CHAPTER 5

Process Modeling Using UML

GREGOR ENGELS, ALEXANDER FÖRSTER, REIKO HECKEL, and SEBASTIAN THÖNE

5.1 INTRODUCTION

The Unified Modeling Language (UML)\(^1\) is a visual, object-oriented, and multipurpose modeling language. Primarily designed for modeling software systems, it can also be used for business process modeling.

Since the early 1970s, a large variety of languages for data and software modeling like entity-relationship diagrams [2], message sequence charts [5, 10], state charts [9], and so on, have been developed, each of them focusing on a different aspect of software structure or behavior. In the early 1990s, object-oriented design approaches gained increasing attention, for instance, in the work of James Rumbaugh (Object Modeling Technique or OMT [21]), Grady Booch [1], and Ivar Jacobson [12].

The UML emerged from the intention of Rumbaugh, Booch, and Jacobson to find a common framework for their approaches and notations. Furthermore, the language was also influenced by other object-oriented approaches like that of Coad and Yourdon [3]. The first version, UML 1.0 [20], was released in 1997 and accepted as a standard by the Object Management Group (OMG)\(^2\) the same year. The OMG, which took over the responsibility for the evolution of the UML from then on, is a consortium from both industry and academia and is also responsible for other well-known initiatives like CORBA, MDA, and XMI. OMG specifications have to undergo a sophisticated adoption process before being agreed upon as a standard by the OMG members. Since many important tool builders and influential software companies are involved in the OMG, UML has quickly been accepted by the software industry, especially since version UML 1.3 appeared in 1999. When writing this book, the current UML version was UML 2.0 [18], a major revision of the language.

UML is a conglomeration of various diagram types. Therefore, the challenge is to provide a uniform framework for all these heterogeneous diagram types and accounting for relationships between them. In UML, this is solved by a common meta-model that formally defines the abstract syntax of all diagram types. The

---

\(^1\)www.uml.org

\(^2\)www.omg.org
meta-model is defined with the help of the OMG Meta-Object Facility (MOF) [16]. Such a declarative meta-model is an alternative to grammars usually used to define formal languages.

Besides the meta-model and a notation guide defining a concrete syntax for the meta-model elements, the UML specification also informally describes the meaning of the various meta-model elements. In the past, this informal semantics description has raised many issues about how to interpret certain details of the language. Even in the latest revision, UML 2.0, there are still a number of contradictions and ambiguities to be found in the specification. At some points, the UML 2.0 specification is intentionally left incomplete, providing so-called variation points that allow tool builders and modelers to interpret the language according to their specific purposes.

This chapter provides an introduction to UML, focusing especially on those parts relevant for process modeling. It covers five major aspects of process models, namely (1) actions and control flow, (2) data and object flow, (3) organizational structure, (4) interaction-centric views on business processes, and (5) system-specific process models used for process enactment. Although not every detail of the language can be presented, we intend to provide at least the most important concepts required for UML-based process models.

For discussing the various process modeling aspects, we use activity diagrams as fundamental tools for process modeling with UML. Section 5.2 explains the control flow concepts of activity diagrams, and Section 5.3 extends the process models by integrating object flow. Aspect (3), the modeling of underlying organizational structures, is covered by Section 5.4 with the help of class and object diagrams. Section 5.5 then covers aspect (4) and deals with a different view of business processes, focusing more on the interactions among involved business partners. To model such an interaction-centric view, we introduce sequence diagrams. To facilitate process enactment according to aspect (5), system-specific models should describe how to relate existing software components to the desired process activities. Thus, Section 5.6 introduces structure diagrams for describing available software systems and for specifying provided operations, which are then integrated into the considered process models. The chapter is concluded by a summary and exercises of varying degrees of difficulty.

Throughout the chapter, the different diagram types are illustrated by a running example that deals with an e-business company selling hardware products. For simplicity reasons, the company’s product range is limited to monitors and computers only. It processes incoming orders by testing, assembling, and shipping the demanded products.

5.2 Modeling Control Flow with Activity Diagrams

The basic building block of a process description in UML is the activity. An activity is a behavior consisting of a coordinated sequencing of actions. It is represented by an activity diagram. Activity diagrams visualize sequences of actions to be performed, including control flow and data flow. This section deals with the control flow aspect of process models in UML.
5.2.1 Basic Control Flow Constructs

Figure 5.1 shows a first small example of an activity. This activity describes a business process of our exemplary e-business company, which sells computer hardware products. The activity is visualized by a round-edged rectangle. If the activity has a name, it can be displayed in the upper-left corner of the rectangle. The name of the example activity in Figure 5.1 is “Sell computer hardware.” Inside the activity rectangle we find a graphical notation consisting of nodes and edges that represents the activity’s internal behavior. There are two kinds of nodes to model the control flow: action nodes and control nodes.

As a first step in the formulation of a business process, we need to model what tasks the process has to perform while executing. In an activity diagram, this is described by actions. An action stands for the fact that some transformation or processing in the modeled system has to be performed. Activities represent the coordinated execution of actions. Action nodes are notated as round-edged rectangles, much like that of an activity, but smaller. Actions have names that are displayed inside the action symbol, for instance, “check order” or “get products” in our example. Actions can manipulate, test, and transform data or can be calls to another activity. What has to be done when executing an action can be described by the name of the action such as “check order.” Actions can also be specified using programming language expressions such as $c := a + b$ or formal expressions. The execution of actions takes place over a period of time.

Actions need to be coordinated. This coordination of actions within an activity is expressed by control flow edges and control nodes. The most fundamental control structure is the sequence, in which one action can start executing when another action stops executing. A simple example of a sequence of actions can be seen in Figure 5.2. The arrows between the action nodes are called activity edges and specify the control flow.

