch [13],
- *Object-Modeling Technique (OMT)* by Rumbaugh, Blaha, Premerlani, Eddy, and Lorensen [105],
- *Object-Oriented Software Engineering (OOSE)* by Jacobson, Christerson, Jonsson, and Overgaard [49], and
- Other contributions to modeling complex systems, e.g., *statecharts* by Harel [41].

The first version of UML (version 1.1) was approved by OMG in year 1997 [71]; afterwards UML has been revised with several releases (UML 1.3, 1.5, 2.0, 2.1.1, 2.1.2, 2.2, 2.3, 2.4.1, and 2.5 [81]) by fixing some problems and adding new notational capabilities. The latest standard released by OMG is UML version 2.5 (UML version 2.0 is a major rewrite of UML 1.x ("x" denotes the main version and any subversion of specification) and was released in 2015).

The UML became widely accepted as the standard for object-oriented analysis and design soon after it was first introduced [54] and still remains so today [22,103]. Since the release of first UML version a

large number of practitioner and research articles and dozens of textbooks have been devoted to articulate various aspects of the UML, including guidelines for using it. In fact, since the UML specification is a specification and thus it is written in a manner to specify every aspect of the language's constructs, it does not contain guidelines on how to apply the language elements in real-life situation. So just reading the language's specification does not give an insight of its application. We advise to read UML specification together with guidelines describing approach or methodology of applying UML diagrams throughout software development lifecycle. Since the UML as a language includes 14 kinds of diagrams and many elements building them up, the scope of UML-related research areas is wide:

- Formalization of UML semantics (e.g., [31,42] (both after UML 1.1 was released), and [122] (after UML 2.0 was released)),
- Extending the UML (e.g., [64,99], and review of a number of UML profiles developed by different researchers and groups [103]),
- Formalizing the way, the UML diagrams are developed (e.g., [88,96]),
- Ontological analysis of UML modeling constructs (e.g., [125]),
- Empirical assessments (e.g., [22,32]),
- Analysis of the UML's complexity (e.g., [30,111,112]),
- Difficulties of learning UML (e.g., [113]) and how to avoid them (e.g., [11]),
- Transformations between UML diagrams (e.g., [61,57,66]),
- Software code generation and related issues with generated code quality (e.g., [59,108,118]), and
- Experiments that evaluate aspects of UML models effectiveness (e.g., [17]).

The large number of researches regarding UML evolving and strengthening is caused by the basis on which UML was developed. According to Dobing and Parsons [22] the "*UML was not developed based on any theoretical principles regarding the constructs required for an effective and usable modeling language for analysis and design; instead, it arose from (sometimes conflicting) 'best practices' (e.g., Booch, OMT, OOSE) in parts of the software engineering community.*"

The next section of this chapter introduces in brief with the diagrams found in UML specification. The review of elements that build up UML within this chapter is based on UML version 2.4.1

specification which is divided into two volumes (both volumes cross-reference each other and the specifications are fully integrated):

- *Infrastructure* [77]—defines a metalanguage core that can be reused to define a variety of metamodels, including UML, Meta-Object Facility (MOF) [74], and Common Warehouse Metamodel (CWM) [70]; and the core metamodel on which the *Superstructure* is based. The *Infrastructure* of the UML is defined by the *InfrastructureLibrary* package which consists of two subpackages:
  1. *Core*—contains core concepts which are used when metamodeling and
  2. *Profiles*—defines the mechanisms that are used to customize metamodels.
- *Superstructure* [78]—defines the notation and semantics for diagrams and their elements. The *Superstructure* metamodel is specified by the UML package, which is divided into a number of subpackages that deal with structural and behavioral modeling.

Although the UML specification 2.5 has been extensively rewritten from its previous version 2.4.1 by combining together the *infrastructure* and *superstructure* parts, the metamodel itself remains unchanged from UML 2.4.1 superstructure [79]. Thus, the amount and types of UML diagrams have not changed from version 2.4.1 to version 2.5.

1.2 UNIFIED MODELING LANGUAGE DIAGRAMS

A system should be specified from different viewpoints to get a broader and more comprehensive insight and understanding of the intended software. The more efforts are added to consider the system from different viewpoints at the very beginning of the software development lifecycle, the more risk of producing irrelevant or unnecessary software system is reduced or even avoided [98]. Another benefit of such approach hides in fact of reducing the need of overworking or overdoing things that seem to be completed. The bunch of UML diagrams allows us to take a look at the system from various viewpoints. "*A diagram is the graphical presentation of a set of elements, most often rendered as a connected graph of vertices (things) and arcs (relationships). A diagram is a projection into a system*" [15]. Additionally, the UML diagrams can be developed in different abstraction levels showing the most important aspects in each one of the level. The abstraction levels

allow to describe and to show the system with the appropriate information amount for the stakeholders and developers.

In context of software development, there are five complementary views that are important in visualizing, specifying, constructing, and documenting software architecture [106]:

1. The use-case view,
2. The design view,
3. The interaction view,
4. The implementation view, and
5. The deployment view.

Each of these views involves structural modeling (static aspect of a system) as well as behavioral modeling (dynamic aspect of a system). Let's take a look at the diagrams included in UML specification and evolution history of them. The very first UML specification (version 1.1) released by OMG in 1997 [71] contained only nine diagram types. By evolving the UML, the amount of diagram types has also grown. The newest UML specification (version 2.5) [79] released by OMG in year 2015 contains 14 diagram types (see Fig. 1.1) which are organized in two major diagram types and one subtype:

- *Structure diagrams*—aimed to visualize, specify, construct, and document the static aspect of a system,
- *Behavior diagrams*—are used to visualize, specify, construct, and document the dynamic aspect of a system (modeling dynamic aspect of a system can be considered as representing its changing parts),
  - *Interaction diagrams*—show interaction, consisting of a set of objects and their relationships, including the messages that may be dispatched among them. By using interaction diagrams, it is possible to reason about flow of control within an operation, a class, a component, a use case, or the system as whole.

All of UML diagrams are briefly described in following subsections: Section 1.2.1 describes structure diagrams and Section 1.2.2—behavior diagrams.

1.2.1 Structure Diagrams

The UML's structural diagrams are aimed to visualize, specify, construct, and document the static aspect of a system [15,78,79].

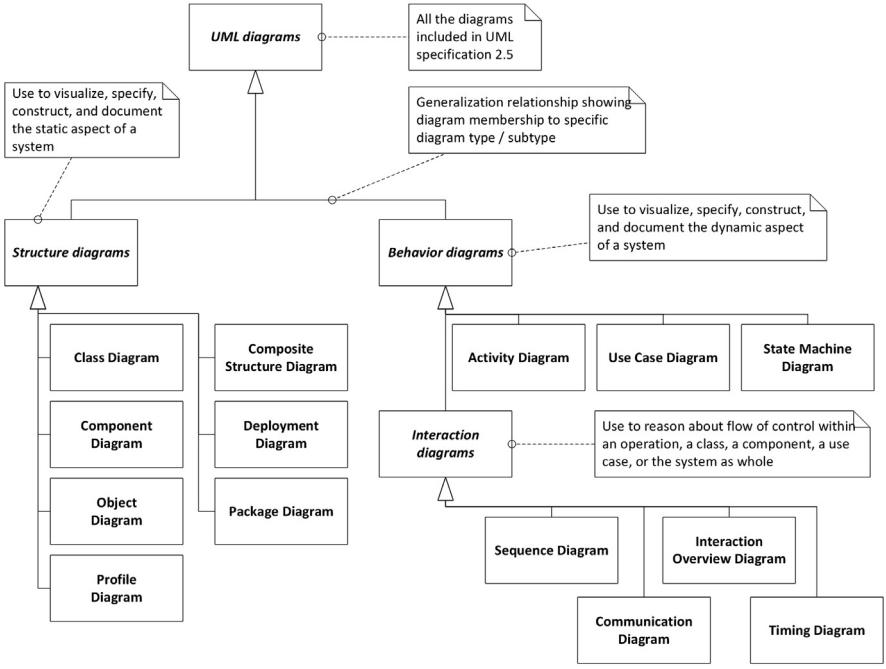

Figure 1.1 Diagram types included in UML specification version 2.5.

All structure diagrams are listed and described in the subsections of this section. Each subsection name denotes corresponding UML diagram and the contents include description, main elements, UML version in which the diagram is included, and an example of a diagram.

1.2.1.1 Class Diagram

Class diagram is the most common diagram found in object-oriented systems and it is used to illustrate the static viewpoint of a system. It shows a set of classes, interfaces, and their relationships. Class diagrams are also the foundation for a couple of related diagrams: component and deployment diagrams. The *class diagram* is included in UML specification since the first (1.1) version.

The class diagram includes following elements:

- *Class*—a template for creating objects, providing specification of attributes and operations that an instance of the class can complete. In the context of programming languages, the operation

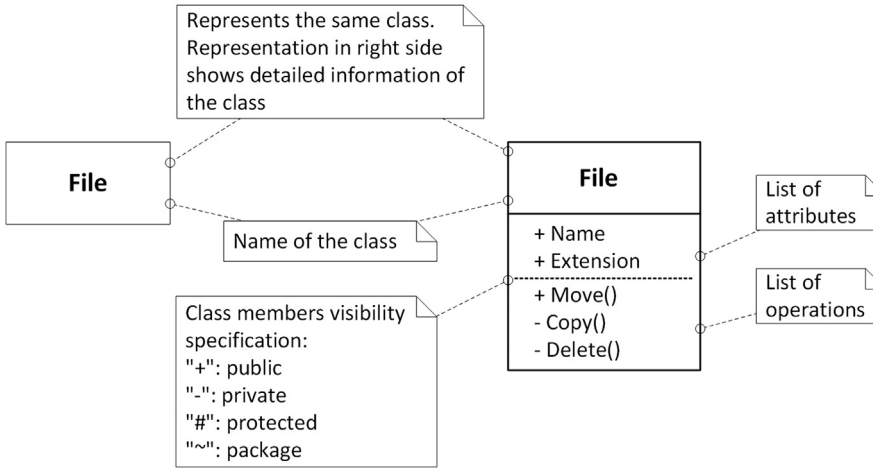

Figure 1.2 Example of a class representation.

is addressed as function or procedure. When an object is created by a constructor of the class, the resulting object is called an instance of the class. In certain circumstances it is needed to also model instances of classes at the specific moments in time—for those cases Object diagram should be used (see Section 1.2.1.2). See example of class representation using different notations in Fig. 1.2.

- *Interface*—specifies a contract consisting of a set of coherent public attributes and operations for a class. Any instance of a class that realizes the interface must fulfill that contract. Since interfaces are declarations, they are not instantiable. Instead, an interface specification is implemented by an instance of a class. Each class may implement more than one interface and each interface may be implemented by a number of different classes. Some object-oriented programming languages, such as .NET [68] and Java [19] uses interfaces to "implement" multiple inheritance. Multiple inheritance denotes that particular child can have more than one parent. Fig. 1.3 shows example of class diagram having multiple interfaces and classes realizing and requiring them.
- *Relationship*—a concept that specifies some kind of relationship between elements, i.e., it references one or more related elements. The relationship can model either physical or logical relations. The UML specification contains definition of multiple relationships that

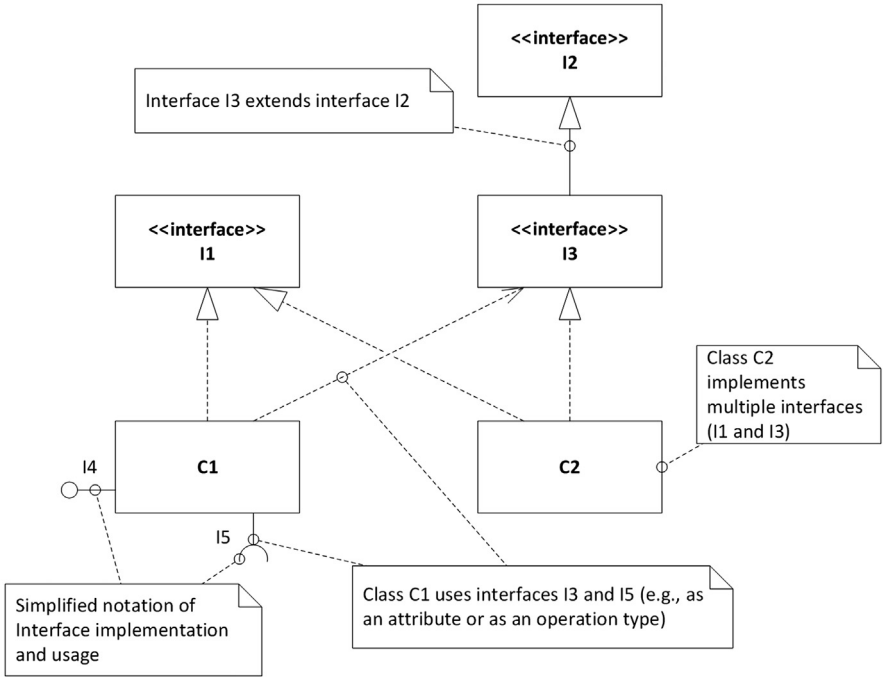

Figure 1.3 Class diagram showing implementation and usage of interfaces.

can exist between elements. The most important of relationships are the following three which are illustrated in Fig. 1.4:

1. *Generalization*—relates generalized classes to specialized classes, i.e., it shows the parent-child relations or the superclass-subclass relations.
2. *Association*—structural relationships among classes showing the physical structure of things. For example, transport vehicle has fuel tank. It shows also the multiplicity between things. For example, car has four tires.
3. *Dependency*—states that one entity uses the information and services of another entity. For example, car uses petrol station to fill fuel tank.

