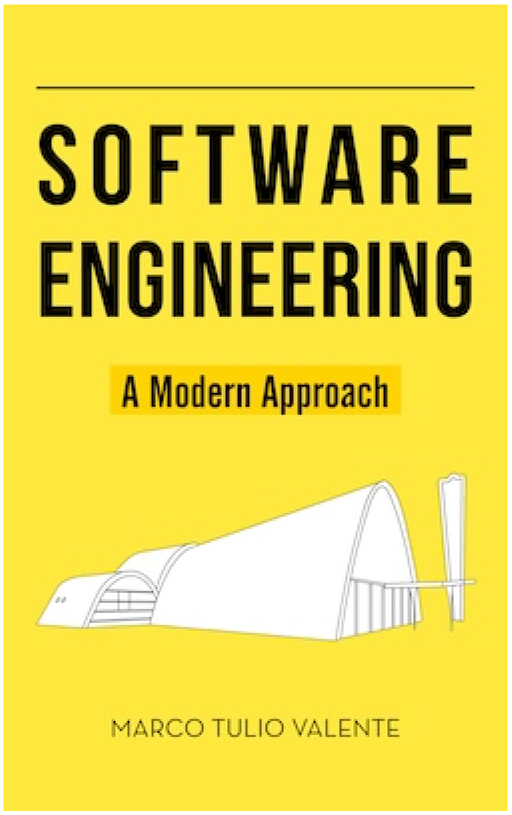


Buy e-book on Leanpub

To report errors or typos, use this form.

Home | Dark Mode | Cite

Software Engineering: A Modern Approach

Marco Tulio Valente

4 Models 🔗

> *All models are wrong, but some models are useful. So the question you need to ask is not "Is the model true?" (it never is) but "Is the model good enough for this particular application?"* – George Box

This chapter starts with a presentation on software models (Section 4.1). Next, we provide an overview of UML, which is the most widely used graphical notation for building software models (Section 4.2). We also emphasize that we will study UML for creating software sketches, rather than detailed technical blueprints. Subsequently, we explore four UML diagrams: Class Diagrams (Section 4.3), Package Diagrams (Section 4.4), Sequence Diagrams (Section 4.5), and Activity Diagrams (Section 4.6).

4.1 Introduction

As we discussed in the previous chapter, requirements document "what" a system should do, using a level of abstraction close to the problem and to the stakeholders. Conversely, the source code offers a concrete, low-level, and executable representation of the system's behavior. Thus, a gap exists between these two worlds: requirements and source code. Software engineers have attempted to bridge this gap since the inception of the field by creating models. Essentially, software models aim to simplify the understanding and analysis of a system. For this reason, they offer more details than requirements specifications but are also less complex than the system's code.

Models are also instrumental in other engineering fields. For example, a civil engineer might create a scale model to demonstrate how the bridge she has been hired to build will look. She could then simulate and verify properties of the bridge, such as maximum load, resistance to wind, waves, earthquakes, etc., by creating mathematical and physical models of it.

Unfortunately, software models—at least to date—have had less impact than the mathematical models widely used in other engineering fields. The reason is that by discarding crucial details, they often eliminate part of the complexity that is essential to the system being modeled. Frederick Brooks comments on this issue in his seminal essay *No Silver Bullet* (link):

> *The complexity of software is an essential property, not an accidental one. Hence, descriptions of a software entity that abstract away its complexity often abstract away its essence. For three centuries, mathematics and the physical sciences made great strides by constructing simplified models of complex phenomena, deriving properties from the models, and verifying those properties by experiment. This paradigm worked because the complexities ignored in the models were not the essential properties of the phenomena. It does not work when the complexities are the essence.*

The quote that opens this chapter, from British statistician George Box, also reflects on the practical use of models. Although he was probably referring to mathematical models, his insight applies to all forms of models, including software models. According to Box, all models are wrong because they are simplifications or approximations of reality. Therefore, the main issue is assessing whether, despite these simplifications, a model retains its value for studying the properties of the object or phenomenon it represents.

Thus, our first goal in this chapter is to set accurate expectations regarding the study of software models. In particular, models can play a significant role in software design. During requirements specification, the focus is primarily on defining the problem that the system will solve. When we move to design activities, the attention shifts towards modeling a solution capable of solving it. After this solution is designed, it must be implemented using programming languages, libraries, frameworks, databases, etc.

Specifically, in this chapter, we will study a subset of the diagrams proposed by UML (Unified Modeling Language). We will begin by describing the history and context that led to the creation of this modeling language. Then, we will examine some of the most important UML diagrams.

In-Depth: Since the 1970s, researchers have investigated the use of mathematical models in Software Engineering through so-called **Formal Methods**. These methods use a mathematical notation—based on logic, set theory, or Petri Nets, for example—to derive formal specifications for software systems. These specifications are precise and unambiguous, and can be used to demonstrate properties of a system before it is implemented. For example, it is possible to prove that a concurrent system doesn’t have deadlocks or race conditions. While this may seem ambitious, it is common in other engineering fields. For example, civil engineers have for centuries used mathematical models to ensure, before construction, that a bridge will support a certain load and resist specific weather conditions. However, the use of mathematical specifications in Software Engineering has not advanced as much as in other engineering fields. Consequently, their use today is primarily restricted to mission-critical systems.

4.2 UML for Creating Sketches

The **Unified Modeling Language (UML)** is a graphical notation for software modeling. The language defines a collection of diagrams to assist in the design of software systems, particularly object-oriented ones. The origins of UML date back to the 1980s, when the object-oriented paradigm was maturing and evolving rapidly. At that time, several object-oriented languages emerged, such as C++, as well as notations for software modeling. The Waterfall model was the dominant process during this period, requiring a large and upfront design phase and the creation of several documents and models, which would then be passed on to the programmers to be converted into code.

UML is the result of a combined effort to unify the graphical notations that developers used in the early 1990s. Specifically, the first version of UML was proposed in 1995 to unify notations being developed independently by three software engineers: Grady Booch, Jim Rumbaugh, and Ivar Jacobson. Simultaneously, tools emerged to create and edit UML diagrams, which were referred to as **CASE tools** (Computer-Aided Software Engineering). The name is inspired by CAD tools (Computer-Aided Design), widely used in traditional engineering fields. The standardization proposed by UML was important to ensure developers could create, access, and edit their models using various CASE tools. In 1997, UML became a standard managed by the OMG, which is an organization funded by software industries.

How to use UML? 🔗

Martin Fowler classifies three ways to use UML: as blueprint, as programming language, or as sketch (link). Let's take a look at each.

UML as blueprint corresponds to the use of UML envisioned by its creators back in the 1990s. In this context, after the requirements specification phase, a set of models—or blueprints—used to be produced to document various aspects of a system. Typically, these models were created by analysts using CASE tools and then handed over to programmers for coding. UML as blueprint is thus recommended when using processes like the Rational Unified Process (RUP). RUP was, in fact, proposed by software engineers with a strong connection to UML. However, as we discussed in Chapter 2, the use of UML to construct detailed and complete models is increasingly rare. For example, in agile methods, we do not have a long and upfront design phase. Instead, design decisions are proposed and implemented throughout the sprints. Consequently, we will not focus here on using UML as blueprint.

UML as programming language corresponds to the use of UML advocated by the OMG after the standardization of the language. The goal was to automatically generate code from UML models. This approach is also known as Model-Driven Development (MDD). In an attempt to make MDD viable, UML was expanded with new features and diagrams being introduced into the language. Consequently, the language gained a reputation for being heavy and complex. However, even after the addition of this extra complexity, the use of UML for code generation remains uncommon, at least for the vast majority of systems.