In UML 2.0, the semantics of activities are defined based on token flow. Tokens can be anonymous and undistinguishable; in that case, they are called control tokens.
Tokens can also reference data objects. These tokens are called object tokens. See Section 5.3 for an introduction to the concept of object flow.

Tokens flow along the control edges, thus determining the dependencies in the execution of the actions. Actions can only begin execution when tokens are available from all preceding actions along the incoming edges (step 1 in Figure 5.3). When the execution of the action starts, all input tokens are consumed and removed from the incoming control flow edges simultaneously (steps 1 and 2 in Figure 5.3). After completion of the action, tokens are offered to all outgoing edges simultaneously (step 3).

In a control flow, actions sometimes have to be executed alternatively depending on conditions. This corresponds to the control structure often called “XOR-split” or “simple choice” (see Chapter 8), which is represented in activity diagrams by decision nodes, merge nodes, and guards. The diamond symbol in Figure 5.4 represents a decision node if one edge enters the node and multiple edges leave it. In the opposite case, if multiple edges enter the diamond symbol and one leaves it, it is a merge node, which corresponds to an “XOR-join.” Diamond symbols with both multiple edges entering and multiple edges leaving them are combined decision and merge nodes.

In order to describe the conditions for the choice of the alternative control flows, the edges leaving a decision node are usually annotated by guards. Guards are logical expressions that can differentiate true from false. They can be formulated using natural language, programming language constructs, or formal expressions such as mathematical logic or OCL. OCL stands for Object Constraint Language [17], which was also developed by the OMG. It is a language for describing constraints whenever expressions over UML models are required. In an activity diagram, guards have to be enclosed in square brackets. An edge can only be traversed if the guard attached to that edge, if any, is true.

If a guard expression becomes very lengthy, one can also attach a decision-input note box to the diamond containing the text of the guard condition.
note box is connected to the decision node with a dashed line, as in Figure 5.1. In the example, a product is either a computer or a monitor. As there exist two different test facilities for monitors and computers, the control flow has to be split into two different alternatives.

A special case of a guard is \[\text{else}\], which is true if and only if all other guards on all other edges leaving the same node are false. The use of guards is not restricted to edges leaving decision nodes. As a general rule, control edges can only be traversed if their guard conditions are true.

In process models, one frequently has to model concurrent control flows. Concurrency in activity diagrams can be expressed by using fork and join nodes. They are equivalent to the concept of “AND-splits” and “AND-joins” described in Chapter 8. A thick-lined bar is a fork node if one edge enters it and multiple edges leave it, as in Figure 5.5. At a fork node, the control token becomes duplicated and the control flow is broken into multiple separate control flows that execute in parallel. In order to simplify the model, one can also draw multiple outgoing edges leaving an action node (implicit fork). In our example in Figure 5.1, the action “save order information in archive” can be executed in parallel with the action “get products” and the product tests, as indicated by the fork node.

A join node is used to combine the concurrent control flows. It is represented by a thick-lined bar with multiple edges entering it and one edge leaving it. It synchronizes the control flows at the incoming edges since the execution is stopped until there are tokens pending along all incoming edges. A thick-lined bar with multiple incoming and outgoing edges is a combined join and fork node, as depicted in Figure 5.5. Actions with multiple incoming edges represent implicit joins as the action
“assemble bundle” in our example in Figure 5.1. Figure 5.6 shows an action with implicit fork and join.

In Figure 5.1, there are two more control nodes. A solid circle indicates an initial node, which is the starting point for an activity. A solid circle surrounded by a hollow circle is the final node, indicating the end of the control flow. It is possible to have more than one final node in one activity. In that case, the first final node reached stops all flows in the activity. A detailed analysis of control structures in workflow models can be found in [13].

5.2.2 Advanced Concepts

Pre- and Postconditions. In process models, it is often required to formulate assertions and conditions that need to hold locally at certain points in the control flow, at the overall beginning of an activity, or at its end.

In order to express global conditions for an activity, the activity can be constrained with pre- and postconditions. Whenever the activity starts, the precondition is validated. Whenever the activity ends, the postcondition has to be fulfilled. Both pre- and postconditions are modeler-defined constraints. They are indicated by the keywords <<precondition>> and <<postcondition>>, typically in the upper part of an activity box, as in Figure 5.7a.

Local pre- and postconditions can be attached to actions. They are displayed as note boxes containing the keywords <<localPrecondition>> or <<localPostcondition>>, as in Figure 5.7b. A token can only traverse an edge when it satisfies the postconditions of the source node, the guard condition for the edge, and the precon-
ditions for the target node all at once. The constraints can be formulated in natural language, programming language expressions, or any formal language like OCL, mathematical logic, and so on.

**Hierarchical Process Composition.** Business processes can easily become very complex. It is advantageous for a process description language to allow hierarchical nesting in order to reduce the complexity. Thus, actions as part of a UML activity can be calls to other activities. The nesting of activities results in a call hierarchy in which activities can be found on different levels of abstraction. An action that calls another activity is symbolized by a hierarchy fork within the action symbol (see action “test computer” in Figure 5.8.)

**Edge Weights.** In business processes, it is sometimes necessary to describe a situation in which a defined number of objects or tokens have to accumulate at a certain point in the process before the execution can continue. In our example, one needs to collect all monitors and computers of an order before they can be bundled for shipment. With activity diagrams, it is possible to describe such situations. Edges can carry multiple tokens at the same time. They can also have weights that are displayed by writing \{weight=n\} next to an edge. The weight expression by which \(n\) is replaced determines the number of tokens that are consumed from the source node on each traversal. The traversal of the edge is delayed until the required number of tokens is offered by the source node.

**Connectors.** If edges cross large parts of a diagram, one can use connectors to split a control flow edge into two parts (see Figure 5.9). Connectors are circles containing a label. The label has to match uniquely with the label of one other connector.