- *Enumerator*—used to specify definite set of available values. For example, Boolean can be specified as enumerator with two values *true* and *false* respectively (take a look at Fig. 1.5).

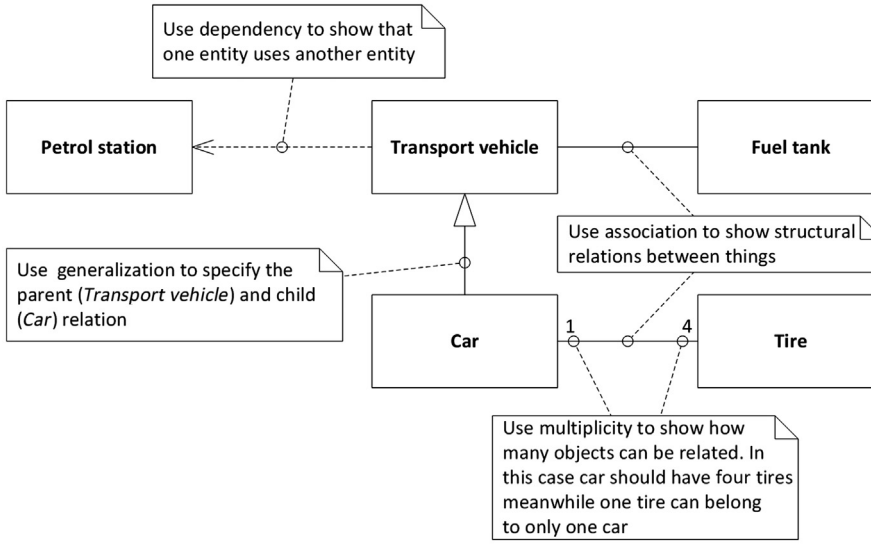

Figure 1.4 Example of relationships between classes.

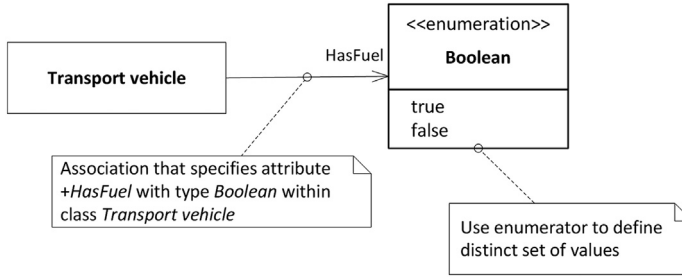

Figure 1.5 Enumerator definition for Boolean and its application.

Fig. 1.6 shows an example of class diagram which is developed as a part of data synchronization system development project which is later discussed in details in Part III, Topological UML Modeling Explained. It contains one abstract class (*DataSource*) with specialized classes of it (*SourceDataSource* and *TargetDataSource*), additional three classes and an enumerator which are tied together by associations and dependencies.

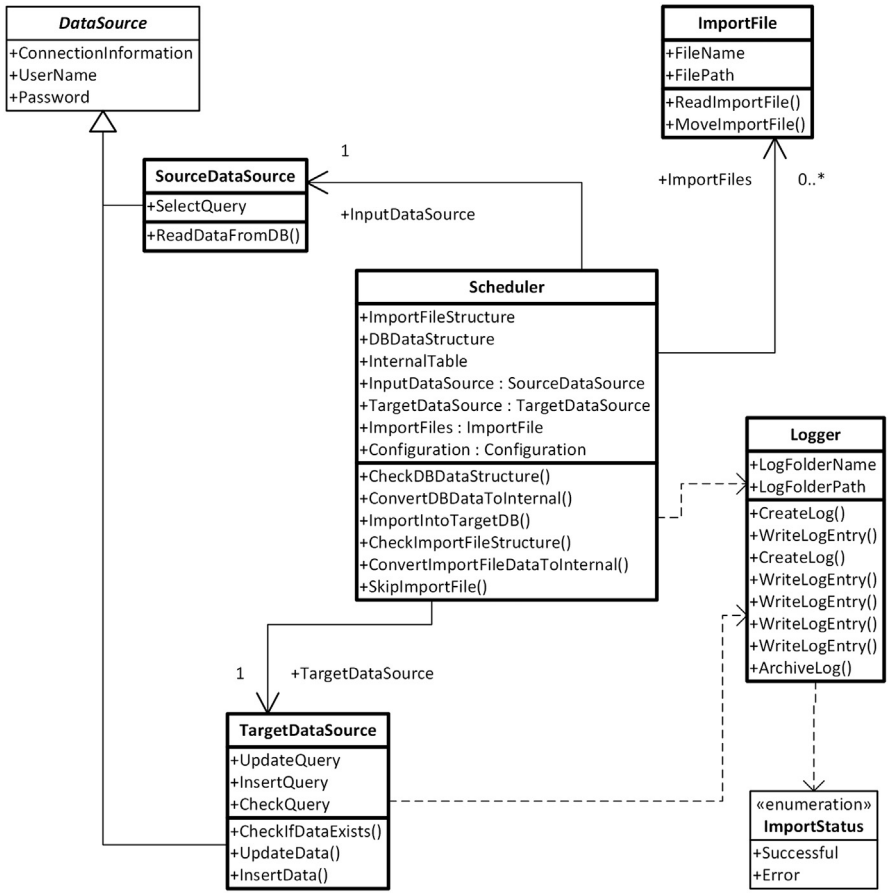

Figure 1.6 Class diagram of enterprise data synchronization system.

1.2.1.2 Object Diagram

While objects are instances of classes, an object diagram is a snapshot of the objects in a system at a specific point in time including the relations (links) between them. You should use object diagrams whenever it is needed to model or take a look on the values of attributes and state of the object at different stages during the execution of the software. They are very useful to model step-by-step execution of complex process or calculation operation. You can take a look at the initial stages of objects, during the process, and of course the final stages of objects to see the whole picture. Since it shows instances rather than classes, it is also called an instance diagram.

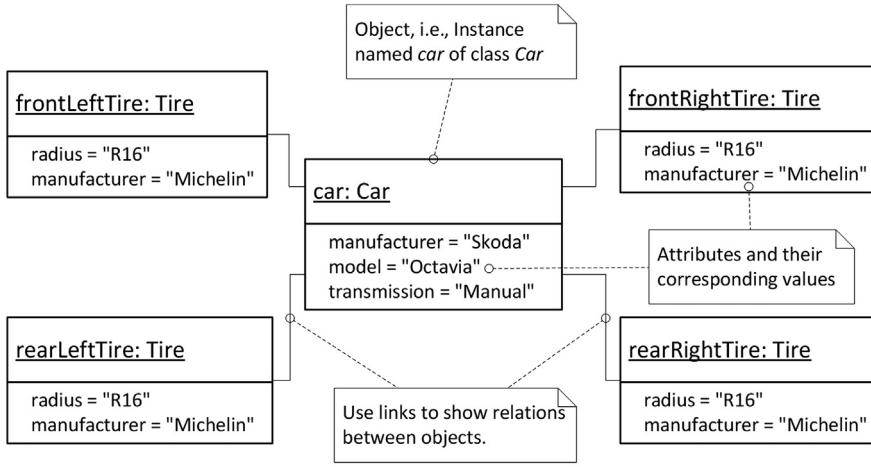

Figure 1.7 Object diagram showing car and all of its four tires.

The object diagram is included in UML specification since the first (1.1) version and it commonly contains following main elements: *object* and *link,* as you can see in Fig. 1.7. An association in class diagram becomes a link in object diagram. The example shows a car with four tires—the snapshot includes data about car itself (manufacturer and model) as well as tires (radius and manufacturer).

1.2.1.3 Package Diagram

The package is general-purpose mechanism for organizing modeling elements into groups, i.e., classes in groups or in namespaces and the relationships between them. Packages are used to arrange modeling elements (e.g., classes, interfaces, components, nodes, diagrams, collaborations, use cases, and other packages) into larger chunks that it is possible to manipulate them as a group. Packages can also be used to present different views of system's architecture and they can be incorporated into components to build up their internal structure. Well-designed packages group elements that are semantically close and tend to change together. Package diagram was first introduced in UML version 2.0. The elements that build up a Package diagram are as follows:

- *Name*—all packages should have a name that distinguishes them from other packages and allows to identify it. Since packages can be graphically displayed with slight differences—the name of package typically is placed in the middle of the package, or in the upper right side of the package (see examples in Fig. 1.8).

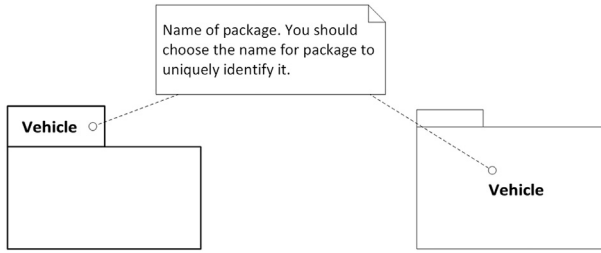

Figure 1.8 Name of package.

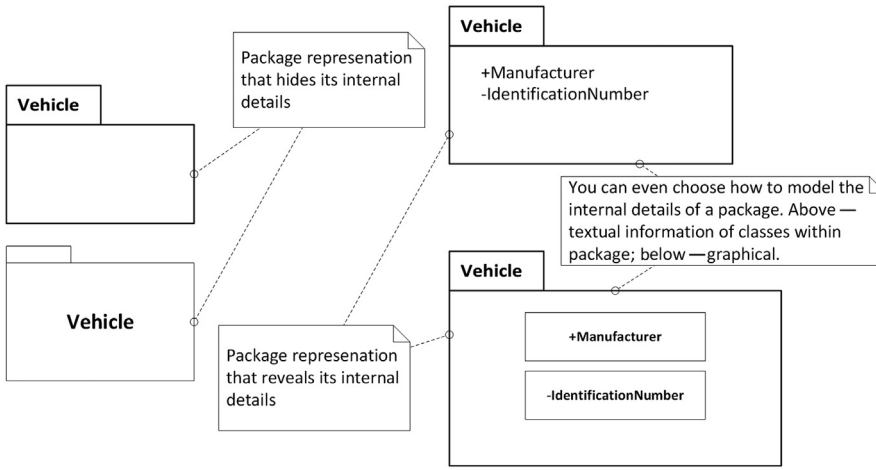

Figure 1.9 Packages.

- *Package*—a logical boundary of modeling elements to group together. You can either choose to show or hide the contents of a package. One way of how to organize packages—the commonly used namespaces of the classes. Graphically a package is drawn like a closed folder. Take a look at Fig. 1.9 to see the different ways of drawing package and revealing/hiding its details.
- *Element*—a package contains elements, i.e., classes, interfaces, components, nodes, use cases, diagrams, and other packages grouped into it. Every element that is included in the package is defined within it. If we destroy the package, all the elements within it are destroyed as well. Right side of Fig. 1.9 shows nested elements of package Vehicle while Fig. 1.10 gives an example of modeling package within package.

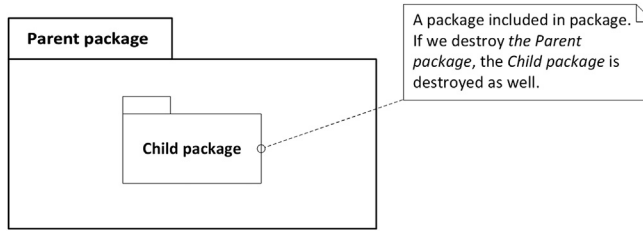

Figure 1.10 A package within a package.

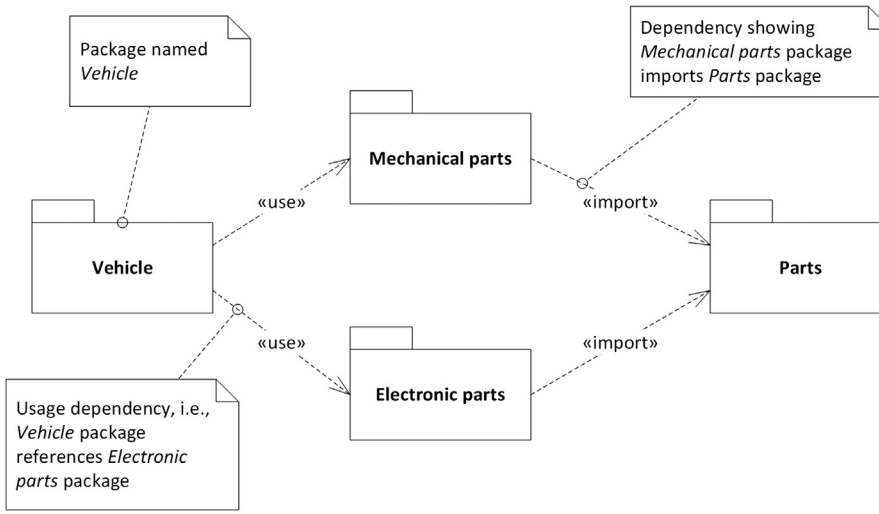

Figure 1.11 Package diagram showing packages and their relations.