The third usage scenario is **UML as sketch, which corresponds to the scenario we will study in this book**. In this case, we use UML to build light and informal diagrams of parts of a system, hence the name sketch. These diagrams are used for communication among developers, in two main situations:

- Forward Engineering: in this case, developers use UML to discuss design alternatives before any code is written. For example, consider a user story allocated to the current sprint of a project. Before implementing this story, developers can meet and sketch out the classes they will have to implement, as well as the relationships between them. The goal is to validate the design before coding begins.

- Reverse Engineering: in this case, developers use UML to analyze and discuss a feature that is already implemented in the code. For example, a senior developer might draw some UML diagrams to explain to a newly hired developer how a given feature is implemented. It is usually easier to conduct this explanation using models and graphical diagrams than by explaining each line of code. As the saying goes, "a picture is worth a thousand words".

In both situations, the aim is not to produce detailed and complete models. Thus, there is no need for expensive tools, such as CASE tools. Moreover, the sketches are not used as input for code generation tools. Most of the time, the diagrams are drawn on a whiteboard and, later, photographed and erased. Additionally, only a small subset of UML diagrams is used.

As sketches are small and informal, one may question the need for a modeling language in this case. However, it is better to use a notation that has been in use for years and that many developers know than to invent one's own notation.

Specifically, using UML as sketch avoids two extremes. On the one hand, it avoids the rigid, detailed, and systematic use of UML. On the other hand, it avoids the use of an informal notation, whose semantics may not be clear to the team members. Furthermore, UML is often used in books, tutorials, and documents to explain design techniques. For example, in Chapter 6, we will use UML to illustrate the mechanics of some design patterns. Thus, if you have never had contact with UML, you may have difficulty understanding such diagrams.

In summary, software models and UML diagrams are used for communication among developers. They are created by developers and for developers. This is distinct from the goal of requirements documents, such as use cases, which are created by developers but can also be read and verified by the stakeholders of the system under development.

Real World: In 2013, Sebastian Baltes and Stephan Diehl—both researchers at the University of Trier, in Germany—asked 394 developers to complete a survey on the usage of sketches in software design activities (link). The developers were spread across 32 countries, though the majority were from Germany (54%). The analysis of their responses revealed interesting results about the use of sketches in software projects, as described below:

- 24% of participants created a sketch on the day they answered the survey, and 39% within the previous week.
- 58% of these sketches were archived (digitally: 42%, on paper: 6%, both: 10%).
- 40% of the sketches were created on paper, 18% on whiteboards, and 39% on computers.
- Sketches were created for various purposes: 52% to assist in architecture design, 48% to help in the design of new features, 46% to explain a task to another developer, 45% to analyze requirements, and 44% to assist in understanding a task. The sum exceeds 100% because participants could select multiple answers.
- 48% of the sketches included UML elements, with 9% based entirely on UML.

These findings highlight the common use of sketches by developers, with nearly half including UML elements, which underscores the relevance of studying UML.

UML Diagrams 🔗

UML diagrams are divided into two major categories:

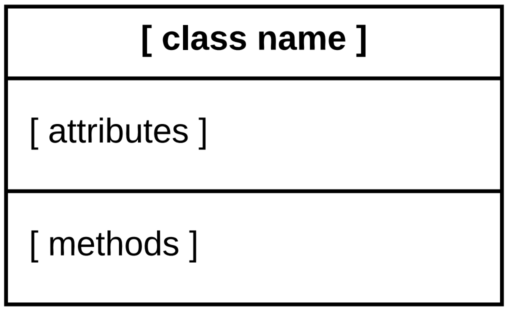
- **Static (or Structural) Diagrams** model the structure and organization of a system, including information about classes, attributes, methods, packages, and more. In this chapter, we will study two static diagrams: Class Diagrams and Package Diagrams.
- **Dynamic (or Behavioral) Diagrams** model events that occur during a system's execution. For example, they can model a sequence of method calls. In this chapter, we will study two dynamic diagrams: Sequence Diagrams and Activity Diagrams.

To clarify the distinction between these categories, static diagrams deal with information that is directly available from the analysis of the code. This information is static because it does not change unless changes are made to the code. Dynamic diagrams, on the other hand, provide a runtime view. They are dynamic because it is common to have different execution flows of the same code. For example, users may run the program with different inputs, select different menu items, etc. In short, if you are interested in representing the structure of the code, you should use static diagrams. If you want to represent the behavior of a program—that is, what can happen during its execution, which methods are actually executed, etc.—you should use a UML dynamic diagram. As a reminder, we already studied Use Case Diagrams, which are considered dynamic diagrams, in Chapter 3, when we introduced techniques for requirements specification.

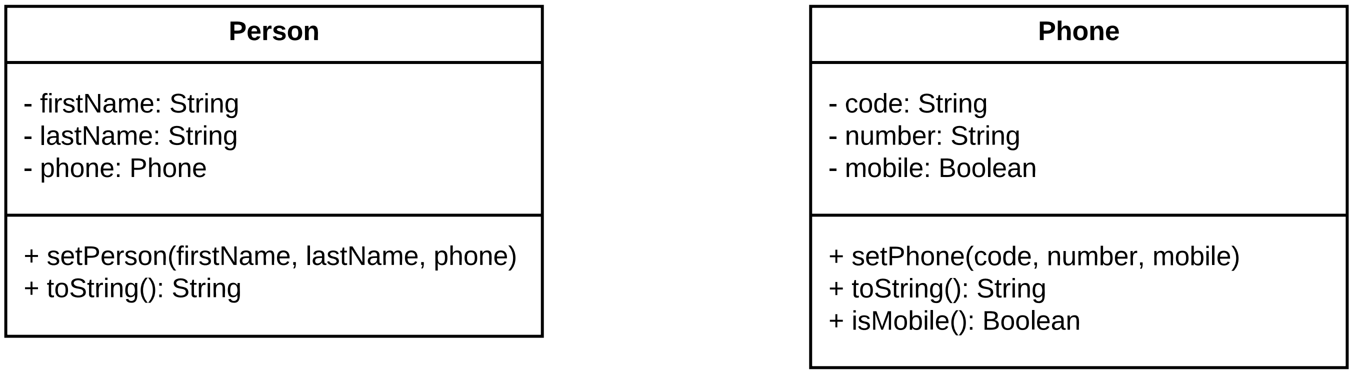
Notice: There are many versions of UML. For the rest of this chapter, we will use the **UML version described in the 3rd edition of the book UML Distilled, by Martin Fowler** (link). This book was one of the first to discuss the use of UML as sketches. Specifically, we will study a subset of the diagrams from UML version 2.0. In addition to covering only four diagrams, we will not present every feature of each. Our challenge when writing this chapter was to select the 20% (or less) of UML features that represent 80% (or more) of its use today when drawing sketches. As an illustration of UML's complexity, the specification of the version 2.5.1 of the language has 796 pages.

4.3 Class Diagrams

Class diagrams are the most common type of UML diagram. They provide a visual representation of a set of classes, offering information about attributes, methods, and relationships among these classes. In a class diagram, each class is represented by a rectangle with three compartments, as shown in the following figure. These compartments contain the class name (usually in bold), attributes, and methods.



The following diagram presents an example with the classes `Person` and `Phone`.