**Process Interaction and Signaling.** If the modeled system contains multiple threads of control or different activities or instances of activities running at the
same time, process interaction may be required to coordinate the execution between these control flows. Process interaction can be facilitated by sending and receiving signals. In activity diagrams, there are two special nodes representing this functionality, as shown in Figure 5.10: send signal action and receive signal action.

If a token reaches the send signal action, it triggers the emission of the signal. Signals can be received by receive signal actions. Corresponding send and receive actions can be determined by the signal name and optionally by a dashed line connecting sender and receiver. As soon as the signal is sent, the control token can pass on.

Receive signal actions may be included in the control flow, that is, they have an incoming control edge. In that case, they become activated as soon as there is a token available along their incoming edge. When the incoming signal is received, the execution can continue and the control token will be passed on. Receive signal actions without incoming edges become activated as soon as the activity starts execution. After that, activities can always receive signals.

** Constructs to Model Exception Handling.** The UML provides constructions for exception handling. A common problem is that in part of a process an exceptional condition can arise that requires actions to be performed apart from the regular workflow. This situation can be reflected in activity diagrams by introducing an interruptible activity region. Such a region contains one or more actions. It is displayed by a round-edged dashed rectangle surrounding the actions that form the interruptible region. A lightning-bolt-shaped edge called the interrupting edge leaves the interruptible region. The semantics of this construction is that if the interrupting edge is traversed, all other actions within the region are canceled and all remaining tokens within the interruptible region become abandoned. Two alternative notation options are available for the interrupting edge, as shown in Figure 5.11.

Another exception handling situation occurs when an exceptional condition arises within one single action. For example, the action could be a mathematical divi-
sion operation, possibly leading to a division by zero. In activity diagrams, an exception handler can be attached to single actions, as in Figure 5.12. In this case, the exception handler is a behavior that is executed whenever a predefined exception occurs while an action is being executed.

Multiple exception handlers can be attached to catch different types of exceptions. The execution of the exception handler substitutes for the execution of the action during the time it is running. After the execution of the exception handler has terminated, the control flow is continued, at the point where the execution was triggered.

The exception handler does not have own incoming and outgoing control edges since it only replaces the execution of the interrupted action. In the cases in which an exception cannot be caught, it becomes propagated to the next-higher nesting or abstraction level; that is, if the action raising the exception is part of an activity A that has been called by an activity B, then the exception is propagated to B.
if it is not caught by A. If no exception handler can be found, the system behavior is undefined.

5.3 MODELING OBJECTS AND OBJECT FLOW

All processes perform operations on physical objects. For example, goods are produced from raw materials or logical objects like information and data. With UML, it is possible to model the types, properties, and states of those objects as well as to integrate corresponding object flows into the activities.

For instance, consider the order handling process of our computer hardware company (see Figure 5.1), which comprises the packing of product bundles for incoming orders. This process involves two basic object types, namely, hardware products and order forms. From this simple scenario, we can derive the following three requirements for modeling objects and object flow:

1. We want to model data structures, objects types (in object-oriented languages called classes), and relationship types in order to classify objects, define common properties, restrict possible relationships, and explain internal structures. For instance, we want to describe that order forms always contain a list of order items and that each item refers to a certain product type. For this purpose, we will introduce UML class diagrams.

2. We want to represent individual objects with their concrete properties and relationships. For instance, we want to describe pending orders and available products at a particular point in time. For this purpose, we will introduce UML object diagrams.

3. We want to define the dependencies between objects and actions occurring in activities, in particular input and output relationships as well as object flow dependencies. For instance, we want to describe that our packaging process requires a new order as input and how this order is processed at the different stages of the process. For this purpose, we will explain object flow concepts as part of UML activity diagrams.

5.3.1 Object Types and Instances

Since UML is an object-oriented language, objects and their types are fundamental concepts of the language. They can be used to represent physical entities like products or persons, information like data or documents, as well as logical concepts like product types or organizations. Object types, also called classes, are defined in UML class diagrams. Objects are instances of these types, and they are represented in UML object diagrams.

Figure 5.13 summarizes the basic constructs that can be used within a class diagram. In principle, each class diagram is a graph with classes as nodes and relationships as edges. A class defines a set of common properties, also called attributes, that all instances of the class assign concrete values to. A property is defined in the
second compartment of a class symbol by a property name and a property type like string, integer, and so on.

Besides the classes as object types, a class diagram can contain three different kinds of relationship types (see Figure 5.13):

- A *generalization relationship* (depicted as a triangle-shaped arrow) is used to factorize common properties of different classes in a common superclass. The subclasses inherit all the properties and associations of their superclasses. If it is not intended or meaningful to create own instances of the superclass; it can be declared to be an *abstract class* (indicated by its name printed in italics).

- An *association* (depicted as a line between classes) is used to define possible links between objects. The usual form are binary associations between exactly two classes. Besides a name, an association has cardinality constraints at its ends, which are given as a fixed value or as a range of lower and upper bounds (the symbol * means “unbounded”). For each association end, the cardinality constraint restricts the number of objects that can be associated to an instance of the opposite association end. A small solid arrowhead next to the association name can be used to indicate a reading direction for ambiguous association names.

- An *aggregation* (depicted as an association with a diamond symbol next to the container class) is a special association indicating a containment relationship. It is used to model object types that have other objects as parts.

Coming back to our example, consider the class diagram in Figure 5.14. It states that every Order is submitted by a Customer and that it is composed of one or more OrderItems. The Producttype class and its subclasses Computertype and Monitortype are used to describe the product range of the company. Every OrderItem refers to a Producttype that the customer wants to order. The Product class and its subclasses Computer and Monitor are used to describe the physical products to be sold. The association isOfType between Product and Producttype is used to assign a type to every product. Both Product and Producttype are abstract classes so that only their subclasses can have instances. Due to the generalization, the subclasses inherit the isOfType association and the name attribute. Products can be aggregated to a Bundle.