- *Relationships*—several kinds of relationships are used when modeling packages: *import, export*, and *dependency use* relationship. If elements of one package uses elements of another package, then *import* or *use* relationship is used. If package contains public elements (e.g., public class *manufacturer* in Fig. 1.9) —it is called an *export*.

As you can see in Fig. 1.11, the package diagram is very useful to avoid unnecessary circular code references in the implementation. We should avoid circular references to enable normal code compilation—with such references both referenced classes must be recompiled every time either of them is changed. The role of package diagram is to identify as early as possible the situations where circular references could be used as a temporal solution (as we all know the temporal solutions

are commonly with the most longest life) and to redesign the solution with the least efforts required.

1.2.1.4 Component Diagram

A component diagram shows the internal parts, connectors, and ports that implement a component. When the component is instantiated, copies of its internal parts are also instantiated. The UML component diagram shows how a software system will be composed of a set of deployable components—dynamic-link library (DLL) files, executable files, or web services—that interact through well-defined interfaces and which have their internal details hidden.

The *component diagram* is included in UML specification since the first (1.1) version and it contains the following elements:

- *Interface*—specifies a contract consisting of a set of coherent public attributes and operations for a class. Any instance of a class that realizes the interface must fulfill that contract. Since interfaces are declarations, they are not instantiable. Instead, an interface specification is implemented by an instance of a class. Each class may implement more than one interface and each interface may be implemented by a number of different classes. Fig. 1.12 shows example of component diagram having multiple interfaces and component providing and requiring them.
- *Component*—represents a modular part of a system that encapsulates its contents, it defines its behavior in terms of provided and required interfaces. As such, a component serves as a type whose conformance is defined by these provided and required interfaces (encompassing both their static as well as dynamic semantics). One component may therefore be substituted by another only if the two

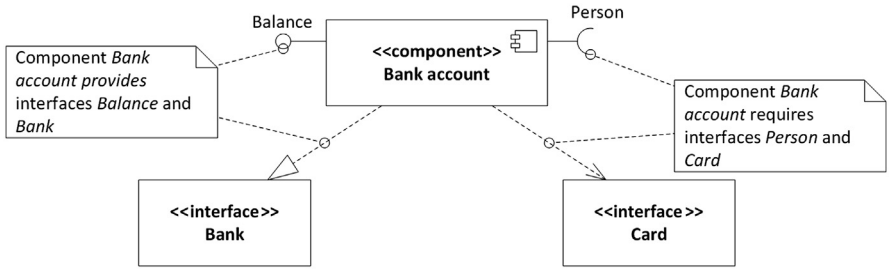

Figure 1.12 Component diagram showing component "Bank account" that provides and requires specific interfaces.

are type conformant. An example of component with provided and required interfaces is given in Fig. 1.13.

- *Port*—an explicit window into an encapsulated component. All of the interactions into and out of such component pass through ports. Each port provides or requires one or more specific interfaces. There can be multiple ports providing or requiring the same interface. It allows greater control over implementation and interaction with other components. Fig. 1.14 considers component with two named ports that each requires the same interface. The first port *Cash withdrawal* is used when bank's client takes out cash from automated teller machine (ATM) using his card. The other port named *Payment in shop* is used when making payments with card at shop.
- *Internal structure*—used to specify structure of a complex component, i.e., typically components are composed of smaller components thus building up the system. Fig. 1.15 gives example of an internal structure representation.
- *Part*—a component that builds up internal structure of a more complex component. You can consider part as a subcomponent. An example of showing parts within component is given in Fig. 1.15.

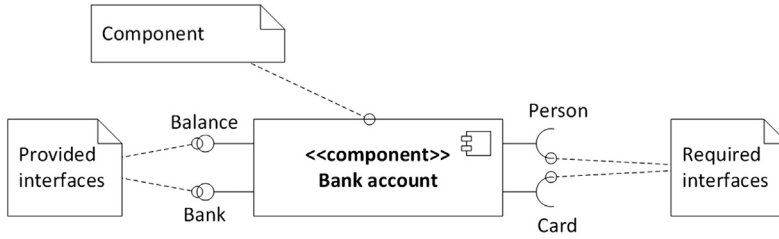

Figure 1.13 Example of component.

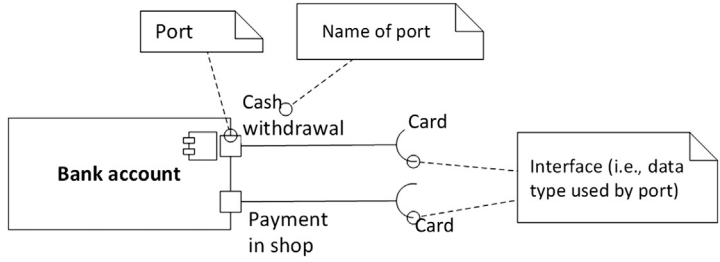

Figure 1.14 Component with two ports.

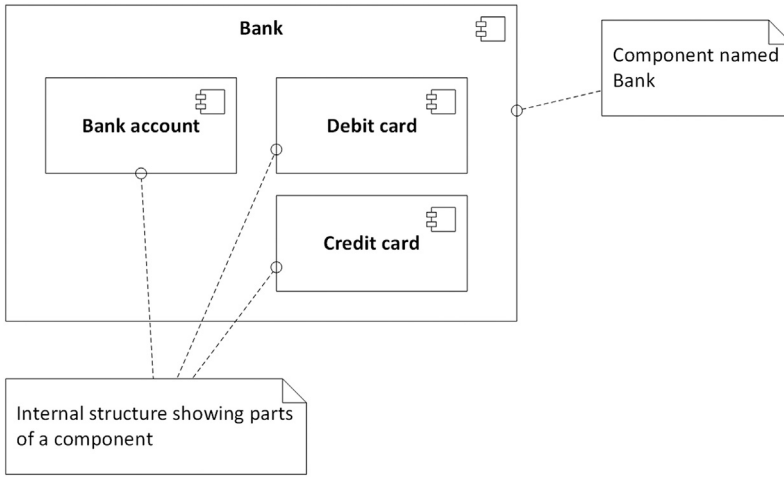

Figure 1.15 Internal structure of a component "Bank."

- *Connector*—a relation between ports of components. If one port provides interface required by the other port, they can be linked together. There are several types of connectors that we can draw between parts and components: A direct connector links together two ports of parts, connector by interfaces links together two ports by relating together required—provided interfaces using lollipop and socket notation, and delegation connector which links together port of a part and port of a component thus providing interface (i.e., provides services to other components) or requiring interface (i.e., consuming services of another component). Take a look at an example showing all three kinds of connectors in Fig. 1.16.

There is an additional diagram type within UML 2.x versions—*composite structure diagram*. It shows the internal structure of a class or collaboration and uses interface, component, port, and connector to show the internal structure. The difference between components and composite structure is tiny.

1.2.1.5 Deployment Diagram

A deployment diagram commonly is used to specify how the components of a system are distributed across the infrastructure and how they are related together. To model such a view deployment diagrams uses just two kinds of elements—nodes (i.e., components of a system

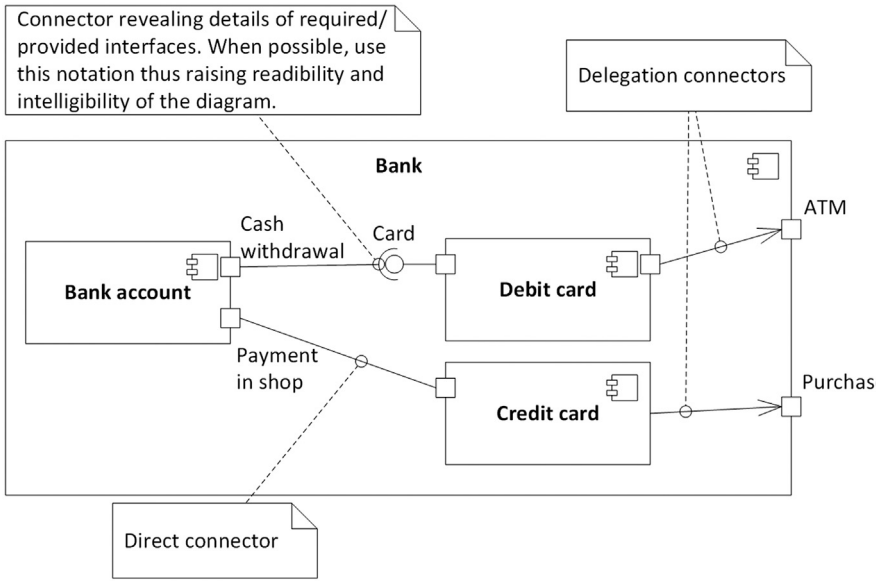

Figure 1.16 Example of connectors for parts and components.

or the infrastructure artifacts) and relationships that links nodes together. Deployment diagram shows the static deployment view of architecture. Deployment diagram typically is related to a component diagram in a way that nodes typically encloses one or more components. A deployment diagram is a diagram that shows the configuration of runtime processing nodes and the artifacts that live on them.

The *deployment diagram* is included in UML specification since the first (1.1) version and it includes following elements:

- *Node*—artifact of a software system (i.e., a component) or artifact of an infrastructure (e.g., server, network segment, sensors, etc.).
- *Relationship*—used to tie together nodes within deployment diagram thus building up a graph consisting of arcs (relationships) and vertices (nodes). Typically, association and dependency relationships are used.

An example of deployment diagram showing servers, their relationships and communication with client devices is shown in Fig. 1.17, which consists of three infrastructure layers—web-front-end servers, application server and data-storage server. The web-front-end servers contains all the components needed to render html pages on client

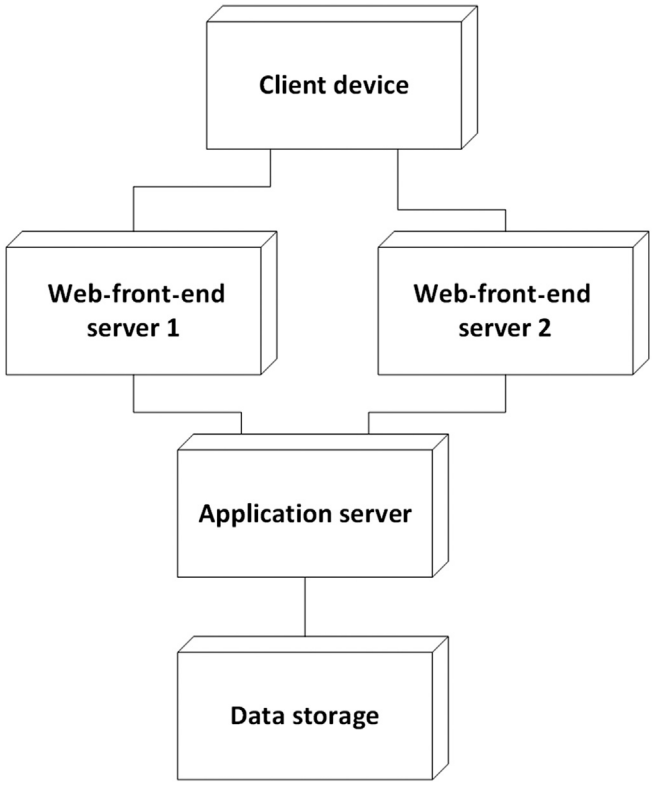

Figure 1.17 Deployment diagram showing servers as nodes and relations between them.

browsers, including communication with application layer to read and manipulate with data. The application layer (node of *Application server*) is hidden from the client thus improving security aspect of the deployment and contains all the artifacts that are required to work with data (i.e., implements read and write operations) while the *Data storage* node is responsible for storing and retrieving data bytes from the media. The *Client device* holds components such as internet browser for user to be able to interact with the system. The *Client device* node adds more understanding to the deployment diagram showing that the client can access data only using the web-front-end layer.