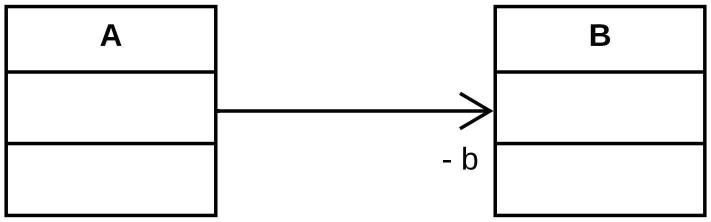


In this diagram, `Person` is a class with three attributes— `firstName`, `lastName`, and `phone` —and two methods— `setPerson` and `toString`. The three attributes are private, as indicated by the "-" symbol before each one. The diagram also specifies the type of each attribute. The two methods are public, as indicated by the "+" symbol. The diagram indicates a second class, called `Phone`, with three private attributes— `code`, `number`, and `mobile` —and three public methods— `setPhone`, `toString`, and `isMobile`. For the methods, the diagram also provide the names of their parameters and the return types.

At first glance, the previous diagram might give the impression that the classes are islands without communication among them. However, one of the main objectives of class diagrams is to visually show the relationships among the classes of a system. Therefore, they can also include arrows to represent three types of relationships: **association**, **inheritance**, and **dependency**. We will describe each of these in the following sections.

4.3.1 Associations

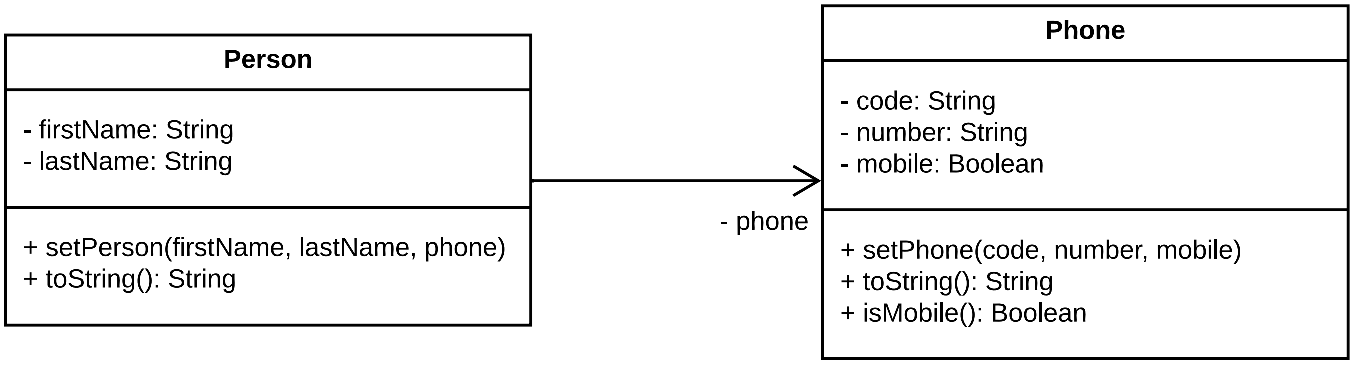
When class A has an attribute `b` of type B, we say there is an association from A to B, represented by an arrow from A to B. At the end of the arrow, we indicate the name of the attribute responsible for the association—in this case, `b`. The following example illustrates this concept (in this example, we only show the information that is relevant; therefore, the compartments for attributes and methods are empty):



To further clarify, we also show the code for classes A and B:

```
class A {
   ...
   private B b;
   ...
}

class B {
   ...
}
```

By using associations, we can transform the diagram with the "isolated" classes `Person` and `Phone` into the following one:

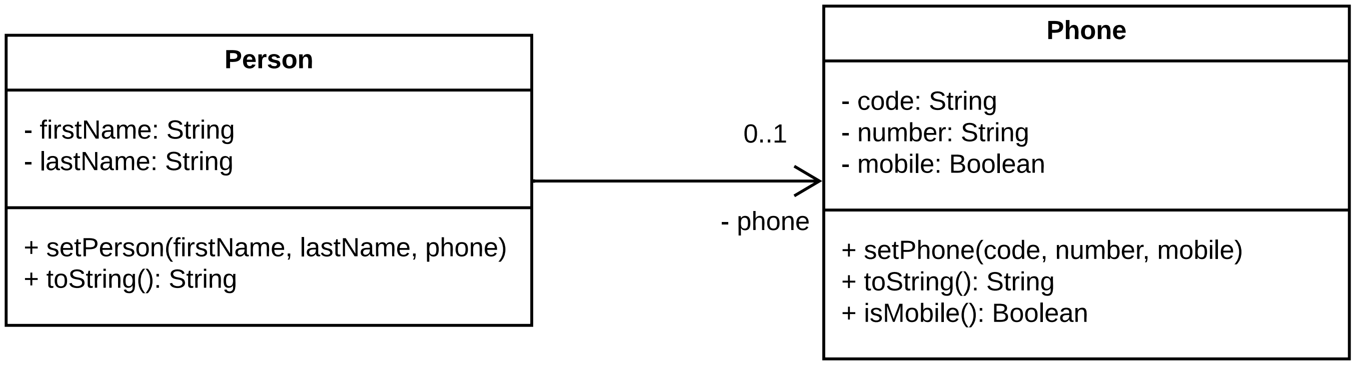


The two versions of the diagram are semantically identical. However, in the first version, the classes appear isolated, while in the second version, it becomes visually clear that there is an association from `Person` to `Phone`.

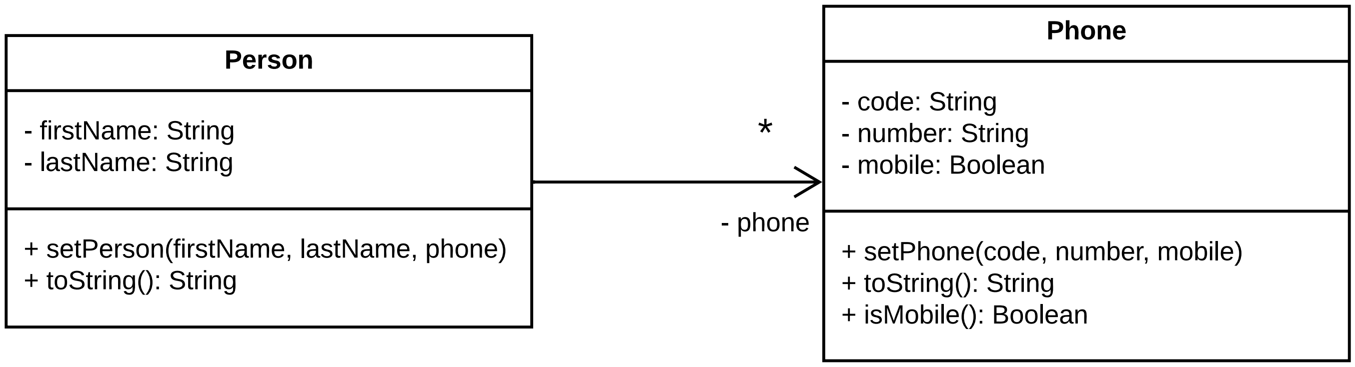
To elaborate, in both diagrams, `Person` has an attribute `phone` of type `Phone`. The key difference is that in the first version, this attribute is represented in the attribute compartment of the class `Person`, whereas in the second version, it is represented outside of that compartment—specifically, at the end of the arrow linking `Person` to `Phone`. This representation emphasizes that the attribute belongs to `Person`, but it points to a `Phone`.