Objects, being instances of the defined classes, have unique identifiers and concrete values for their properties. A snapshot of the objects existing at a certain point...
in time is modeled by a UML object diagram, as shown in Figure 5.15 for our application example. In contrast to classes, objects are depicted with underlined identifiers and type names. Objects that are parts of composite objects can be shown within the rectangle of the container object.

5.3.2 Extending Activities with Object Flows

In Section 5.2, we introduced UML activities that focus solely on the control flow aspect. Now, we can combine the control flow with object flow.

Figure 5.14 Class diagram example.

Figure 5.15 Object diagram example.
In UML activities, we use object nodes to model the occurrence of objects at a particular point in the process. If we expect objects of a certain type only, we can typify object nodes by one of the classes defined in the class diagram. Since business processes usually perform transformations on physical objects or data objects, it is often useful to add information about the current state in the object life cycle to an object node. In general, an object node is depicted as a rectangle containing the type name and, in square brackets, the state information, as shown in Figure 5.16a.

In order to also capture object flow, the token flow semantics of activity diagrams is extended by object tokens. An object token behaves like a control token, but, in addition, it carries a reference to a certain object. Edges between object nodes represent flows of such object tokens. If the target object node of such an edge has a type, it can only accept tokens with objects that are instances of this type. Thus, the modeler has to consider type compatibility, and an object flow edge is only allowed if the type of the target object node is the same as or a supertype of the type of the source object node.

Whenever an object token arrives at an object node, it is immediately offered along outgoing edges to downstream nodes. If the node has more than one outgoing edge, they have to compete for the object token and only one of them can retrieve it. If no guard condition is given, the winning edge is determined nondeterministically. Otherwise, if we want to allow all downstream nodes to have concurrent access to the object, we can insert an explicit fork node since this causes a duplication of the object token. Then, each downstream node receives a token referring to the same object.

However, if none of the downstream nodes is ready to accept tokens, the object node can temporarily store the tokens and pass them on in the same order (FIFO or “first in first out”). Instead of FIFO, one can also specify a different kind of queuing order like LIFO (“last in first out”), “by priority,” and so on by a suitable selection note as shown in Figure 5.16b. Moreover, an upper bound can be given that restricts the number of tokens allowed to accumulate in an object node. Object tokens cannot flow into the node if that limit has already been reached.

With the help of object nodes and object flows, we can model how objects are directed through the different actions of an activity and how they are assigned to the input and output parameters of the various actions. To facilitate the latter, object nodes can also appear in the form of input pins and output pins, which are directly attached to an action node. Input pins are assigned to the input parameters of the ac-

![Figure 5.16](c05.qxd_8/7/2005_10:18 AM_Page_97)

Figure 5.16 Object nodes (a and b), connected pins (c), and stand-alone notation (d).
tion, and output pins to its output parameters. As shown in Figure 5.16c, pins are depicted as small hollow squares with their types written above the square.

An action can start execution only if all its input pins hold an object token. Then, the action consumes the tokens from its input pins and, after completion, places new object tokens on all of its output pins. Figure 5.16c shows two actions whose output and input pins are connected by an object flow edge. If the connected output and input parameters have the same name and type, the standalone notation can be used instead of the two pins, as shown in Figure 5.16d.

In the following paragraphs, we show how these object flow concepts apply to our example business process of Figure 5.1, and we explain the different usages of object nodes in more detail. The resulting extended activity model is shown in Figure 5.17.

Similarly to the individual actions, the overall activity can have input and output parameters, too. Those activity parameters are modeled with object nodes playing the role of activity parameter nodes. In our order process, for instance, each arrival of an “Order” object places a corresponding object token at the input parameter node of the activity. From there, the token is directed to the first action, and the process is executed until the last action places a token with the “Bundle” object at the output parameter node.

The first action “check order” of the process validates an incoming “Order” object and, if successful, passes it on through its output pin. Since the downstream actions require “Order” objects in the state “checked,” too, we can use the standalone notation for object nodes here.

If the check is not successful, we want the process to terminate and to reject the invalid order. We can model this as an exception output parameter: Both actions and activities can have such output parameters, which are used only when an excep-

![Figure 5.17](image-url)  
**Figure 5.17** Example activity with object flow.
tion occurs. As shown for “check order” in Figure 5.17, the output pins and output parameter nodes for exceptions are indicated by a small triangle. A token is placed there only after an abnormal termination. Otherwise, if the action or activity completes successfully, it does not place any object token there.

According to the control flow model of Figure 5.1, the downstream actions are divided into two parallel paths. Since both of them need information from the “Order” object, we let the fork node duplicate the object token. One copy of the token goes to the “get products” action and another copy to the “save order information in archive” action.

The “get products” action takes the “Order” object and retrieves the ordered products from the warehouse. The resulting “Product” objects are placed on the output pin of the action. Since this is the only output pin of the action, the first “Product” token placed on that pin would cause the termination of the action. However, we want the action to continue until it has provided individual tokens for all “Product” objects to be retrieved.

To generalize the problem, we require special input and output pins that associate incoming and outgoing tokens to the same execution of an action. In UML activity diagrams, this is done by declaring pins a stream (depicted by a filled square). For instance, the output pin of the “get products” action in Figure 5.17 is a stream. Actions with streaming output pins can continue to place tokens there while they are executing. Similarly, actions with streaming input pins can continue to accept new input tokens while a single execution of the action is running.

Coming back to our example, we want the subsequent testing actions to treat the retrieved “Product” object tokens separately because each product has to pass its own quality test. Consequently, the pins of the testing actions are not declared as streams again. Since different tests are required for computers and monitors, a decision node is used to direct the products to the right test action. As shown in the example, we can use information about objects and their attributes in the branching conditions.