1.2.1.6 Profile Diagram

The profile diagram contains mechanisms that allow extending and adapting metaclasses from existing metamodels for different purposes; includes the ability to tailor the UML metamodel for different platforms or domains. The profile diagram is the most younger diagram

among the diagrams of UML—it is introduced only in version 2.2 thus finally allowing legally extending the UML metamodel—the profiles mechanism is consistent with the MOF. While there were no profile diagram the solution to "extend" UML was to rewrite the specification and add required elements. But this created additional headaches for bringing the new thing into practice—there were no standardized tool support for such new language specification. The UML profile diagram consists of following elements:

- *Metamodel*—a referenced model that is extended through the profile (e.g., UML).
- *Reference*—a dependency relationship with attached stereotype "*reference*" that is directed from profile to referenced metamodel.
- *Profile*—a special package that extends a referenced metamodel by adding stereotypes to it. Like packages in package diagram can be drawn at different abstraction levels revealing or hiding its content, the profile can be drawn in the same manner.
- *Metaclass*—a class that is extended by a stereotype. The metaclass is represented with the same node as regular class by attaching stereotype "*metaclass*."
- *Stereotype*—extends existing UML vocabulary by adding a new element to it and it describes how an existing metaclass can be extended enabling the integration of platform or domain-specific terminology or notation in the modeling language (a set of stereotypes build up the profile). A stereotype extension is used to indicate that the properties of a metaclass are extended through a stereotype. The stereotyped class is represented with the same node as regular class by attaching stereotype "*stereotype*."
- *Extension*—a special binary association, extension end is used to tie an extension to a stereotype when extending a metaclass. Extension relationship is directed from stereotyped class to the metaclass it extends.
- *Profile application*—a dependency relationship with attached stereotype "*apply*" between a package and a profile that allows to use the stereotypes from the profile in the model elements of the source package. Profile application relationship is directed from package that applies profile to the profile package.

Profiles in more details are explored in Chapter 3, Adjusting Unified Modeling Language, while an example showing generic profile with name *TestML* is given in Fig. 1.18. The example of profile extends

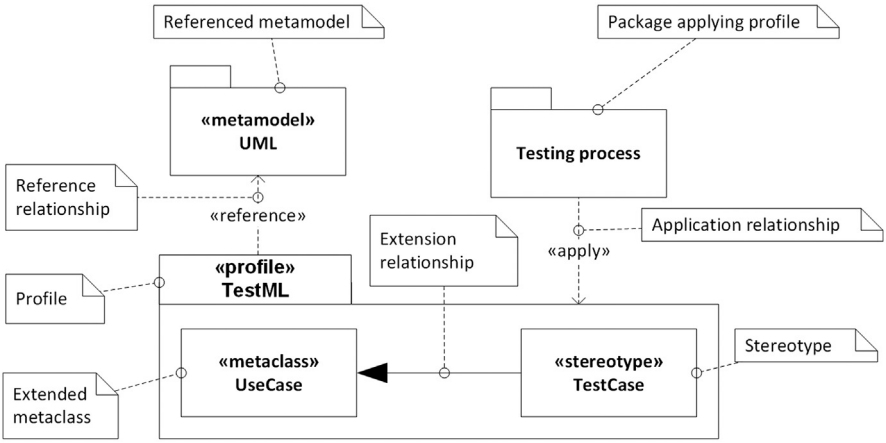

Figure 1.18 Profile diagram showing example of profile and profile application.

UML by adding stereotype *TestCase*. The stereotype *TestCase* extends metaclass *UseCase* by using extension relationship. The profile is applied by the package *Testing process*.

1.2.2 Behavior Diagrams

The UML behavior diagrams are used to visualize, specify, construct, and document the dynamic aspect of a system (modeling dynamic aspect of a system can be considered as representing its changing parts). Behavior diagrams include activity diagram, use case diagram, state diagram, and interaction diagrams. Interaction diagrams are a special subset of behavior diagrams and they are sequence, communication, interaction overview, and timing diagrams. An interaction diagram shows an interaction, consisting of a set of objects and their relationships, including the messages that may be dispatched among them. These diagrams share the same underlying model, although in practice they emphasize different things. By using interaction diagrams, it is possible to reason about flow of control within an operation, a class, a component, a use case, or the system as whole in two ways: (1) focusing on how messages are dispatched across time and (2) focusing on the structural relationships among the objects in an interaction and then consider how messages are passed within the context of that structure [15,37,78,79].

All behavior diagrams, including interaction diagrams, are listed and described in the subsections of this section. Each subsection name

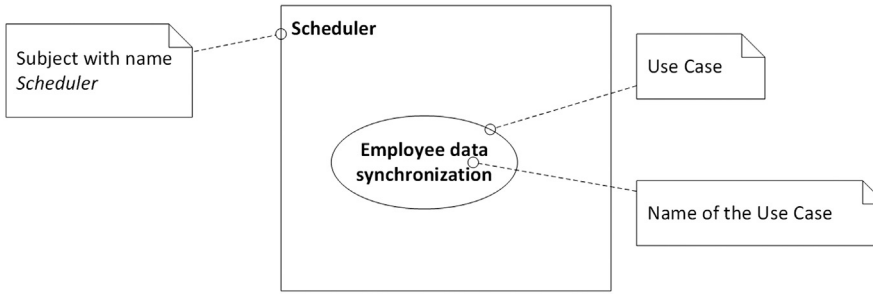

Figure 1.19 Example of use case *within* subject.

denotes corresponding UML diagram name and the contents includes description, main elements, UML version in which the diagram is included, and an example of a developed diagram.

1.2.2.1 Use Case Diagram

Use case diagram shows a set of use cases and actors and their relationships; it is used to organize and model the dynamic aspect—required usages—of a system. Each use case in use case diagram typically is supplemented by a full use case specification—a written statement detailing the preconditions (what must be true before the use case is performed), the sequence of events, including alternate sequence of events in case of exception or specific conditions, and the post-conditions (what must be true after the use case has completed).

The use case diagram is included in UML specification since its first (1.1) version and it includes following elements:

- *Subject*—usually it is a system or subsystem, i.e., a set of use cases together describes the behavioral aspect of the subject under consideration. Fig. 1.19 contains example showing subject.
- *Use case*—it describes what a system or subsystem is doing, it does not include and does not specify how it is doing it. Commonly specification of use case is written in natural language, structuring the description as sequential steps performed by the actors and the subject involved in the use case scenario. Use case scenario includes both the main scenario and an alternative scenario which is used in the case of exception or specific conditions becoming true during the execution of it. The use case is drawn as an ellipsis showing the name of use case within it (see example in Fig. 1.19).

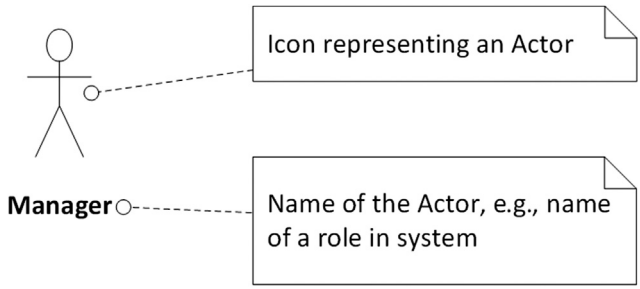

Figure 1.20 Actor.

- *Actor*—represents a coherent set of users, roles, external systems that interacts with the subject through specific use cases. Actors model entities that are outside the system. Taking a look from the subject's viewpoint—the external systems are also drawn as actors interacting with the use case. Fig. 1.20 shows a typical graphical representation of an actor although you can create a stereotype of the actor to visually represent in any form or image according to needs to better illustrate the actor.
- *Relationship*—there are three types of relationships used within use case diagram. The first one is between use cases showing dependency *extend* and *include* relations. The extend dependency is used to show an alternate flow of events in the case if specific conditions are met, e.g., an alternate scenario, while the include dependency is used to specify a common scenario included in multiple use cases (it avoids the duplications of the same scenarios/requirements). The second type of relationships is between use cases and actors, typically represented as associations showing the communication link between actor and use case. The last type is between actors—typically generalization is used to show the parent-child relationships between roles.

Fig. 1.21 shows an example of diagram which is developed as a part of data synchronization system development project described in detail in Part III, Topological UML Modeling Explained.

1.2.2.2 Activity Diagram

Activity diagram is used to model dynamic view of a system. It shows the control flow from step to step, i.e., from activity to activity. An activity shows set of actions, the sequential or branching control flow,

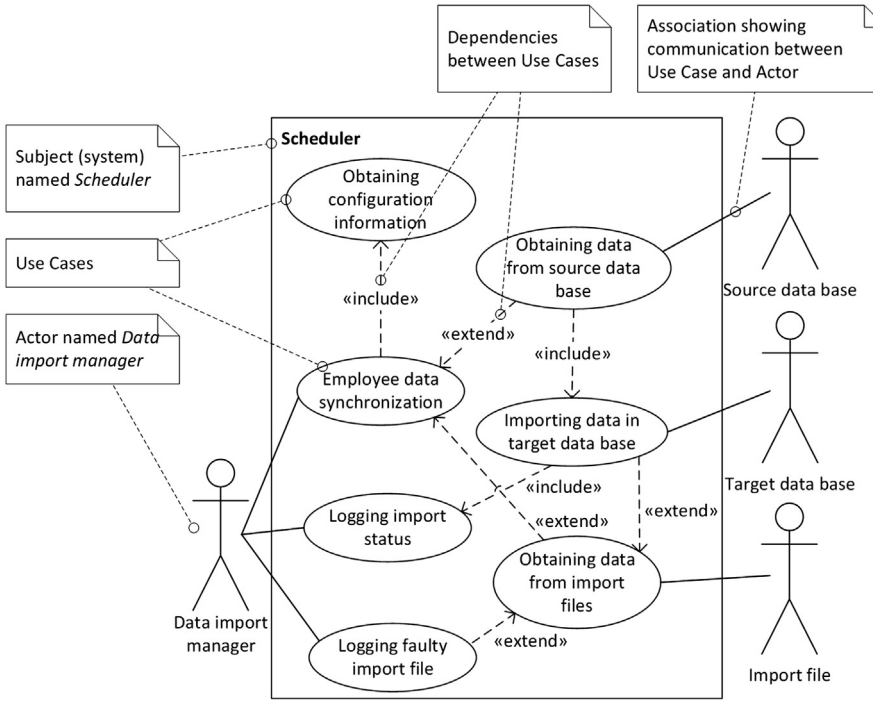

Figure 1.21 Use case diagram of enterprise data synchronization system.

and values that are produced or consumed by actions. The activity diagram is included in UML specification since the first (1.1) version, it includes following elements:

- *Activity*—specifies the flow of subordinate activities and actions, using a control and data flow model. The activity execution is started because of events happening outside that activity, e.g., other activities finish executing, objects and data becoming available. Each activity can include a set of preconditions, postconditions, and input and output parameters. Each precondition should be met for activity to start its execution (thus the availability of objects and data plays an important role). Activities may form invocation hierarchies invoking other activities, ultimately resolving to individual actions. In an object-oriented model, activities are usually invoked indirectly as methods bound to operations that are directly invoked. Activities may describe procedural computation. In this context, they are the methods corresponding to operations on classes.

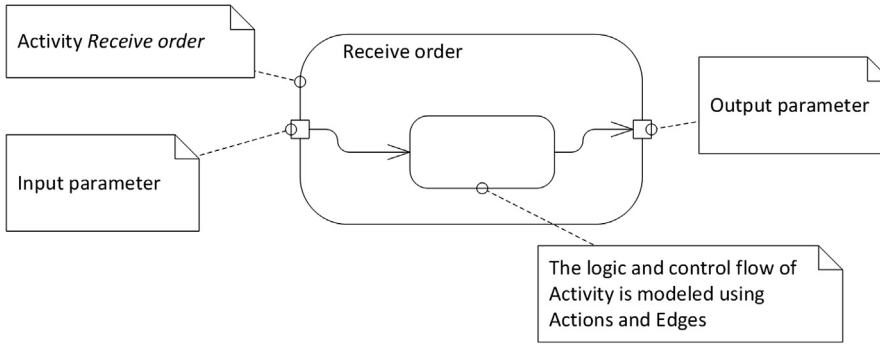

Figure 1.22 Activity.

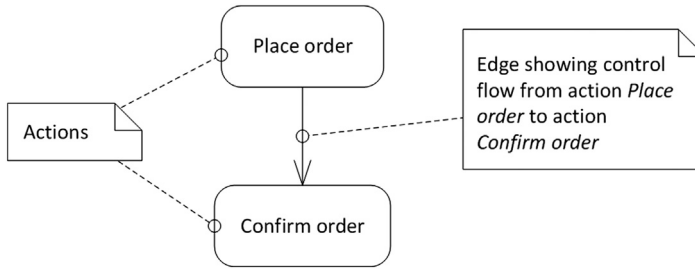

Figure 1.23 Actions and edges.

Activities may be applied to organizational modeling for business process engineering and workflow. In this context, events often originate from inside the system, such as the finishing of a task, but also from outside the system, such as a customer call. Activities can also be used for information system modeling to specify system level processes. An example of general activity is given in Fig. 1.22.

- *Action*—represents a single atomic step within an activity, i.e., it is the smallest step within activity and it is not further decomposed. The dynamics of activity is modeled by all the actions included within it. If there are common actions required in multiple activities, a call behavior action can be used to reference another activity. In this case, the execution of the call action involves the execution of the referenced activity and its actions. While an activity defines a behavior that can be reused in many places, whereas an action is only used once at a particular point in an activity. Example of actions is shown in Fig. 1.23.