Associations often include **multiplicity** information, which indicates how many objects can be associated with the attribute responsible for the association. The most common multiplicities include the following: 1 (exactly one object), 0..1 (zero or one object), and * (zero or more objects).

In the following example, the multiplicity of the association between `Person` and `Phone` is 0..1. This information is indicated above the name of the attribute responsible for the association, in this case, `phone`. It means that a `Person` can have zero or only one phone. In programming terms, the attribute `phone` of `Person` can have the value `null`, indicating that the `Person` in question may not have a `Phone`, or it can refer to just one `Phone` object.



In the next example, the semantics are different. In this case, a `Person` can be associated with multiple `Phone` objects, including none. This multiplicity is represented by the * above the arrow of the association.

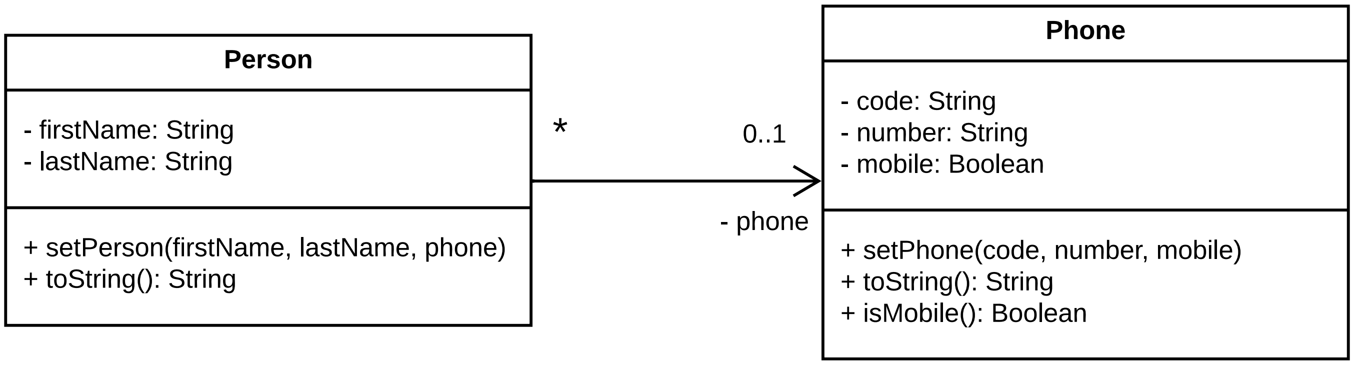


In this latter example, the attribute `phone` is an array, as illustrated in the following code:

```
class Person {
   private Phone[] phone;
   ...
}

class Phone {
      ...
  }
```

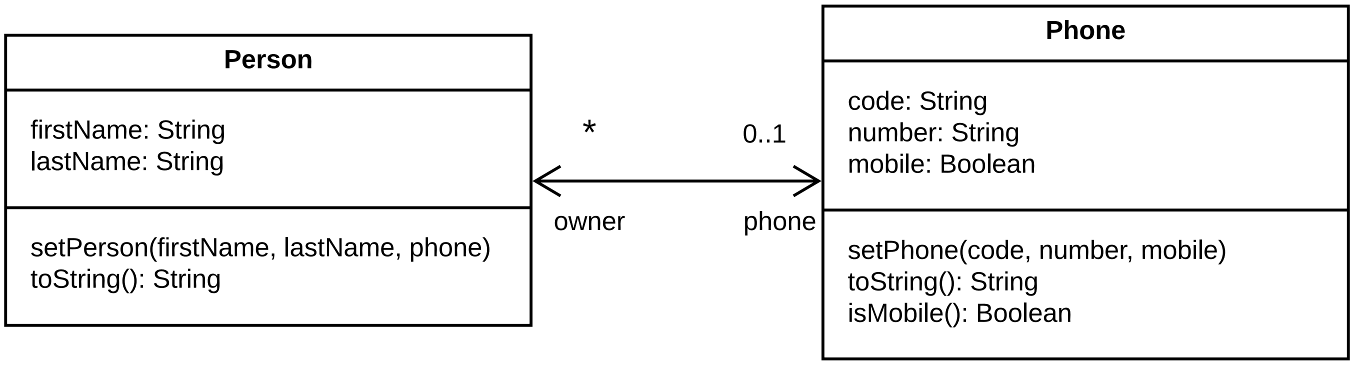
A reader might wonder whether a `Person` should have a maximum of one `Phone` (0..1) or any number of `Phone` (*). The answer is simple: it depends on the requirements of the system. In fact, the customers are the ones who should answer this question. For our purposes, what matters is that class diagrams can model both scenarios.

In some cases, multiplicity is also shown at the opposite end of the arrow, as illustrated in the next example.



In this diagram, there is a second multiplicity at the opposite arrow end, denoted by the symbol *, indicating that a `Phone` can be associated with more than one `Person`. This means that two distinct people can share the same `Phone`. However, the association remains unidirectional, as `Phone` does not have an attribute storing the `Person` instances it refers to. For this reason, given a `Person`, we can easily retrieve their `Phone` by accessing the attribute `phone`. However, given a `Phone`, it is not possible to determine, by directly accessing an attribute, which `Person` instances are associated with it.

To conclude, consider a scenario where it is important to navigate both ways in the association—from `Person` to `Phone` and also from `Phone` to `Person`. The solution to this requirement is to use a **bidirectional association** by adding an arrow at each end of the line connecting the classes, as illustrated in the next diagram.



To clarify the semantics of bidirectional associations, here is the code for both classes:

```
class Person {
   private Phone phone;
   ...
}

class Phone {
   private Person[] owner;
   ...
}
```