After the quality test, the products should be collected again before they are assembled into bundles. We can model this by a central buffer node, which is a special object node (labeled as <centralBuffer>) that can be used to manage object flows of various incoming and outgoing edges. Central buffer nodes are not directly connected to actions but to other object nodes or pins. Thus, they provide additional, explicit means for queuing object tokens. In our example, the buffer type “Product” is compatible with both upstream types “Computer” and “Monitor” since they are subtypes according to the class diagram of Figure 5.14.

The “save order information in archive” action has to store statistical information about the order in an archive. If we want to model such persistent storage of data, we can use data store nodes, which are specialized central buffer nodes (labeled as <datastore>). In contrast to central buffer nodes, a data store node keeps all tokens that enter it, copying them when they are chosen to move downstream.

Eventually, the “assemble bundle” action packages all “Product” objects into a “Bundle” object and passes it to the output parameter node of the activity. The action must not start execution unless all ordered products have finished the quality test and are available from the central buffer. This can be guaranteed by the weight
expression, which delays the object flow until as many “Product” tokens are available as order items are contained in the “Order.”

5.4 MODELING ORGANIZATIONAL STRUCTURE

The actions included in activities that describe business processes are executed by specific persons or automated systems within a company. Companies are complex sociotechnical organizations. It is necessary to link the underlying organizational structure of a company to the activities of its business processes in order to describe which actions have to be performed by which organizational entities. This corresponds to the resource and organizational perspectives of workflow modeling discussed in Chapter 2. This section describes how UML can be used to address the following key requirements for modeling organizations and resources:

1. Companies consist of a multitude of organizational entities such as persons, machines, and systems. Actions, for example, in an activity diagram, can be associated with any of these organizational entities. To build a coherent model of a company, all these different organizational entities should be described in one single model together with their specific properties and relationships. Examples of such relationships are leadership hierarchies, ownership and shareholder relationships, department affiliation, project group affiliation, and communication structures. We will use UML object diagrams to model concrete organizations.

2. The organizational structures in companies usually follow typical patterns, such as hierarchically organized leadership structures, functional division of labor in departments, or matrix organizations. With UML, it is possible to flexibly model the majority of these general organizational structures in such a way that concrete organizations can be treated as instances of these structures. General organizational structures can be modeled by UML class diagrams.

3. Finally, the control and object flow description contained in activity diagrams and the organizational view expressed in class and object diagrams have to be linked to each other because actions and activities need to be assigned to the organizational entities that are responsible for their execution. For this purpose, we will introduce the concepts of activity partitions and swim lanes in activity diagrams.

A general introduction to UML object and class diagrams has been presented in Section 5.3. In this section, we will focus on the usage of object and class diagrams for organizational modeling.

5.4.1 Modeling Organizational Structures with Object and Class Diagrams

Figure 5.18 shows an example of an object diagram describing the concrete organization of our exemplary computer hardware sales company. Object diagrams are al-
Figure 5.18  Object diagram describing a simple concrete organization.
ways instances of corresponding class diagrams. In this diagram, there are objects of three different classes:

- Objects of the class Employee for concrete persons
- Objects of the class Department for departments
- Objects of the class Owner for legal persons that own an equity stake of the company

These three classes represent three types of organizational entities in our example company. Different organizational entities can have a different set of properties. In UML, these properties are described by attributes. By associating different kinds of organizational entities with different classes, one can have different attribute sets for each kind of organizational entity. This is reflected in our example in Figure 5.18. Employees have the attributes “position” and “salary.” Departments have the attribute “location” and owners have the attribute “share,” describing the equity share they own of the company. These observations lead us to the corresponding class diagram as in Figure 5.19, which describes the general organizational structure of the company. The object diagram in Figure 5.18 is an instance of this class diagram.

To show the full potential of organizational modeling with class diagrams, we make some more observations about our example company:

- Departments have a number of employees that work for them. The organizational structure consists in our simplified example only of departments.
- Each department has exactly one employee or one owner as head of the department.
- The company has a board of directors that consists of the owners of the company.
- The owners of the company form the board of directors.
- Each employee can work for either another employee or an owner.

![Figure 5.19](image-url) Class diagram representing a simple organizational structure.
Figure 5.20 shows a class diagram that integrates all the observations about our example organization. Now there are classes for the organizational entities: Employee, Owner, CompanyMember, BoardOfDirectors, and Department.

In the version of the class diagram in Figure 5.19, “Employee” has two distinct associations called “works_for.” Employees can either work for another employee or an owner. It is possible to introduce an abstract superclass, “CompanyMember,” making “Employee” and “Owner” subclasses of “CompanyMember.” Then the class diagram can be optimized by having only one association called “works_for,” from “Employee” to “CompanyMember.”

With the abstract superclass “CompanyMember,” it is also possible to model the fact that owners as well as employees can be the head of a department by changing the association “is_head_of” to be between “Department” and the new class “CompanyMember.” As “CompanyMember” is an abstract class, in an object diagram describing a concrete organization, an “Employee” or an “Owner” has to take the place of the “CompanyMember.” The cardinality “1” at the association “is_head_of” expresses that there has to be exactly one head of a department. The hierarchy of the company is built up by the departmental structure of the organization and by the association “works_for.”

In the class diagram of Figure 5.20, we introduce a new class representing the organizational unit “BoardOfDirectors.” The board of directors is built from the set of owners, which is reflected by the aggregation relationship “belongs_to” symbolized by the association line with the diamond symbol. The “Department” class has an aggregation relationship to the class “Employee” because departments consist of employees who work for the department. The cardinality “1” expresses that every employee belongs to exactly one department.

We can now describe the complete concrete organizational structure of our example. If we add the “BoardOfDirectors” and the “belongs_to” associations to the object diagram of Figure 5.18, we get the diagram in Figure 5.21.