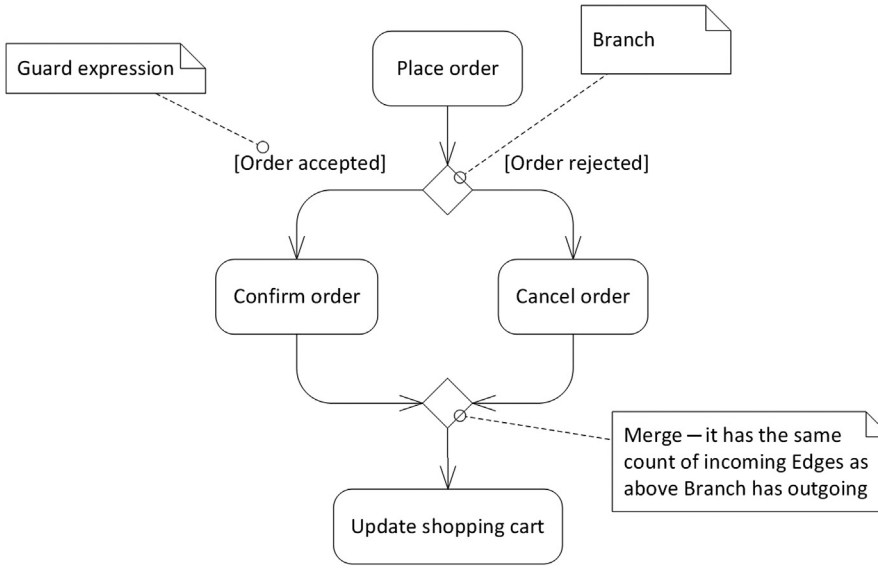

Figure 1.24 Branching and merging in activity diagrams.

- *Edge*—is used to model the control flow from activity to activity, i.e., it is a link between actions having arrowhead at the end of it pointing to the next action which is to be executed. In fact, an action may have sets of incoming and outgoing activity edges that specify control flow and data flow. An action will not begin execution until all of its input conditions are satisfied. A sample of edge connecting two actions is given in Fig. 1.23.
- *Branching and merging*—while actions and edges are used to model activity's behavior, the branching allows to introduce alternate execution paths based on conditions of each branch. A branch is represented in a form of a diamond, it has one incoming edge and two or more outgoing edges. Each outgoing edge has guard (a Boolean expression) to model which action will be executed next. Since activity diagrams beginning with UML version 2 is based on the formalism of Petri nets [21], all the branched flows should be merged together. Merge is represented with the same diamond as branch but it has two or more incoming edges and at least one outgoing edge. An example of branching and merging is illustrated in Fig. 1.24.
- *Forking and joining*—in the case of modeling concurrent control flows fork and join should be used. A fork has one incoming and

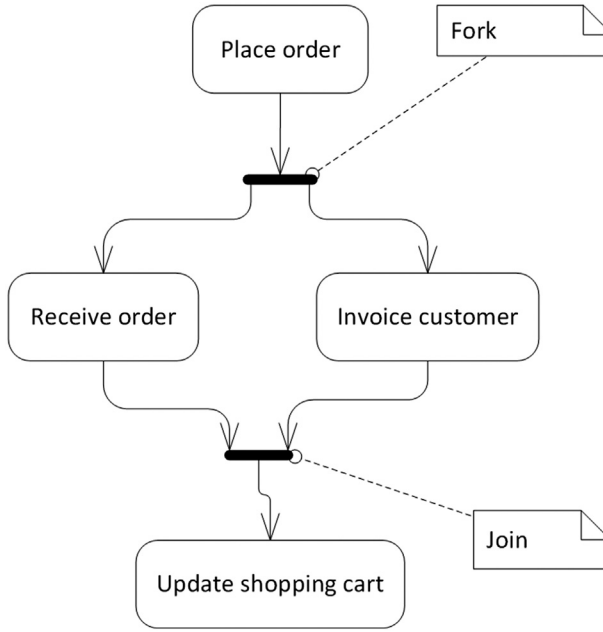

Figure 1.25 Forking and joining in activity diagrams.

two or more outgoing edges. Since activity diagrams is based on the formalism of Petri nets, all the forked flows should be joined together. The execution of an activity after join continues only when all the flows after fork have come to the join. A join has two or more incoming edges and one outgoing edge. Fork and join is represented using thick bar. An example illustrating use of fork and join is given in Fig. 1.25.

- *Initial and final nodes*—an initial node shows a starting point for executing an activity. One activity can have multiple initial nodes. In such case the invoking of the activity starts multiple flows—one at each initial node. Note that flows can also start at other nodes, e.g., the parameter node of activity (see example in Fig. 1.22). An initial node is represented in the form of circle. A final node is an abstract control node at which a flow in an activity stops—when a final node is reached the execution of activity is terminated. The execution termination occurs also in the case of forking—if one of the flow reaches final node, all the concurrent flows are terminated. The final node is represented with a filled circle within unfilled circle. If you need to terminate only one concurrent flow, a

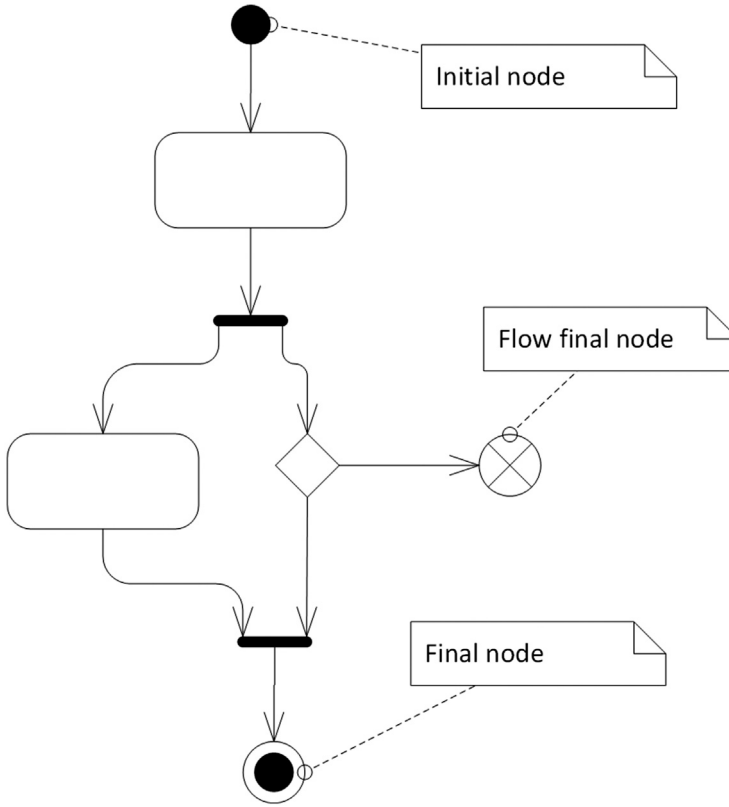

Figure 1.26 Initial and final nodes in activity diagram.

flow final node should be used. The flow final node is represented using X within a circle. See an example of initial and final nodes in Fig. 1.26.

Fig. 1.27 shows an example of activity diagram which is developed as a part of data synchronization system development project.

1.2.2.3 State Diagram

State diagram essentially is a state machine, consisting of states, transitions, events, and activities. While activity diagram shows a flow of control from activity to activity across number of objects involved in execution of those activities, state diagram shows flow of control from state to state within single object. State diagram specifies the sequences of states an object goes through during its lifetime in response to events, together with its responses to those events. The state diagram is

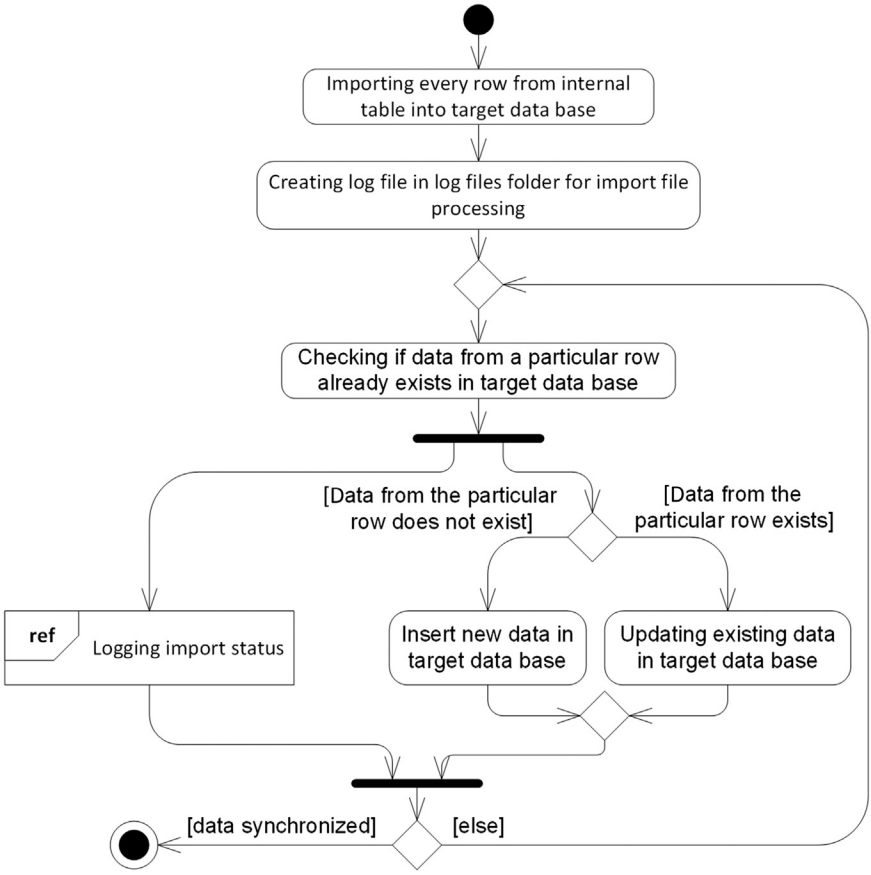

Figure 1.27 Activity diagram modeling data import in target data base.

included in UML specification since the first (1.1) version, it includes following elements:

- *State*—shows a state of an object. For an object to be in a particular state it should met some condition or situation in its lifetime, perform some activity, or wait for some event. If we take a look at file writing software, it could have a set of following states—*idle* (the software waits for a new file to be written in hard drive), *writing* (its writing bytes of file to hard drive), and *waiting* (the file is locked for writing by another process). Launching of such file writing software puts it in the situation where it is waiting for next operation thus setting the state to *Idle*. When user wants to write file, an event of this will be fired and the status is changed to *Writing*. If the software comes to condition where the file is exclusively locked for writing, it

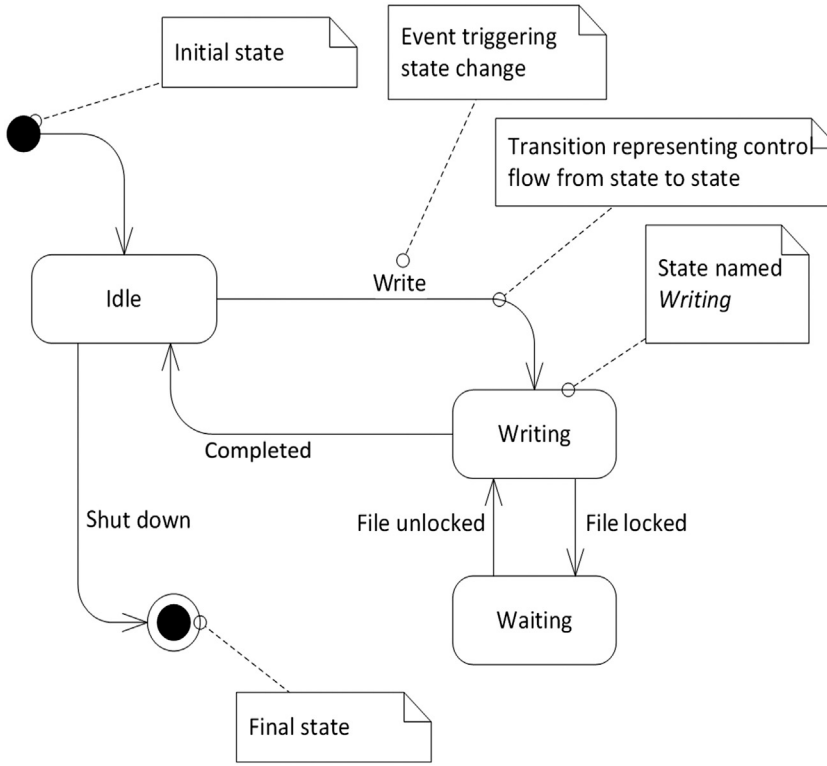

Figure 1.28 States of a file writing software.

is going to the state *Waiting*. After the file is unlocked the unlock event is fired and the software returns to *Writing*. See example in Fig. 1.28.

- *Event*—a specific occurrence or happening that plays a significant role in an object's lifetime and thus can trigger the state transition. See example in Fig. 1.28.
- *Transition*—a directed relationship between a source state and a target state showing that an object will transit from one specified state to another if specific conditions are met and specific events occur. See example in Fig. 1.28.
- *Initial and final states*—a special kind of states showing the initial state of an object and a final state. When the final state is reached, it means that the state machine is completed. These two states are similar to initial and final nodes in activity diagrams (see Section 1.2.2.2).

Fig. 1.28 shows an example of a state diagram showing states of file writing software.