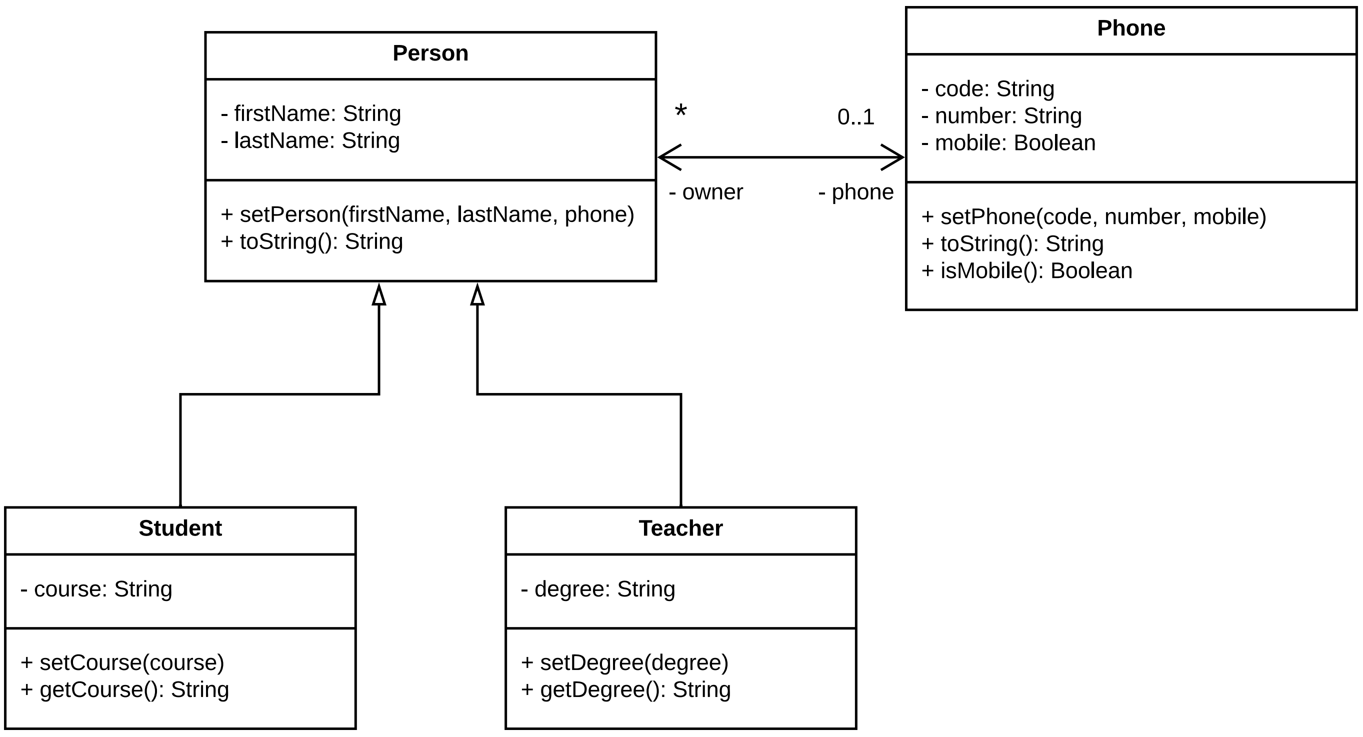
In this code, `Person` has a private attribute `phone` of type `Phone`, which could be `null`; thereby, this comply with the 0..1 end of the bidirectional association. Furthermore, `Phone` has a private array, named `owner`, that references `Person` objects; thus, this complies with the * end of the same association.

In the last class diagram, we omitted all visibility symbols, both public (+) and private (-). This omission was intentional to emphasize that we are using UML for creating sketches. Therefore, the diagrams do not need to be syntactically perfect. Minor mistakes or omissions are tolerated, especially when they do not affect the purpose of the diagram.

In-Depth: UML—depending on the version in use—admits different notations for associations. For example, sometimes, an association name is provided, which is shown along the arrow that connects the two classes. In other cases, for bidirectional associations, the two arrows are omitted since UML standardization states: “an association in which neither end is marked with a navigability arrow is navigable in both directions.” However, these alternative notations can be confusing or even ambiguous. For instance, Gonzalo Génova and two researchers from the University of Madrid observed the following about bidirectional associations without arrows: “Unfortunately, this leads to an ambiguity in the graphical notation, because we cannot distinguish between bidirectional associations and associations with unspecified navigability. Or, worse, unspecified associations are assumed to be bidirectional without further analysis" (link, Section 3, fourth paragraph). Two other concepts frequently discussed in the context of UML associations are composition and aggregation. In a composition, the destination class cannot exist independently of the source class. Conversely, when the two classes have independent life cycles, we have an aggregation. In practice, these concepts often generate confusion, which is why we have not explored them in this chapter. This view is shared by other authors. For example, Fowler states that "aggregation is strictly meaningless; so, I recommend that you ignore this concept in your diagrams" (link, page 68).

4.3.2 Inheritance

In class diagrams, inheritance is represented using arrows with an unfilled end. These arrows connect subclasses to their base class. In the following diagram, for example, `Student` and `Teacher` are subclasses of `Person`. As is typical in object-oriented programming, subclasses inherit all the attributes and methods from the base class, while also being able to add new ones. For example, `Student` has a `course` attribute and `Teacher` has a `degree`.



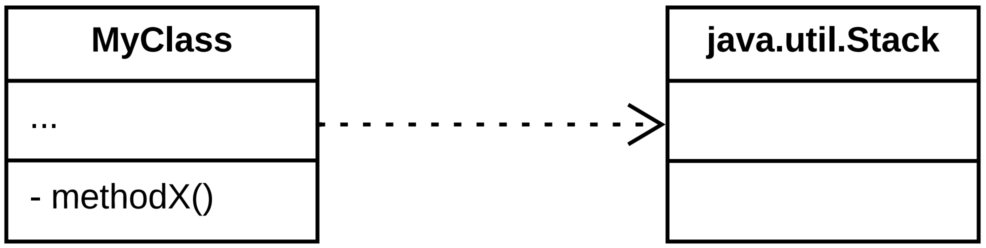
4.3.3 Dependencies

A dependency exists from class A to class B, represented by a dashed line from A to B, when class A uses class B, but this use is not through an association (i.e., A does not have an attribute of type B) or inheritance (i.e., A is not a subclass of B). Dependencies occur, for example, when a method in A has a parameter or local variable of type B or when a method in A throws an exception of type B. A dependency is a weaker form of relationship between classes than associations and inheritance.

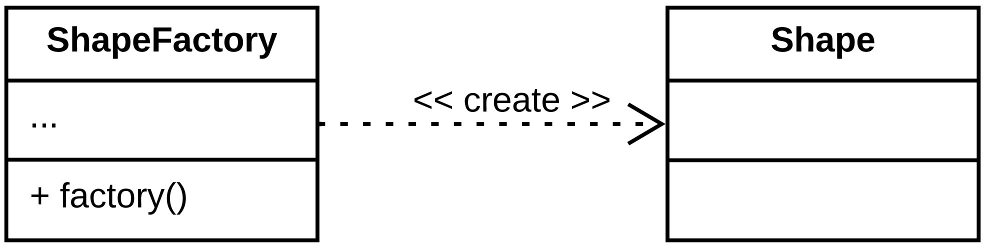
To illustrate the use of dependencies, consider the following code:

```
import java.util.Stack;

class MyClass {
   ...
   private void methodX() {
     Stack stack = new Stack();
     ...
   }
}
```

In this example, `methodX` of `MyClass` has a local variable of type `java.util.Stack`. Consequently, we say that there is a dependency from `MyClass to java.util.Stack`, which is modeled as follows:



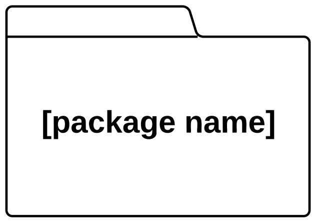
Often, along the dashed arrow, we provide information about the type of the dependency, using terms such as “create” (to indicate that the source class creates objects of the target class) or “call” (to indicate that the source class calls methods from the target class). These words are written between << and >> signs. In the following diagram, for example, `ShapeFactory` is a class that creates `Shape` objects.



A class can have many dependencies. For this reason, it is uncommon to represent all of them in class diagrams. Instead, we only show the most important dependencies and those directly related to the design decisions we intend to model.