**Additional Remarks.** The structure of the class diagram in Figure 5.19 indicates that, in principle, every employee can be subordinate to every other employee or owner, but every employee can only belong to one department. Therefore, the class diagram stipulates a hierarchical department structure.

![Figure 5.20](image-url)  
**Figure 5.20** More sophisticated organizational structure.
Figure 5.21  Complete object diagram for the example company.
It is also possible to describe organizational structures other than hierarchies. For example, many companies have on the one hand functional departments like “production,” “accounting,” and “development,” and on the other hand departments for different product lines. This leads to a two-dimensional matrix organization. To model such an organizational structure, the cardinality “1” between “employee” and “department” has to be changed to “2.” Sometimes, not every position in the matrix is staffed. For example, some employees fulfill the same function for different products. In that case, the cardinality between “employee” and “department” can be changed to “2..*” for a two-dimensional matrix. Figure 5.22 shows an example of an excerpt of an object diagram for a matrix organization with two product lines for monitors and computers. Some objects and associations are left out in the diagram to account for clear arrangement. In this example, we added the department “Procurement” and the employee “Mr. Taylor.” Mr. Taylor is responsible for the procurement for both product lines, so the corresponding class diagram can be the same as in Figure 5.20, but the cardinality between “employee” and “department” has to be “2..*.”

### 5.4.2 Integration of Organizational Structures in Activity Diagrams

Now that we have seen how organizational structures can be modeled using class diagrams and concrete organizations can be described using object diagrams, we have to connect these organizational models to the process models. In UML, this connection is done within an activity diagram using the notational elements *activity partition* and *swim lane*.

Activity partitions divide the set of nodes within an activity into different sections. Their use is not restricted to modeling organizational units. For example, they can also be used to constrain other resources among the nodes of an activity.

Activity diagram nodes can belong to none, one, or more partitions at the same time. Partitions can be divided into subpartitions. Partitions can be visualized in two different ways. The partition name can be written in brackets over the action name within the action symbol as in Figure 5.23a. The other possibility is the use of swim lanes as in Figure 5.23b.

Swim lanes are lines that are drawn through the activity diagram dividing it into different sections. The name of the partition is displayed on the top of the swim lane. In our case, that would be the name of the organizational unit that is responsible for execution of the actions in that partition.

With swim lanes, simple organizational structures can be reflected. In the previous section, we introduced hierarchical and matrix organizations. Simple situations of the two organizational structures can also be displayed by swim lanes. They can be hierarchically structured as shown in Figure 5.24a. Swim lanes can also intersect each other, as in Figure 5.24b, to represent example matrix organizations. Then the actions are associated with multiple partitions at the same time.

The model of the organizational structure can now be integrated into the business process models of our running example. The activity depicted in Figure 5.17 contains a number of actions that have to be executed either by the accounting de-
Figure 5.22 Object diagram excerpt for a matrix organization.
partment or the production department. In Figure 5.25, swim lanes are included in the activity diagram to describe that the actions “check order” and “save order information in archive” are performed by the accounting department and the other actions are performed by the production department.

5.5 MODELING BUSINESS PARTNER INTERACTIONS

So far, we have concentrated on modeling the various dependencies between the different actions of a business process. However, a complementary view of business processes is more centered around the interactions that take place between different participants. Such interactions occur, for example, among the employees of a certain department as well as across department and company borders. In a supply chain, for instance, the involved business partners have to interact in order to coordinate demand and supply of certain materials.
In such cases, the involved participants have to agree on the way they will interact. An interface process, as defined in Chapter 4, constitutes an approach to define the interactions between partners, represented by their provided endpoints. In an interface process, interactions are described from the perspective of one of the involved endpoints. As shown in Chapter 4, an interface process can be described through an activity diagram in which the activities produce or consume interaction events. In some situations however, a more interaction-centric (rather than activity-centric) view of the relevant processes is more appropriate. This view allows modelers to focus on the interactions themselves, and provides a more global perspective on how multiple partners interact, as the description does not focus on the events produced or consumed by a specific participant.

For this purpose, UML provides so-called sequence diagrams. They comprise the participants involved in an interaction. Each of them has a lifeline representing its progress in time (usually from top to bottom). Arrows between the lifelines indicate the passing of a message. The sequence of arrows along the lifelines represents the order of message exchanges.

As an example, we consider the interactions of a hardware sales company with its customers, its warehouse, and a shipping service that is in charge of delivering ordered products to customers. Figure 5.26 shows the corresponding sequence diagram.

This sequence diagram is named “order interactions” and comprises a “Customer,” the “Company,” its “Warehouse,” and the “ShippingService” as participants. Every participant is depicted as a rectangle that contains the name of the par-

---

**Figure 5.25** Exemplary activity with swim lane notation for the organizational entities.
participant and its type. In contrast to the notation of objects in object diagrams, these names and types are not underlined because they represent a certain role rather than a concrete instance. At process enactment time, the role names have to be bound to concrete entities of the specified type. For instance, a concrete customer submits the order and a specific shipping service is selected. If the role name is not referenced later in the diagram, one can also omit it and specify just the role type.

The messages attached to the arrows represent, for example, a request for some provided service, a response to the requester, the transmission of a certain signal, the sending of a certain return value, the transportation of some objects, and so forth. As with action names in activities, there are different degrees of formalization possible, starting from simple keywords down to operation calls with formal parameters.

In UML sequence diagrams, one distinguishes between synchronous (filled arrowhead) and asynchronous (open arrowhead) message passing. The synchronous mode means that the sender stops its activity after sending the message and waits until the corresponding response message arrives. In our example, we use only asynchronous message passing, meaning that the partners remain active after having sent a message independent of the response.