1.2.2.4 Sequence Diagram

A sequence diagram is an interaction diagram that emphasizes the time ordering of messages sent between objects. It shows a set of objects or roles and messages sent and received by them. Sequence diagram has two features that distinguish them from communication diagrams—presence of lifeline and focus of control. The sequence diagram is included in UML specification since the first (1.1) version, it includes following elements:

- *Object or role*—shows object or role which is involved in the communication with other objects or roles.
- *Lifeline*—a vertical dashed bar showing the lifeline of object. The time dimension visually is going from top to down thus we can track the creation and destruction of an object along with the messages sent and received by it.
- *Message*—specifies a particular communication between objects or roles. It is represented as a directed relationship pointing from sender to receiver. Message can be an invocation of an operation, raising a signal, creating or destroying object. The message has a name and it can include also parameters. Using different notations of messages, we can model both synchronous and asynchronous interactions.
- *Control*—shows a period of time during which an object is performing an action requested by the message received, i.e., we can visually show the period of the execution of specific procedure; if we have nested procedure calls then we can visually as soon as possible show the possible bottlenecks raising performance issues in the future. Visually it is represented as a tiny vertical rectangle on the corresponding object's lifeline.
- *Structured control*—while control and messages allows us to model simple communication between objects, in many situations we need to model decision taking, parallel execution, and optional execution. To accomplish this modeling task, there are special graphical notation elements allowing us to model such cases.

Fig. 1.29 shows an example of diagram which is developed as a part of data synchronization system development project.

1.2.2.5 Communication Diagram

Communication diagram like sequence diagram pays attention on objects or roles involved in system and communication between them.

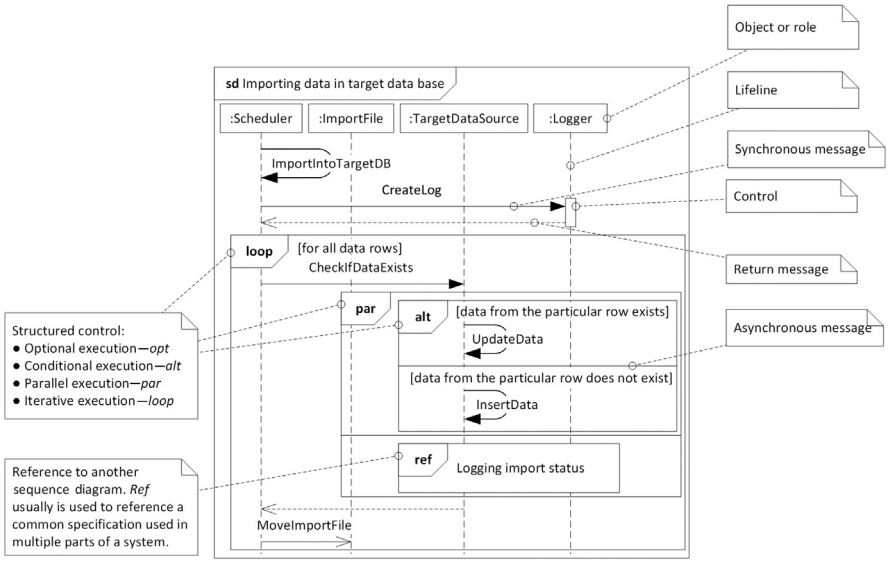

Figure 1.29 Importing data in target data base.

It accents structural organization of objects that send and receive messages; it shows a set of roles, connectors among roles, and messages sent and received by the instances playing these roles. Communication diagram have two features that distinguish them from sequence diagram—path and sequence number of messages (Sequence number indicates the time order of message). A communication diagram is a simplified version of the UML version 1.x collaboration diagram and it is included in UML beginning with version 2.0. It includes following elements:

- *Object or role*—shows object or role which is involved in the communication with other objects or roles.
- *Message*—specifies a particular communication between objects or roles. Since there are no timelines in communication diagram, the sequence of messages is numbered. Having a large communication diagram will lead to quite complex numbering of the messages.
- *Link*—shows a communication link between objects or roles. Communication direction between objects are represented with additional small arrow next to the link pointing from sender to receiver.

Fig. 1.30 shows an example of diagram which is developed as a part of data synchronization system development project.

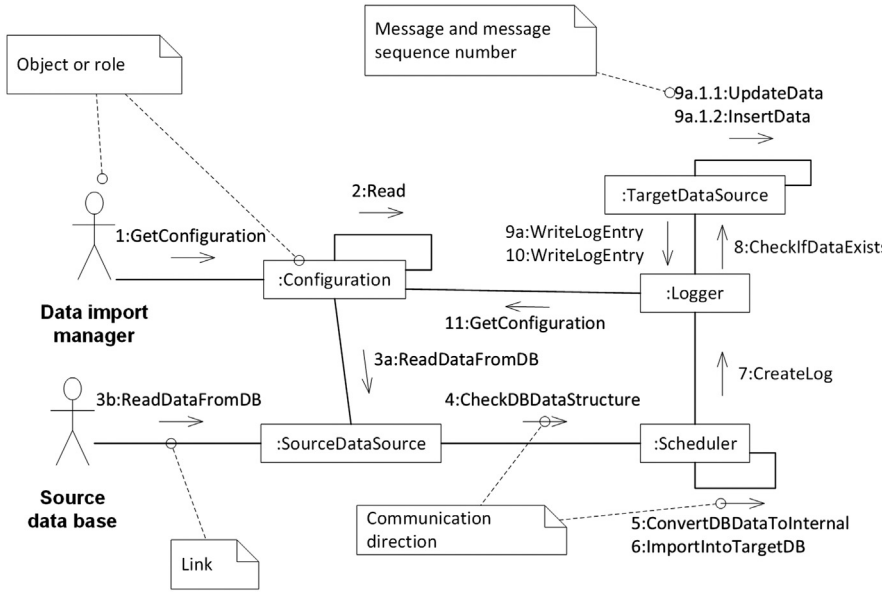

Figure 1.30 Communication diagram representing data synchronization with source data base.

1.2.2.6 Interaction Overview Diagram

Interaction overview diagram define interactions through a variant of activity diagram in a way that promotes overview of the control flow. The lifelines and the messages can be hidden at this overview level. Interaction overview diagrams are specialization of activity diagrams that represent interactions. Interaction overview diagram is introduced starting from UML version 2.0, it includes following elements:

- *Interaction*—emphasizes the time ordering of messages sent between objects. It shows a set of objects or roles and messages sent and received by them. Interaction is represented in a form of a rectangle with name tag, while the contents of this rectangle we can consider as a sequence diagram. Fig. 1.31 shows an example of interaction.
- *Interaction use*—a reference to an interaction. Interaction use hides the contents of the interaction. It is useful if we are using the same interaction across the system. Using interaction use we can define one interaction only once and reference it everywhere where it is needed. Representation of interaction use is given in Fig. 1.32.
- *Edge*—is used to model the control flow from activity to activity, i.e., it is a link between nodes having arrowhead at the end of it pointing to the next node which is to be executed. In fact, a node

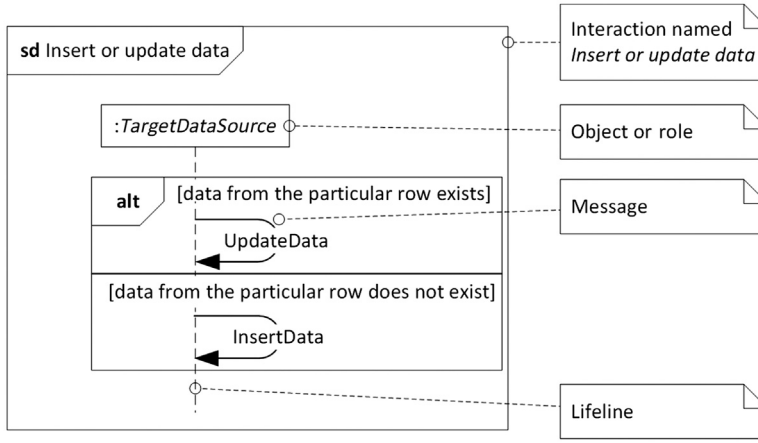

Figure 1.31 Interaction.

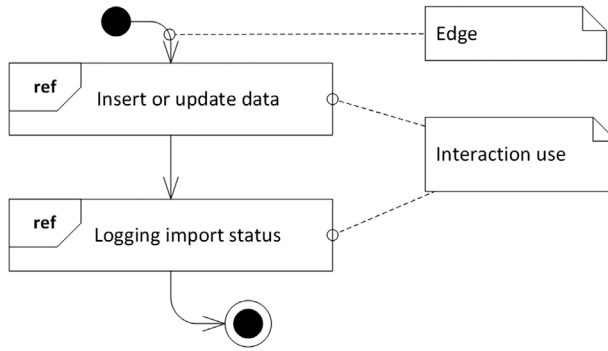

Figure 1.32 Interaction use.

may have sets of incoming and outgoing activity edges that specify control flow and data flow. A node will not begin execution until all of its input conditions are satisfied. An example of edges connecting interaction uses is given in Fig. 1.32.

- *Structured control*—while control and messages allows us to model simple communication between objects, in many situations we need to model decision taking, parallel execution, and optional execution. To accomplish this modeling task, there are special graphical notation elements allowing us to model such cases.
- *Initial and final nodes*—an initial node shows a starting point for executing interaction. An initial node is represented in the form of

circle. A final node is an abstract control node at which a flow in an interaction stops—when a final node is reached the execution of interaction is terminated. the execution termination occurs also in the case of forking—if one of the flow reaches final node, all the concurrent flows are terminated. The final node is represented with a filled circle within unfilled circle.

- *Branching and merging*—while interactions, interaction uses, and edges are used to model simple interactions, the branching allows to introduce alternate execution paths based on conditions of each branch. A branch is represented in a form of a diamond, it has one incoming edge and two or more outgoing edges. Each outgoing edge has guard (a Boolean expression) to model which action will be executed next. Merge is represented with the same diamond as branch but it has two or more incoming edges and at least one outgoing edge. Take a look at Fig. 1.33 which contains an example of branching and merging in interaction overview diagram.

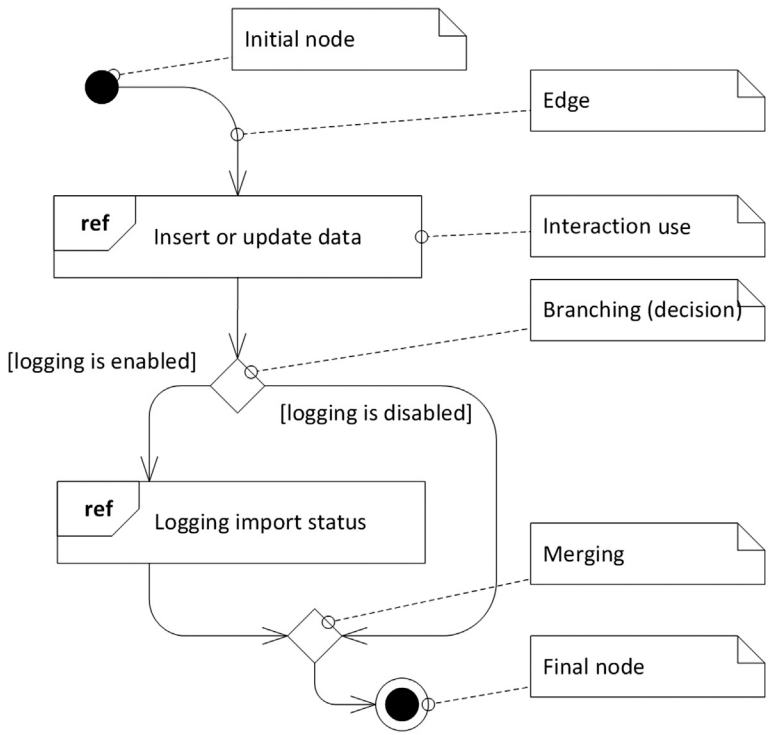

Figure 1.33 Branching and merging in interaction overview diagram.

- *Forking and joining*—in the case of modeling concurrent control flows fork and join should be used. A fork has one incoming and two or more outgoing edges. The execution after join continues only when all the flows after fork have come to the join. A join has two or more incoming edges and one outgoing edge. Fork and join is represented using thick bar. Fig. 1.34 includes an example of forking and joining.

Fig. 1.34 shows an example of interaction overview diagram.

1.2.2.7 Timing Diagram

Timing diagram is used to show interactions when a primary purpose of the diagram is to reason about time; it focuses on conditions changing within and among lifelines along a linear time axis. Timing diagram is a special form of a sequence diagram. The most notable graphical difference between timing diagram and sequence diagram is that time dimension in timing diagram is horizontal and the time is increasing from left to the right and the lifelines are shown in separate compartments arranged vertically. The timing diagram is available since UML version 2.0 and includes elements such as message, lifeline, timeline, and object or role.