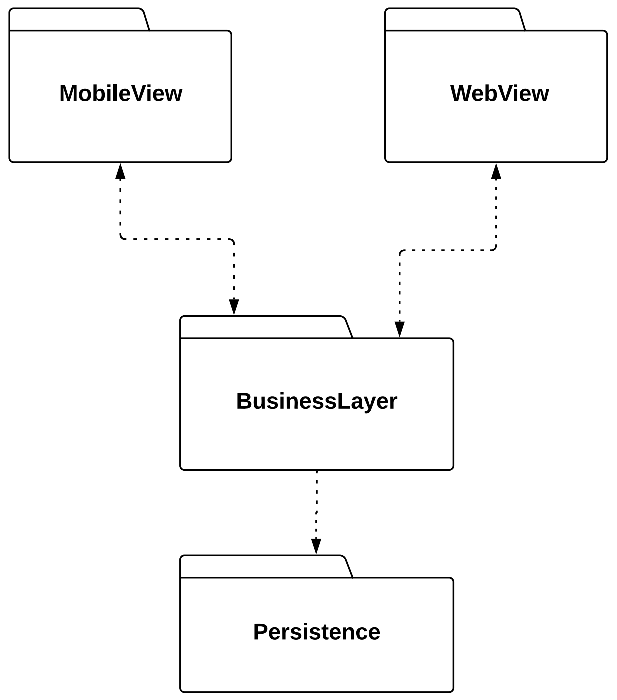
4.4 Package Diagrams

Package diagrams are recommended when we want to provide a higher-level view of a system, showing only groups of classes—that is, packages—and the relevant dependencies among them. For this purpose, UML defines a special rectangle to represent packages, as shown below:



Unlike class rectangles, the package rectangle includes only the package name (in bold). It also has a distinct detail at the top, in the form of a trapezoid, to distinguish it from class rectangles.

Next, we show an example of a package diagram. It refers to a system with four packages: `MobileView`, `WebView`, `BusinessLayer`, and `Persistence`. The diagram also represents the dependencies—dashed arrows—that exist between them. Both `View` packages use classes from `BusinessLayer`. Conversely, classes from `BusinessLayer` also use classes from the `View` packages, for example, to notify them of some events. Therefore, the arrows connecting `MobileView` and `WebView` to `BusinessLayer` are bidirectional. Finally, only `BusinessLayer` uses the `Persistence` package.



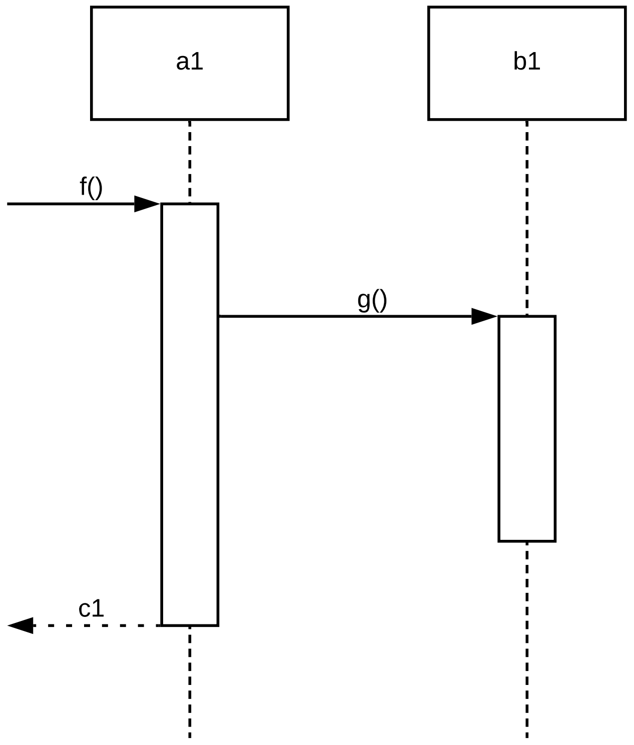
To conclude, we would like to add two important observations:

- In package diagrams, dependencies do not indicate how many classes in the source package depend on classes in the destination package. For example, consider two packages P1 and P2, each with 100 classes. If a single class from P1 uses a single class from P2, this is sufficient to establish a dependency from P1 to P2.
- In package diagrams, a single arrow type is used to represent any relationship. This differs from class diagrams, where associations and inheritance are represented by continuous arrows, while other dependencies are represented by dashed arrows.

4.5 Sequence Diagrams 🔗

Sequence diagrams are dynamic diagrams, also known as behavioral diagrams. Rather than modeling classes, they model objects within a system. Additionally, they provide information about the methods that are executed in a specific usage scenario of a program. Thus, they are recommended when we need to explain the behavior of a system in a given situation. For example, at the end of this section, we will show a sequence diagram that illustrates the methods called when a client uses an ATM to perform a deposit operation.

To introduce sequence diagrams, let's examine the following diagram. Although simple, it illustrates the notation used in such diagrams. As previously mentioned, the diagrams model objects, which are represented as rectangles arranged in the top row. In our example, two objects are represented, named `a1` and `b1`. Below each object, a vertical line is drawn in two forms: (1) a dashed line indicates the object is inactive, i.e., none of its methods are being executed; (2) a rectangular shape indicates one of the object's methods has been called and is currently under execution. When the execution ends, the line returns to the dashed form. A horizontal arrow marks the execution start. The return of the call is indicated by a dashed arrow, labeled with the name of the returned object. However, this arrow is sometimes omitted, such as in the case of the `g` method call. Two reasons justify this omission: either the return type is `void`, or the returned object is not relevant enough to be represented.

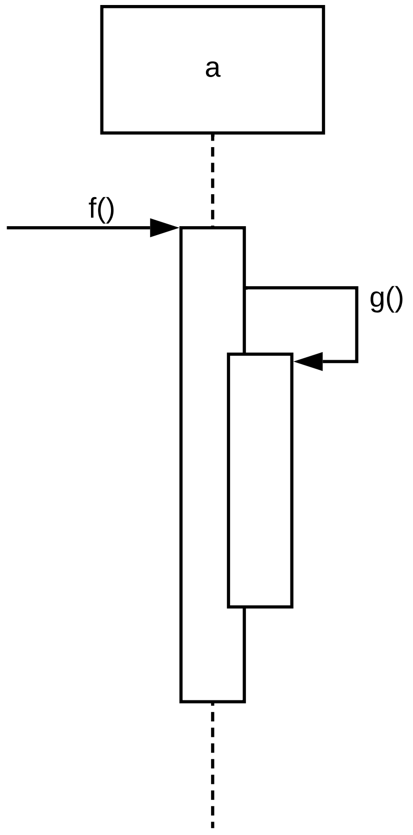


While the previous example includes only two objects (`a1` and `b1`), a sequence diagram can incorporate more objects. However, the number of objects should be limited to maintain clarity, as an overly complex diagram would become difficult to understand. For example, it might not fit on a single page.