In our example, the “Customer” at first submits the order, which is then checked by the “Company.” Although one should usually abstract from internal actions like “check order” and concentrate on external interactions in sequence diagrams, we can still model such internal actions as self-related messages if they have an impact on the remaining part of the interaction. In our example, this is the case because the
downstream interactions are divided into two alternative interaction fragments (indicated by the keyword “alt” and the subdivided rectangle) which are chosen according to the outcome of the check order action. Either the order is valid and the products can be retrieved from the “Warehouse” and delivered by the “ShippingService,” or the order is not valid and rejected by the “Customer.”

Besides the “alt” operator for alternatives, sequence diagrams also provide other interaction operators that can be used in combination with interaction fragments, for example, the “loop” operator that indicates that a certain fragment is repeated as long as a certain condition holds, or the “par” operator that indicates that several fragments are executed in parallel. Different fragments can also be nested to model more complex interactions.

Such interaction models provide a complementary view on the business processes modeled before. In contrast to the activity diagrams, they usually hide internal actions that do not affect other participants (e.g., the testing actions of Figures 5.1 and 5.17). Nevertheless, the two different views of the business process must be consistent with each other, which means that they have to preserve the order of overlapping actions and events. For example, in both Figure 5.1 and Figure 5.26, the “check order” action comes before the “get products” action.

## 5.6 SYSTEM-SPECIFIC PROCESS MODELS

The business process models presented so far can be used for design, analysis, or documentation purposes. However, another purpose of process models is to support process enactment. In this case, they have to be refined into activities with atomic actions that are not further subdivided. These actions can then either be performed by humans or executed by machines and computers.

At this point, we want to focus on the latter case, in which processes mainly transform information and can, therefore, be enacted with the help of computer systems (i.e., application-to-application processes). We model the available software components of an enterprise and relate their services to the actions of our process model. Thus, we receive a refined, system-specific model that can be used for process enactment. In the terminology introduced in Chapter 4, this type of model corresponds to an integration process.

In principle, such system-specific process descriptions can be used in two ways. The first option is to feed them into a central process engine that has access to all available software components and invokes their services according to the process description (see Figure 5.27a). Thus, the process engine is responsible for managing the various process instances, the control flow, and the object flow.

The second option is to take the more local point of view of a single component that realizes a new service by using a set of services provided by other components. Then, the process model can be used to describe how the invocations of the required services are coordinated in order to realize the desired service (see Figure 5.27b).

For instance, service-oriented architectures consist of distributed software components that make use of existing third-party services in order to provide new services. Since this usually involves components of different business partners,
process descriptions are needed to adjust the invocation behavior among the different partners. The Business Process Execution Language for Web Services (see Chapter 12) is a textual language for implementing such architectures in which process-driven coordination of services takes place.

As an example of system-specific models, we refine the “check order” action used in the order processing activity of Figure 5.17 and specify how existing services are combined to realize this action. For this purpose, we have to break the action down into atomic subtasks like evaluating the customer’s credit rating and checking the available product supplies. We assume that there are software components such as warehouse and customer management systems that provide services for these tasks. This leads to the following requirements:

1. We require a model of available systems and components that abstracts from their internal computations but specifies their provided and required services. For instance, we want to describe that there is an order management system that provides the service to check incoming orders, and, in order to do so, it requires certain warehouse and rating services. For this purpose, we will introduce UML structure diagrams and interface descriptions.

2. Having specified the provided and required services, we want to integrate them into our process models in order to coordinate their invocation. For instance, we want to describe in which way the services required by the order management system are invoked in order to realize the provided order checking service. Since inputs required by one service might be provided as outputs by other services, we have to consider both control and object flow dependencies. The resulting system-specific process models should serve as a basis for computer-based process execution.

UML structure diagrams provide a high-level view of existing information systems, as shown in Figure 5.28. Components are depicted as boxes, omitting details about their internal computations. Provided and required services, in UML called operations, are summarized as interfaces of the components. Provided interfaces
are depicted as a circle connected to the providing component, and required interfaces as a half-circle connected to the requiring component.

For each required interface, another component is needed that can provide a matching interface. In our case, the CustomerManagementSystem provides CustomerServices to the OrderManagementSystem, the FinancialServices component provides the RatingServices interface, and the WarehouseManagementSystem provides the WarehouseServices interface.

Interfaces are specified in a simple form of class diagram, as shown in Figure 5.29. In contrast to classes used for modeling object structures, the focus is not on structural properties and relationships but on operations. An operation signature is defined in the second compartment of the interface symbol by a name and a set of input and output parameters. If there is no more than one output parameter, we can list the input parameters in parentheses and append the output parameter as the return type of the operation at the end. Otherwise, we have to distinguish input and output parameters by the keywords “in,” “out,” or “inout” (see, e.g., the “checkOrder” operation of the “OrderServices” interface).

In contrast to ordinary classes, interfaces cannot be instantiated but can only be used to indicate that a class or component either provides or requires the set of operations defined in the interface. In order to integrate the invocation of these operations in our process models, we introduce call actions for activity diagrams.

In general, call actions represent the invocation of certain behaviors defined in accompanying diagrams. In our case, we use them to call operations of component

**Figure 5.28** Structure diagram example.

**Figure 5.29** Interface specifications.
interfaces, as shown in the system-specific “checkOrder” activity (Figure 5.30). In contrast to ordinary action nodes, the node symbol contains the exact name of the operation to be called. Below, the operation name, the name of the interface or component type providing the operation, is added in brackets. All input and output parameters defined in the operation signature are transformed into input and output pins of the action node. Thus, when defining the control flow between the call action nodes, one has to consider object flow dependencies that arise from the operation’s input and output behavior.