1.3 BENEFITS OF APPLYING UNIFIED MODELING LANGUAGE

The use of UML for systems' modeling has following benefits [3,22,35,78,82]:

- The *UML is a modeling language and not a method*, *methodology, or technique,* thus making it independent of particular methods and programming languages. The UML specification defines a number of diagrams and the meaning of those diagrams. A method goes further and describes the steps required to develop the software, which diagrams are developed in what order, and who is responsible for completing certain tasks.
- The *UML is platform independent modeling language*—it can be used to design software for implementation in any programming language.
- It is a modeling language *created from* a set of *widely accepted object-oriented software design methods*, thus ending the endless choose between concurrent notations.
- The *UML is a set of standardized object-oriented models*, thus making communication between stakeholders more efficient and meaningful, i.e., if stakeholders are familiar with UML then the created

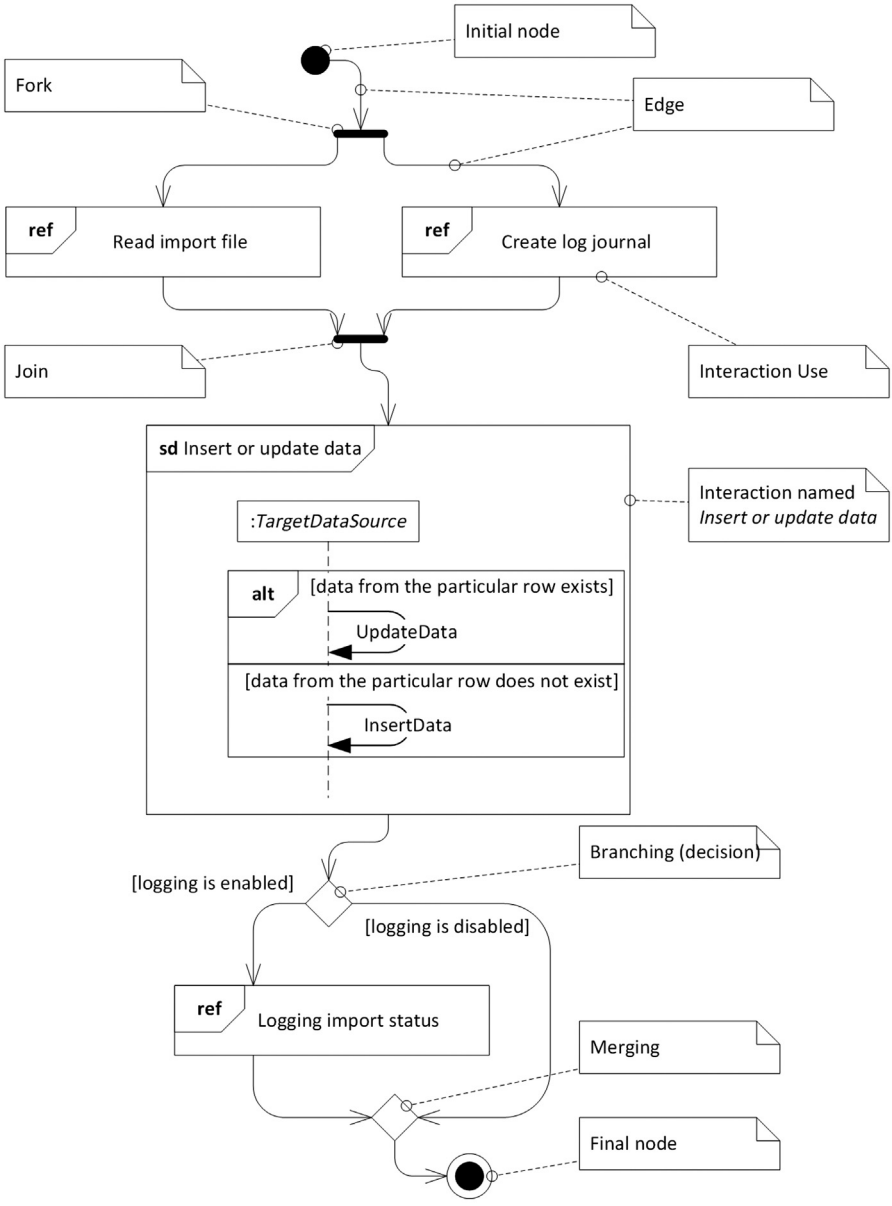

Figure 1.34 Interaction overview diagram.

models of system can be more easily communicated between different development teams, customers, and stakeholders.

- Starting with UML version 2.0 it contains *extension mechanisms*. If the set of models provided by UML are not enough for required solution, it is possible to extend UML in a number of allowed ways.

- The UML can be used for both large and complex systems modeling, as well as for small projects.
- As *UML is defined in accordance with XML Metadata Interchange (XMI)*, the models can be transferred between different tools from different tool vendors. Thus making users of UML less dependent on particular modeling tools.

Despite all above-mentioned benefits that the application of UML within software development has, it has also a number of disadvantages which are discussed in the next subsection.

1.4 DISADVANTAGES OF APPLYING UNIFIED MODELING LANGUAGE

The specification of UML and the UML itself is not developed basing on any theoretical principles regarding the constructs required for an effective and usable modeling language for analysis and design. UML arose from best practices in parts of the software engineering community; in fact, these best practices at some points are even conflicting [22]. Basically, this means that the UML goes without mathematics [83] (except activity diagrams, which are now (starting from UML version 2.0) based on the formalism and mathematics of Petri nets [78]). UML specification is described using the combination of languages—metamodeling, Object Constraint Language (OCL) [124], and the natural language. This has resulted in a language that contains many modeling constructs, which has thus been criticized on the grounds that it is excessively complex and large. At the same time, the UML has also been criticized for lacking the flexibility to handle certain modeling requirements in specific domains. As a result of this criticism, UML has evolved—starting from UML version 2.0 it allows the development of profiles [22].

Main disadvantages of UML are as follows [22,24,51,99]:

- *Size*—UML is a collection of notations that encompasses a wide range of notations. In addition, the provided extension mechanisms of UML allow modelers to add their own, often ad-hoc, extensions to the language. In short, *UML is large and growing*.
- *Incoherence*—UML has brought together a number of notations from different fields. For example, it is not clear how state diagram relates to class diagram and sequence diagram.

- *Different interpretations*—since the semantics of UML constructs are defined by using natural language, they are interpreted differently by different modelers.
- *Frequent subsetting*—organizations tend to define their own UML subset—guidelines on which parts to use; which not to use; own definitions of semantics where the standard is unclear, inconsistent or untenable for the organization concerned.
- *Lack of causality*—despite the fact that UML contains a set of 14 diagrams, none of the existing diagram allows to clearly trace cause-and-effect relationships between both problem and solution domains. This can be related to the fact, that only use case diagram deals with the requirements and computation independent viewpoint modeling.

In regards to the above listed disadvantages of UML, in [114] is presented a list of problems associated with using UML in software development; causes of these problems are various: ambiguous semantics, cognitive misdirection during the development process, inadequate capture of properties of system under consideration, lack of appropriate supporting tools and developer inexperience. By analyzing these problems in detail, part of the researchers claim that some of these problems can be addressed by formalizing UML semantics [122], and the most helpful sequencing of modeling techniques [26]. Others claim that a revision of the UML and its supporting tools is required [31]. Furthermore, it is assumed that largest part of these problems can be addressed to the ambiguous transition between analysis and design models [99].

1.5 FORMALISM OF UNIFIED MODELING LANGUAGE

The UML specification is defined by using a metamodeling approach which adapts formal specification techniques. A metamodel is used to specify the model that comprises the UML. In spite of using the metamodeling approach, the UML specification method lacks some properties of formal specification methods. The specification of UML cannot be considered as formal specification because of natural language (English) use in it. UML specification [77] underlines that the specification as a metamodel does not eliminate the option of specifying it later by using formal/mathematical language (e.g., notation Z [119], Prototype Verification System (PVS) [102], Rigorous Approach to Industrial Software Engineering (RAISE) [69]). Section 1.5.1 takes closer look at the formalism of UML version 1.x specification and

Section 1.5.2 takes closer look at the formalism of UML version 2.x (as the UML version 2.x is major revision of UML version 1.x).

1.5.1 Formalism of Unified Modeling Language Version 1.x

The specification of UML version 1.5 [72] contains language syntax and its static and dynamic semantics. The specification uses a combination of languages—a subset of UML, an OCL, and precise natural language to describe the abstract syntax and semantics of the full UML. Thus, the UML version 1.x specification uses a formal technique for preciseness improving but at the same time keeping readability of it. Despite that the language structure is described in precise specification that is necessary for tool interaction, it is needed to note that the existing description is not a completely formal specification (due to the use of natural language). As stated in [73], a common technique for specification of languages is to first define the syntax of the language and then to describe its static and dynamic semantics. The syntax defines what constructs exist in the language and how the constructs are built up in terms of other constructs. Static semantics of a language define how an instance of a construct should be connected to other instances to be meaningful while dynamic semantics define the meaning of a well-formed construct. These semantics are described using natural language (English).

Summarizing up the UML version 1.x specification, the metamodel of UML is described in a semi-formal way using three views [73]:

1. *Abstract syntax*—presented in a form of UML class diagram. The UML metamodel is defined with the set of interrelated packages. Abstract syntax shows the metaclasses defining the constructs and their relationships and also presents some of the well-formedness rules (mainly the multiplicity requirements of the relationships), and whether or not the instances of a particular subconstruct must be ordered. A short informal description in natural language describing each construct is supplied. The first paragraph of each of these descriptions is a general presentation of the construct that sets the context, while the following paragraphs give the informal definition of the metaclass specifying the construct in UML. Each metaclass has its attributes enumerated together with a short explanation. Besides that, the opposite role names of associations connected to the metaclass is also listed in the same way.

2. *Well-formedness rules*—the static semantics of UML metaclasses (except for multiplicity and ordering constraints) are defined as a set of invariants of an instance of the metaclass. These invariants have to be satisfied for the construct to be meaningful. The rules thus specify constraints over attributes and associations defined in the metamodel. Each invariant is defined by an expression written using OCL together with an informal explanation in English of the expression.
3. *Semantics*—defines meanings of the constructs using natural language (English). The constructs are grouped into logical chunks that are defined together. Since only concrete metaclasses have a true meaning in the language, only these are described in this section.

In summary, the UML metamodel is described in a combination of graphic notation, (precise) natural language, and formal language. The use of natural language for specifying language constructs makes its specification semiformal. This semiformal specification of UML can cause incorrectness and inaccuracy of system models defined with UML (due that statements in natural language can be interpreted with different meanings among different persons (in spite of trying to use it as precise as possible)).

1.5.2 Formalism of Unified Modeling Language Version 2.x

The main goal of major revision of UML within version 2.0 is to increase the precision and correctness of the specification. The set of UML modeling concepts is partitioned into horizontal layers of increasing capability called *compliance levels*. For ease of model interchange, there are only two compliance levels defined for infrastructure specification [77]:

- *Level 0 (L0)*—contains a single language unit that provides capabilities for modeling class-based structures encountered in object-oriented programming languages, and it provides an entry-level modeling capability, and
- *Metamodel Constructs (LM)*—adds an extra language unit for more advanced class-based structures used for building metamodels.

Superstructure specification adds three more compliance levels [78]:

- *Level 1 (L1)*—adds new language units and extends the capabilities provided by Level 0. Specifically, it adds language units for use cases, interactions, structures, actions, and activities.

- *Level 2 (L2)*—extends the language units already provided in Level 1 and adds language units for deployment, state machine modeling, and profiles.
- *Level 3 (L3)—represents the complete UML.* It extends the language units provided by Level 2 and adds new language units for modeling information flows, templates, and model packaging.

All compliance levels are defined as extensions to a single core *UML* package that defines the common namespace shared by all the compliance levels. Level 0 is defined by the top-level metamodel. The UML version 2.x specification is defined by using a metamodeling approach that adapts formal specification techniques. According to [77], the following are the goals of the specification techniques used to define UML 2.x:

- *Correctness*—improves the correctness of the metamodel by helping to validate it.
- *Precision*—increases the precision of both the syntax and semantics. The precision should be sufficient so that there is neither syntactic nor semantic ambiguity for either implementers or users.
- *Conciseness*—the specification techniques should be parsimonious, so that the precise syntax and semantics are defined without superfluous detail.
- *Consistency*—the metamodeling approach should be complemented by adding essential detail in a consistent way.
- *Understandability*—while increasing the precision and conciseness, the readability of the specification should also be improved. For this reason, a less than strict formalism is applied, since a strict formalism would require formal techniques.

The specification technique used in UML version 2.x describes the metamodel in the same way as the version 1.x does, i.e., it uses metamodeling approach and three views (abstract syntax, well-formedness rules, and semantics). Main language constructs are related to metaclasses in the metamodel. Other constructs, i.e., being variants of other ones, are defined as stereotypes of metaclasses in the metamodel. This mechanism allows the semantics of the variant constructs to be significantly different from the base metaclass. Another way of defining variants is the use of metaatributes.

The UML 2.x metamodel contains infrastructure library package which defines a reusable metalanguage kernel and a metamodel

extension mechanism for UML. The metalanguage kernel can be used to specify a variety of metamodels, including UML itself, MOF, and CWM. In addition, the infrastructure library defines a profile extension mechanism that can be used to customize UML for different platforms and domains without supporting a complete metamodeling capability. The UML profile extension mechanism reduces notation size and efforts for specific task solution and allows creating additional constructs along with the benefit of profile reuse in ordinary UML modeling tools. The architectural alignment among UML, MOF, and XMI tries to solve the problem of UML model interchange between tools by using the rules of XMI specification.