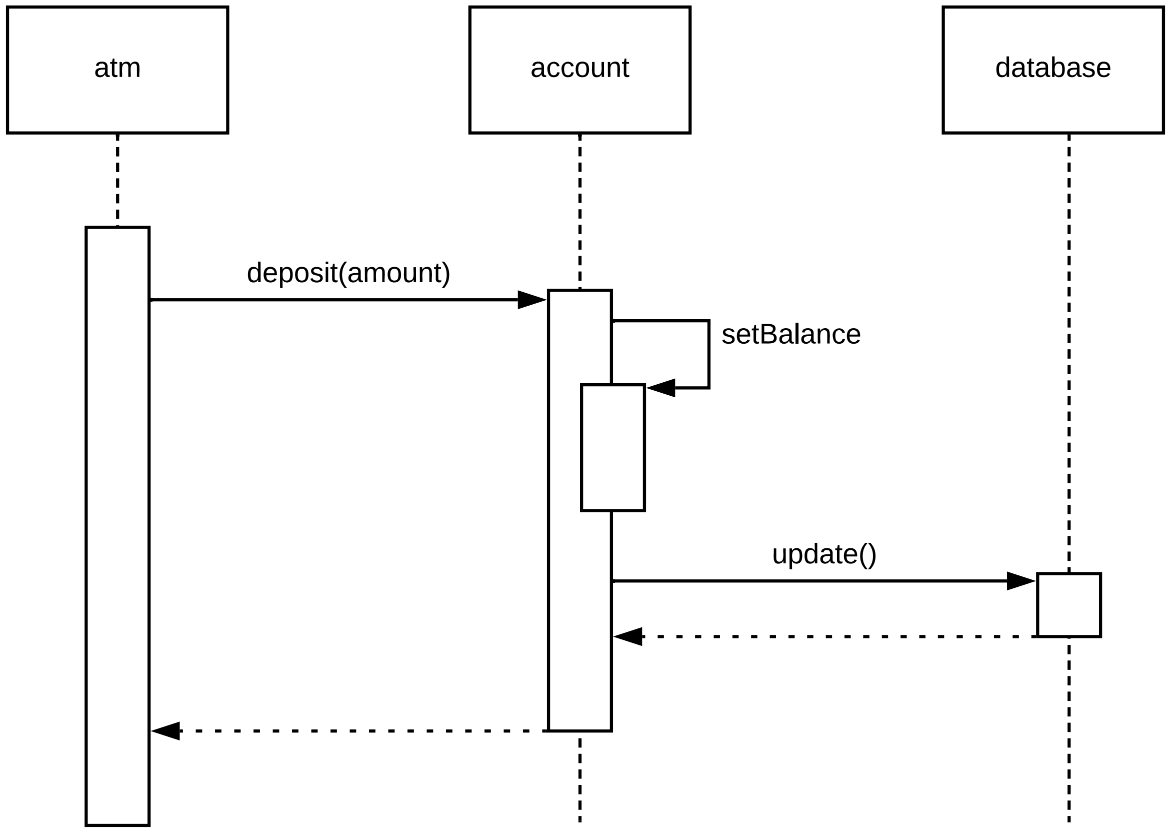
An object can alternate between active and inactive states multiple times within the same diagram. In other words, it can execute a method, become inactive, execute another method, go inactive again, and so on. Additionally, there's also a special case when an object invokes a method on itself, i.e., when it calls a method using `this`. To illustrate this case, consider the following code:

```
class A {
  void g() {
    ...
  }

  void f() {
    ...
    g();
    ...
  }

  main() {
    A a = new A();
    a.f();
  }
}
```

The execution of this program is illustrated in the following diagram. Note how the call to `g()` from `f()` results in a new rectangle emerging from the rectangle representing `f()` 's activation.

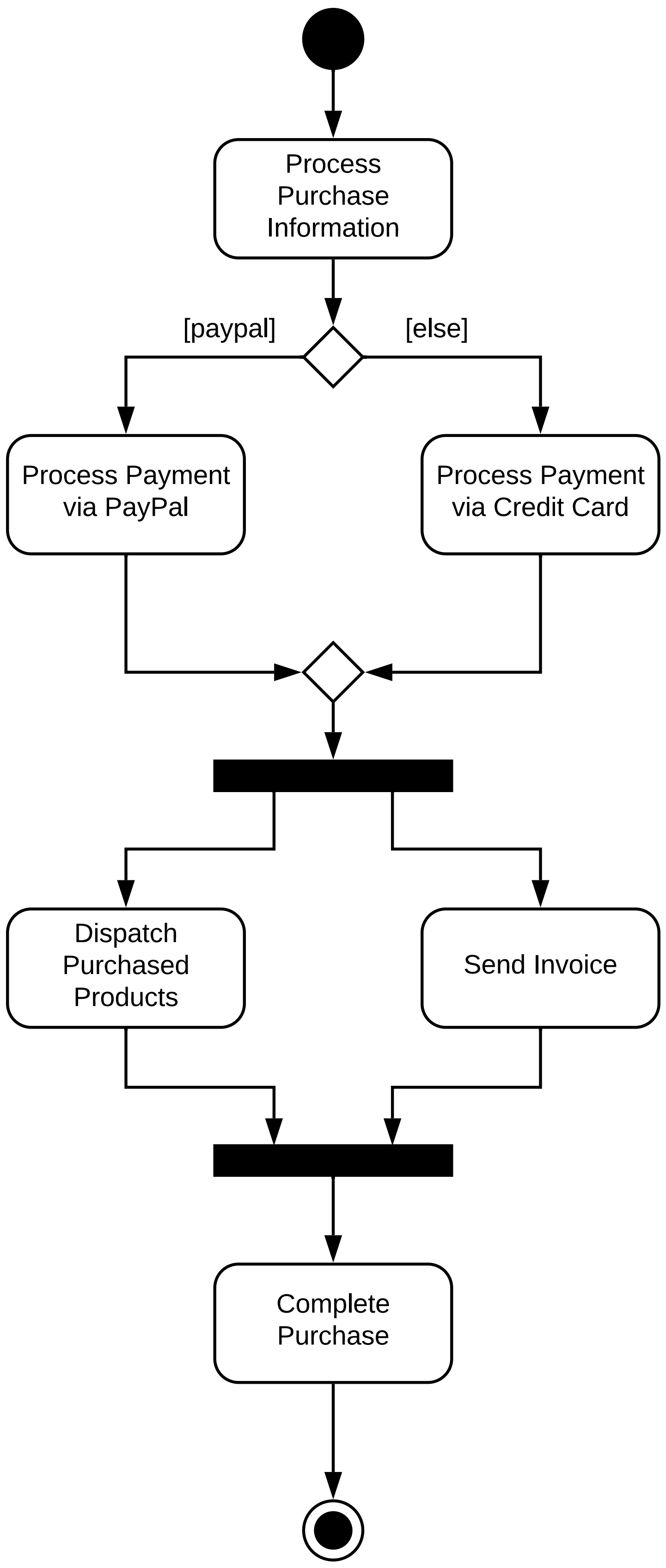


The next diagram shows a more realistic scenario, illustrating the methods called when a client uses an ATM to make a deposit into their account.



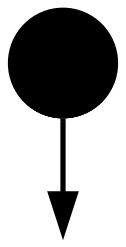
4.6 Activity Diagrams

Activity diagrams represent, at a high level, a business process flow. The main elements of these diagrams are **actions**, represented by rectangles. There are also **control** elements that define the execution order of the actions. The following figure shows an activity diagram that models the process followed after a user completes a purchase in an online store. For this example, we assume that the products are already in the shopping cart.

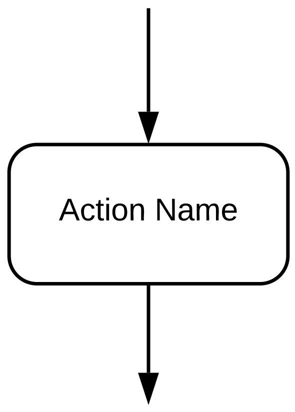

Process Purchase Information
[paypal]
[else]
Process Payment via PayPal
Process Payment via Credit Card
Dispatch Purchased Products
Send Invoice
Complete Purchase

To understand the semantics of an activity diagram, we can imagine a token moving through the nodes of the diagram. We will use this token concept to explain the semantics of each node in such diagrams.

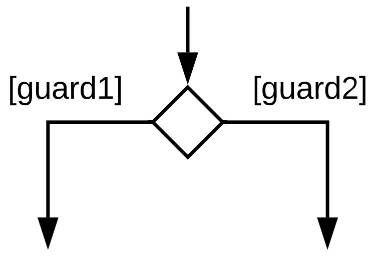
Initial Node: This node creates a token to start the execution. It then passes the token to its output flow. By definition, the initial node has no input flow.