Since, according to the interface description in Figure 5.29, the “checkOrder” operation has an inout parameter of type “Order,” the activity gets a corresponding input parameter node, too. From there, incoming “Order” objects are passed on to the action nodes of the activity until they are eventually placed on the output parameter node shown at the bottom of Figure 5.30.

If any of the involved checks returns a negative result, then the “Order” is rejected and placed at the second output parameter node (shown at the top of Figure 5.30), which is an exception, as indicated by the small triangle. Note that exactly this arrangement of parameter nodes is required if we want to use the activity as an refinement of the “checkOrder” action of Figure 5.17.

The activity involves two checks that can be performed in parallel: First, the customer’s credit rating should have a positive value, and second, the available product supplies of the warehouse should be sufficient to satisfy the demand. Since the “ge-

Figure 5.30  System-specific activity diagram for the checkOrder service.
The “Rating” operation of the “RatingServices” interface requires a “Customer” object as input parameter (see Figure 5.29), we have to insert an action calling the “getCustomer” operation first. This “CustomerManagementSystem” operation retrieves the corresponding “Customer” object from an associated database, which is then passed on to the “getRating” operation.

The two parallel action flows for order checking are enclosed in an interruptible region so that any negative result prevents further effort and directly leads to the “reject order” action causing an exception. However, if both parallel checks are successful, the interruptible region is left, and the “logOrder” operation of the “WarehouseServices” interface is invoked to update the product information stored in the “WarehouseManagementSystem.” Eventually, the checked “Order” is returned as output to the superior process.

As revealed by this example, system-specific process models refine actions and activities of more abstract, business-level process models. Given a mapping of the interfaces to real components with physical addresses (also called deployment description), such system-specific process models can be used for process enactment and coordination of the involved software components. For related work about using activity diagrams in order to integrate applications and software components, the interested reader is referred to [6] and [22].

5.7 SUMMARY

Modeling processes require the description of a number of different perspectives of the process [11, 4]. We have covered five major perspectives of process modeling with UML diagrams: the description of actions and control flow, data and object flow, organizational structure, interaction-centric views, and application integration through system-specific, refined process models for process enactment. Table 5.1 summarizes which UML diagrams we have employed to describe these process modeling perspectives.

For further studies of the UML, the interested reader can find detailed insights into the language concepts in the book by Pender [19]. How to apply the UML for developing information systems from requirements analysis to system design is described, in the work by Maciaszek [14].

<table>
<thead>
<tr>
<th>Table 5.1 Overview of the different UML diagrams</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activity diagram</td>
</tr>
<tr>
<td>Actions and control flow</td>
</tr>
<tr>
<td>Data and object flow</td>
</tr>
<tr>
<td>Organizational structure</td>
</tr>
<tr>
<td>Interaction-centric view</td>
</tr>
<tr>
<td>System-specific models</td>
</tr>
</tbody>
</table>
There are strong efforts underway to further increase the usability of UML for process modeling. The recent revision, UML 2.0, has already improved, among other things, the suitability of activity diagrams. In order to further extend the language according to business process modeling requirements, one can also use the built-in extension mechanisms of UML. These extensibility features allow designers to adapt certain parts of the language to their domain-specific needs while still remaining within the framework of the UML meta-model. For this purpose, so-called stereotypes can be defined that describe semantic extensions as well as syntactical modifications of dedicated meta-model elements. A set of related stereotype definitions forms a UML profile.

Work in progress includes the development of a specialized business process definition profile by the OMG [15]. The objective is to allow groups using a variety of process models, including UML activity diagrams and other process modeling notations, to map to a common meta-model and thus facilitate communication among themselves.

Among others, there are efforts underway to increase the support for collaborating business processes, business process patterns, runtime implications of process definitions, resource assignments, access control, and so on. The extensibility feature of UML will facilitate the efforts to further develop extensions of the UML for business process modeling in order to make it even more powerful and user-friendly.

5.8 EXERCISES

1. Consider the “test computer” and “test monitor” actions in Figure 5.17 and model the case when such a product test fails. For this purpose, you could, e.g., add output pins returning a test report. If the report reveals a negative test result, a substitute product has to be retrieved from the warehouse and the test has to be redone.

2. As preparation for modeling the internals of the testing actions, extend the class diagram of Figure 5.14 as follows. A checklist is associated to each product type. Every such list contains a set of items that describe the properties to be checked for the associated product type. Each item has a property name and a reference value as attributes.

3. Now, refine the “test computer” action of Figure 5.17 into an activity, showing the internals of the action. Model the input and output parameter nodes of the activity according to the pins of the corresponding action node. The activity should contain an archive for all the checklists for the various product types. Whenever a new computer object arrives, the right checklist has to be selected from the archive. You can then freely design your own control and object flow to realize the testing activity.

4. Extend the interaction model of Figure 5.26 with the company’s bank as additional business partner. After ordered products have been delivered to the
customer, the company sends a bill to the customer containing a reference to the bank. Then, the customer can transmit the payment to the company’s bank account. In a second step, try to model that the delivery of the products and the payment can also happen in parallel.

5. Consider the object diagram for the example company in Figure 5.21 and the matrix organization excerpt in Figure 5.22. What would a complete object diagram of the company look like if you combined the two existing diagrams?

6. In the matrix organization in Figure 5.22, we use the organizational entity “Department” both for the functional entities of the company like procurement and accounting, and for the product-oriented entities like monitors and computers. Devise an organizational structure that contains departments and product lines as two distinct organizational entities. Extend the organizational model developed in Section 5.4 with the necessary additional classes. What additional associations have to be defined? How would the object diagram in Figure 5.22 be affected?

7. In Figure 5.21, Mr. Ross is an employee. Now assume that Mr. Ross is not only an employee but also an owner of the company at the same time. How could this be modeled in the class diagram? (Hint: consider multiple inheritance.) How would the object diagram in Figure 5.21 be affected?

REFERENCES