In spite of trying to use natural language in more precise way, the specification of UML cannot be considered as formal specification because of natural language use—the problem considered in previous subsection still exists. The UML specification still underlines that the specification as a metamodel does not eliminate the option of specifying it later by using formal or mathematical language. However, the first steps of formalizing UML constructs are taken—starting from UML version 2.0 the activity diagram is formally based on Petri nets [79].

1.5.3 The Need of Additional Unified Modeling Language Formalization

Since the release of the first UML specification researchers are working and proposing approaches to improve formalization of UML. Researches on UML formalization are performed because the meaning of the language, which is mainly described in English, is too informal and unstructured to provide a foundation for developing formal analysis and development techniques, and because of the scope of the model, which is both complex and large [97]. Despite the fact that the latest UML specification is based on the metamodeling approach, the UML metamodel gives information about abstract syntax of UML but does not deal with semantics in formal way (as discussed previous, the semantics is expressed using natural language). Thus, it is hard to determine how a given change in a model influences its meaning and to verify whether a given model transformation preserves the semantics of the model or not. Since UML is method-independent, its specification tends to set a range of potential interpretations rather than providing an exact meaning [31,122].

According to [31], the formalization of UML specification has following benefits:

- *Clarity*—the formally stated semantics can act as a point of reference to resolve disagreements over intended interpretation and to clear up confusion over the precise meaning of a construct.
- *Equivalence and consistency*—a precise semantics provides an unambiguous basis from which to compare and contrast the UML with other techniques and notations, and for ensuring consistency between its different components.
- *Extendibility*—the soundness of extensions to the UML can be verified (as encouraged by the UML authors).
- *Refinement*—the correctness of design steps in the UML can be verified and precisely documented. In particular, a properly developed semantics supports the development of design transformations, in which a more abstract model is diagrammatically transformed into an implementation model.
- *Proof*—justified proofs and rigorous analysis of important properties of a system described in the UML require precise semantics. Proof and rigorous analysis are not currently supported by UML.
- *Tools*—the tools that make use of semantics, e.g., a code generator or consistency checker, require that semantics to be precise, whether it be expressed as part of the standard or invented in the code by the tool developer.

The current UML semantics are not sufficiently formal to realize all of the above listed benefits. Despite that researches on UML formalization have been made before the release of UML version 2.0, the UML version 2.x specification is not written as a formal specification of language. Therefore, there are a number of ongoing UML formalization researches trying to formalize it from different aspects.

1.5.4 Current Unified Modeling Language Formalization Attempts

After OMG accepted UML version 1.1 as a standard, a precise UML (pUML) group was found with main goal to bring together international researchers and practitioners who share the aim of developing the UML as a precise modeling language [31]. The aim of pUML group was to work firmly in the context of the existing UML semantics. As a formalization instrument they use several formal notations (e.g., OCL [124] or

the formal language Z [119]). The pUML group is an example of researches focusing on formalization of UML semantics. Some of the formalization researches are restricted to the semantics of models, while the others are concerned with the issues of reasoning about models and model transformations. Currently there exist a number of approaches for specifying and formalizing semantics of UML:

- Specifying semantics by formal languages (e.g., using language Z [31] or Object-Z [52]),
- Using category theory—captures relationships between specification objects (e.g., [1,20]),
- Using stream theory—as streams is an adequate setting for the formalization of the semantics of concurrent systems (e.g., [16]),
- Using π-calculus or process algebra (e.g., [126]), and
- Using algebraic approaches (e.g., using mathematical notation [122])

As indicated by Evermann in [33], the researches on UML semantics formalization relate to the internal consistency of the UML, not to its relationship to problem domains. To address the relation of UML elements to problem domains, the researchers are ongoing on formalizing the way the software is developed by using UML diagrams (e.g., the problem domain formalization approach [89], or the software development with the emphasis on topology in constructed models [24]) and describing UML constructs by using ontology, thus relating them with problem domains (e.g., [33,125]).

By summarizing up the attempts to formalize UML, the following formalization directions emerge:

- Formalizing the semantics of UML,
- Formalizing the way, the UML is used, and
- Relating UML constructs to problem domains.

1.6 UNIFIED MODELING LANGUAGE IMPROVEMENT OPTIONS

According to the UML version 2.4.1 specification [77,78] and recent researches (e.g., [22,33,88,114,122,125]) in the field of strengthening UML, its use and analysis, the following UML improvement options arise:

- Extending UML by using UML's extensibility mechanisms,
- Formalizing the semantics of UML,

- Formalizing the way, the UML is used, and
- Relating UML constructs to problem domains.

The UML can be strengthened by using the mathematical topology. The use of topology reflects extending the UML to support topology in its diagrams, and formalizing the way the UML is applied during software development process. Since this work is dedicated to extend UML by its extensibility mechanisms and formalizing the software development process, the following two subsections discuss the UML extensibility mechanisms and its improvement by using mathematical topology.

1.6.1 Unified Modeling Language Extensibility Mechanisms

The UML version 2.4.1 extensibility mechanisms permit to extend the language in controlled ways; these mechanisms include stereotypes, tagged values, constraints, and profiles. If the enumerated four extension mechanisms does not solve the problem why the language should be extended, then UML metamodel can be extended using MOF. By extending UML using MOF there are no restrictions on what are allowed to do with a UML metamodel [15,77].

Stereotypes: A stereotype defines how an existing metaclass may be extended, and enables the use of platform or domain specific terminology or notation in place of, or in addition to, the ones used for the extended metaclass. In other words, a stereotype extends the vocabulary of the UML, allowing to create new kinds of building blocks that are derived from existing ones but that are specific to problem under consideration.

Stereotype can be considered as a type that defines other types, because each one creates equivalent of a new class in the UML metamodel. When an element is stereotyped (such as node or a class), the UML gets extended by creating a new building block just like the existing one but with its own special modeling properties (each stereotype may provide its own set of tagged values), semantics (each stereotype may provide its own constraints), and notation (each stereotype may provide its own icon). The stereotype "*stereotype*" specifies that the classifier is a stereotype that may be applied to other elements [15,77].

Tagged values: A tagged value extends the properties of a UML stereotype, thus allowing creation of new information in element's specification. By using stereotypes it is possible to add new things to the

UML; by using tagged values it is possible to add new properties to a stereotype. Tags that apply to individual stereotypes are defined so that everything with that stereotype has tagged value. A tagged value is not the same as class attribute. Rather, a tagged value can be considered as metadata because its value applies to the element specification, not to its instances [15].

Constraints: A constraint extends the semantics of a UML construct, thus allowing to add new rules or to modify existing ones. Each constraint consists of a textual description in natural language and may be followed by a formal constraint expressed in OCL. If it is not possible to express the constraint in OCL, then in such case the formal expression can be omitted [15,78].

Profiles: The profile mechanism has been specially defined for providing a lightweight extension mechanism to the UML specification. In UML version 1.1, stereotypes and tagged values were used as string-based extensions that could be attached to UML model elements in a flexible way. In subsequent revisions of UML, the notion of a profile was defined in order to provide more structure and precision to the definition of stereotypes and tagged values. Since the UML version 2.0 specification this has been carried further, by defining UML extension as a specific metamodeling technique. Stereotypes are specific metaclasses, tagged values are standard metaattributes, and profiles are specific kinds of packages. A profile defines a specialized version of UML for particular area or solution. Because it is built on standard UML elements, it does not present a new language, and it can be supported by ordinary UML tools [15,77,78].

According to UML version 2.4.1 specification [78], the profiles mechanism is not a first-class extension mechanism (i.e., it does not allow for modifying existing metamodels). Rather, the intention of profiles is to give a straightforward mechanism for adapting an existing metamodel with constructs that are specific to a particular domain, platform, or method. Each such adaptation is grouped in a profile. It is not possible to take away any of the constraints that apply to a metamodel such as UML using a profile, but it is possible to add new constraints that are specific to the profile.

The UML metamodel extension: First-class extensibility is handled by using MOF, where there are no restrictions on what are allowed to

do with a UML metamodel (i.e., it is possible to add and remove metaclasses, constraints, and relationships as necessary) [77].

"*There is no simple answer for when you should create a new metamodel and when you instead should create a new profile*" [78].

1.6.2 Improving Unified Modeling Language by Using Topology

The UML improvement by using mathematical topology is based on topology and formalism of Topological Functioning Model (TFM) [86]. The TFM is a mathematical modeling language intended to design and analyze functionality of a system and it holistically represents a complete functionality of the system from a computation independent viewpoint. It considers problem domain information separate from the solution domain information. TFM has strong mathematical basis and is represented in a form of a topological space. Graphically, it is drawn as an oriented graph where nodes represent functional features of the system, while directed arcs represent their causal relationships. The TFM has topological characteristics: connectedness, closure, neighbor-hood, and continuous mapping. Despite that any graph is included into algebraic topology, not every graph is a TFM. A directed graph becomes the TFM only when theoretical substantiation of the systems is added to the above mathematical substantiation. The latter is represented by functional characteristics: cause-effect relations, cycle structure, and inputs and outputs [88,91].

It is acknowledged that every business and technical system is a subsystem of the environment. TFM enables careful analysis of system's operation and communication with the environment through analysis of functional cycles—a common thing for all system (technical, business, or biological) functioning should be the main feedback, visualization of which is an oriented cycle. Thus, it is stated that at least one directed closed loop (i.e., cycle) must be present in every topological model of system functioning. This cycle shows the main functionality that has a vital importance in the system's life. Usually it is even an expanded hierarchy of cycles. By interrupting this main cycle the system can no longer function or it functions faulty [86]. Therefore, a proper cycle analysis is necessary in the TFM construction, because it enables careful analysis of system's operation and communication with the environment. To better illustrate main cycle in

graph representation of TFM, the arcs belonging to this cycle is drew with bolder lines [84].

The TFM and its construction steps are given in Chapter 4, Topological Unified Modeling Language, of this book.

The UML can be improved by supplementing it with the topological and functioning characteristics of TFM. To allow using topology in UML diagrams, it needs to be extended by using extensibility mechanisms. In such case a new kind of UML is created—*Topological Unified Modeling Language* (*Topological UML*). The idea of Topological UML is adapted from [83]. The core framework proposal for Topological UML profile is presented in [99]. The first research results in [96] shows that *the transfer of topological and functioning characteristics from TFM to UML is sufficient for clearly tracing cause-and-effect relationships in both—problem and solution—domains.*

1.7 SUMMARY

The UML is a visual language for specifying, constructing, and documenting the artifacts of systems. It is a general-purpose modeling language that can be used with all major object and component methods, and that can be applied to all application domains and implementation platforms.

The UML version 1.x (the first version (1.1) is released in 1997) contains nine diagram types. UML version 2.0 (released in 2005) is a major rewrite of UML version 1.x with the main goal to increase the precision and correctness of the specification. The version 2.0 contains 13 diagram types, and the version 2.2 adds additional one diagram type—profile diagram (now in total UML has 14 diagram types). At the moment of writing this work, the newest version is 2.5 and it is released in 2015.

The specification of UML version 2.0 to 2.4.1 is divided into two volumes: infrastructure (core metamodel) and superstructure (notation and semantics for diagrams and their model elements). Actually, the superstructure specification is based on infrastructure specification. Specification version 2.5 is a rewritten version 2.4.1 combining together in a single document the infrastructure and the superstructure. The set of modeling concepts of UML is partitioned into horizontal

layers of increasing capability called compliance levels. UML infrastructure specification defines only two compliance levels (for ease of model interchange): L0 and LM, while the superstructure specification adds three more compliance levels: L1, L2, and L3. In fact, the complete UML specification is given in compliance level L3.

While the application of UML within software development has a number of benefits, it also has some disadvantages. The main benefits are: UML is independent of software development methods, techniques, and platforms; it has an extension mechanism thus allowing to solve specific modeling tasks; and the models can be transferred between different tools from different tool vendors since UML is defined in accordance with XMI. The main disadvantages of UML application are its size, incoherence, different interpretations, frequent subsetting, and the lack of causality. From these disadvantages rises a set of problems like ambiguous semantics, cognitive misdirection during the development process, inadequate capture of properties of system under consideration, lack of appropriate supporting tools and developer inexperience, and inability to trace cause-and-effect relationships between the existing artifacts in problem domain and created artifacts in solution domain. By taking a closer look at benefits and disadvantages, it is visible that some benefits turn into disadvantages (e.g., independency of software development methods leads to cognitive misdirection during the development process). To address the listed disadvantages, a bunch of researches on UML strengthening and formalization are performed and are still ongoing, e.g., formalizing the semantics of UML, formalizing the way the UML is used, and relating UML constructs to problem domains.

The UML can be strengthened by using mathematical topology, thus addressing the disadvantage of lacking causality. Next chapter is dedicated to explore currently existing UML modeling driven software development approaches, thus addressing the disadvantages of UML's size, incoherence, different interpretations, and frequent subsetting.