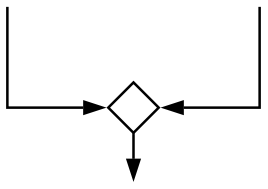
Actions: These nodes have a single input flow and a single output flow. For an action to be executed, a token needs to arrive at its input flow. After the action is completed, the token is passed on to the output flow.



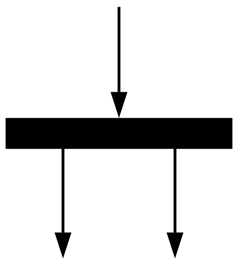
Decisions: These nodes have a single input flow and two or more output flows. Each output flow has an associated boolean expression, called a guard. To make a decision, a token must be received on the input flow. When this happens, the token is passed to the output flow whose guard condition is true. Therefore, the guard conditions should be mutually exclusive.



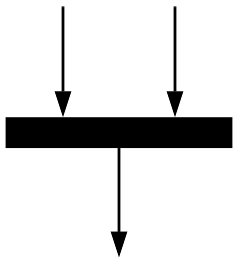
Merges: These nodes can have multiple input flows but only a single output flow. When a token arrives at an input flow, it is immediately passed to the output flow. Merges are used to join the flows from decision nodes.



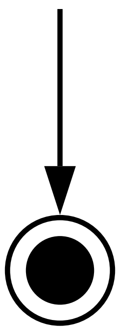
Forks: These nodes have a single input flow and one or more output flows. They act as token multipliers: when they receive a token on the input flow, they create and pass identical tokens to each output flow. As a result, multiple parallel processes begin executing.



Joins: These nodes have multiple input flows but a single output flow. They act as token sinks: they wait for tokens to arrive at all input flows. When this happens, they pass a single token to the output. Therefore, joins are used to synchronize processes, converting multiple execution flows into a single flow.



Final Node: This node can have multiple input flows but has no output flows. When a token arrives at any of the input flows, the execution of the activity diagram terminates.



In-Depth: There are at least three other alternatives for modeling processes:

- **Flowcharts,** which emerged as soon as the first programs for modern computers began to be developed. Activity diagrams are similar to flowcharts; however, they include support for concurrency, via forks and joins.
- **Petri Nets** are a graphical notation, proposed by German mathematician Carl Adam Petri in 1962, for modeling concurrent systems. Petri nets have a graphical representation and also use tokens to mark the current state of the system. They also have a formal definition. However, compared with Activity Diagrams, they offer a more complex notation.
- **BPMN** (Business Process Model and Notation) is a more recent effort, initiated in the 2000s, to provide a user-friendly graphical notation for modeling business processes that is more accessible than that of activity diagrams. Its primary objective is to ensure that business analysts can read, interpret, and validate these diagrams.

Bibliography

Martin Fowler. UML Distilled: A Brief Guide to the Standard Object Modeling Language. Addison-Wesley, 2003.

Grady Booch, James Rumbaugh, Ivar Jacobson. The Unified Modeling Language User Guide. Addison-Wesley, 2005.

Craig Larman. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development. Prentice-Hall, 2004.

Exercises

1. Explain and discuss the three possible uses of UML: (a) as blueprint; (b) as sketch; (c) as programming language.

2. Describe scenarios where class diagrams can help with (a) reverse engineering tasks; (b) forward engineering tasks.

3. Model the following scenarios using class diagrams. The class names are written in a different font style.

a. A `BankAccount` has exactly one `Customer`. A `Customer`, in turn, may have several `BankAccount`. Navigation is possible in both directions.

b. `SavingsAccount` and `SalaryAccount` are subclasses of `BankAccount`.

c. `BankAccount` has a local variable of type `Database`.

d. `OrderLine` refers to a single `Order` (without navigation). An `Order` can have several `OrderLine` (with navigation).

e. `Student` has attributes `name`, `course`, and `GPA` (all private); and methods `getCourse()` and `cancelEnrollment()`, both public.

4. Draw a class diagram to represent the following scenario about scientific journals:

   - A `Journal` has a `title`, `ISSN`, and `publisher`.
   - A `Journal` has several `Edition`, which have the following attributes: `number`, `volume`, and `date`.
   - An `Article` has a `title` and `author`. An `Article` is published in an `Edition`, which must have at least 10 and at most 15 `Article`.

5. Draw a class diagram for the following class:

```
public class HelloWorldGUI {

  public static void main(String[] args) {
    JFrame frame = new JFrame("Hello world!");
    frame.setVisible(true);
  }
}
```

6. Draw a class diagram for the following class:

```
class HelloWorldGUI extends JFrame {

   public HelloWorldGUI() {
     super("Hello world!");
```

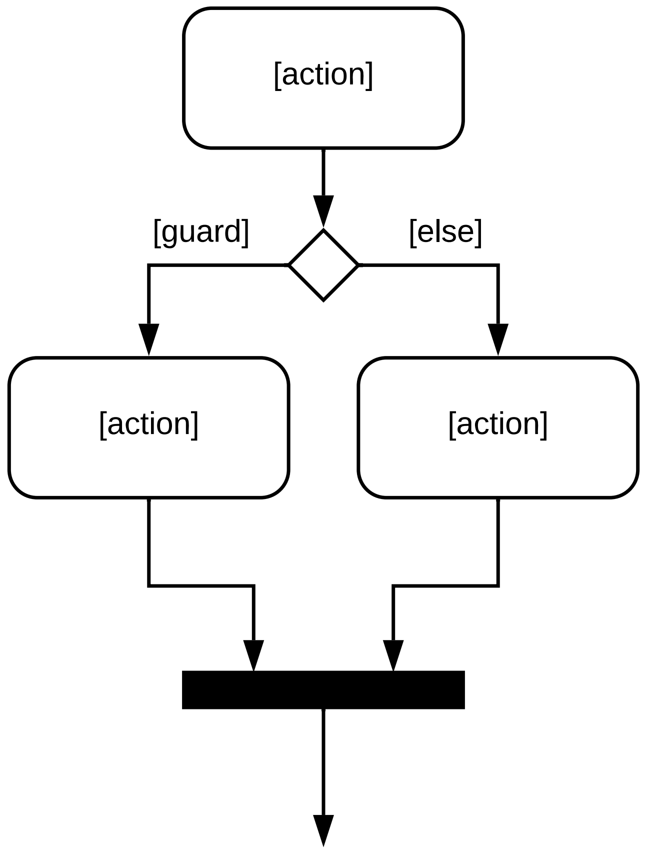
```
    }

    public static void main(String[] args) {
      HelloWorldGUI frame = new HelloWorldGUI();
      frame.setVisible(true);
    }
}
```

7. Draw a sequence diagram for the following code. The diagram should start with the call `a.m5()`.

```
A a = new A(); // global variables
B b = new B();
C c = new C();

class C {
   void m1() { ... }
}

class B {
   void m2() { ... c.m1(); ... this.m3(); ... }
   void m3() { ... c.m1(); ... }
   void m4() { ... }
}

class A {
   void m5() { ... b.m2(); ... b.m3(); ... b.m4(); ...  }
}
```

8. In activity diagrams, explain the difference between merges and joins.

9. What is the error in the following activity diagram? Modify the diagram to fix this error and therefore reflect the intention of the software designer.


[action]
[guard]
[else]
[action]
[action]