

CHAPTER 3

Fundamental UML

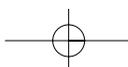
To effectively use UML, we must understand how we represent diagrams in code.

Before attempting to understand how the UML can be used on a software project, we first must understand some of the fundamental elements that compose the UML. By understanding these fundamental elements, we gain insight into the building blocks of the UML at all levels. In this chapter, we begin our studies of many of the most commonly used elements within the UML and how these elements map to the Java language.

This type of discussion often is centered around the UML metamodel. This metamodel describes many of the entities and the relationships between these entities that compose the UML. While this discussion is important, the metamodel becomes truly important for those individuals responsible for developing modeling tools. For the majority of corporations, focusing on the metamodel to gain a deeper understanding of the UML may not be the best use of their time. Our discussion in this chapter takes a different approach. We'll emphasize how the UML is built from the ground up, emphasizing the most often used elements.

3.0 Models and Views

The UML is more than a set of disjointed diagrams. Instead of examining the UML from a diagram-centric perspective, let's turn our attention to an illustration of the UML from three different perspectives. Figure 3.1 depicts three divisions within the UML. Further insight into these divisions enables us to realize



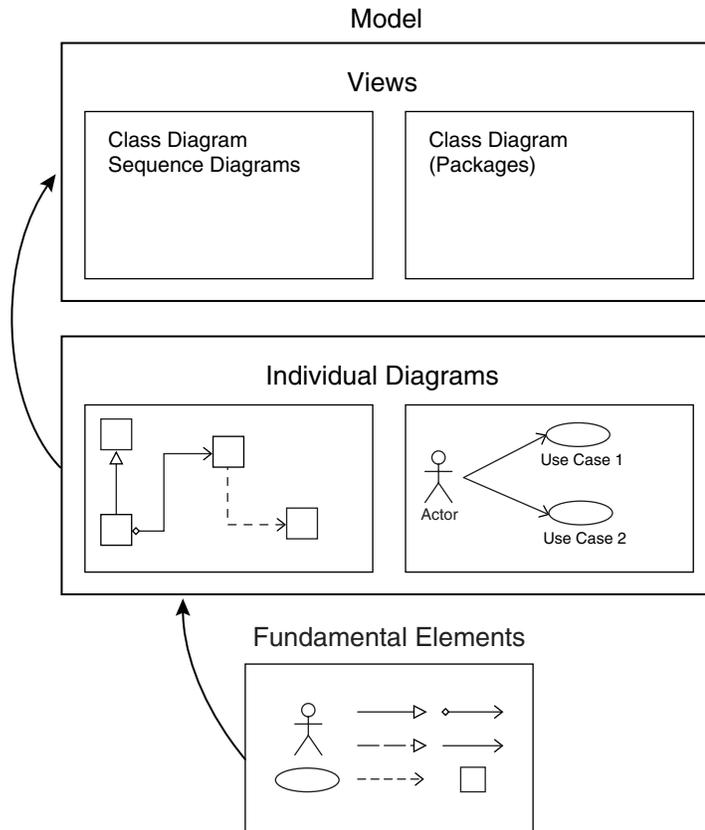
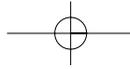
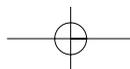


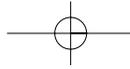
Figure 3.1 UML Perspectives

one of the greatest benefits of modeling, which is creating different views of our software system.

3.0.1 Fundamental Elements

At the lowest level in Figure 3.1 exist the fundamental elements. These basic building blocks are the elements of which diagrams are composed. By themselves, these elements contribute little to the specification of a software system. However, understanding the intent of each element enables us to create precise diagrams because each of the elements has a very unambiguous meaning. These lower-level elements are described in detail in section 3.2, later in this chapter. Keep in mind that once we understand the characteristics of an element, these characteristics apply wherever that element is used.





3.0.2 Diagrams

Above the fundamental elements is the perspective on the UML that many of us are familiar with. The individual diagrams contribute more to the specification of a software system than a single building block. In essence, we can think of a diagram as the composition of many of the fundamental elements. For the majority of us, diagrams play the most important role in contributing to the specification of our software system. These diagrams are the mechanism that developers use to communicate and solve problems in the complex aspects of the system. For instance, class diagrams, which describe the structural relationships that exist among the classes within our system, can guide developers in understanding our software system's class structure.

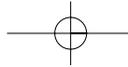
As we begin to incorporate the UML into our environment, it's typical to begin by using the individual diagrams to communicate our software structure and solve problems in the challenging design scenarios. For developers, the most common diagram is the class diagram. Each of the other diagrams, however, plays an important role as well. Because of the specific nature of each diagram, it can be quite effective to use diagrams in conjunction with one another to help us more fully understand our system.

3.0.3 Views

As we become more proficient in modeling, we begin to realize that using a combination of diagrams to communicate is most effective. For instance, a class diagram is valuable in communicating the structural relationships that exist

Models and Views

A *model* is a self-contained representation of a system. Given a model, a user need not have any other information from other models to interpret the system. A *view* can be thought of as any artifact that helps to simplify the representation of our system. A view is a slice through a model, whereas a model is a complete view of a system. In this regard, a view is a subset of a model, and in fact, a model is a view, albeit a complete one. A view focuses on communicating the system from a particular perspective. Views almost always omit elements of a model not relevant to the given situation. In this regard, views describe the architecturally significant elements of a model from a particular perspective. The differences
(continues)



(continued)

between a model and a view are subtle. Our intent in this discussion is to help the reader understand one of the primary advantages in modeling—that is, to create different representations of the system from different perspectives to aid in communication, while maintaining consistency throughout.

between the individual classes within our application. However, a class diagram says nothing about the ordering of messages sent between the objects within our application. By combining a class diagram with a diagram whose intent is to communicate our system's dynamics, the ability to communicate our system's overall intent becomes more powerful.

The combination of a class diagram with a diagram whose intent is to communicate our system's dynamics is a *view*. A view is a depiction of our system from a particular perspective. By combining diagrams to form complete views into the system, we have the innate ability to represent the system from many different perspectives. In Philippe Kruchten's article, "Architectural Blueprints—The 4+1 View Model of Software Architecture," he describes five distinct views from which individuals associated with the software development process actually see the system [KRUCHTEN95]. Figure 3.2 shows these views, and Table 3.1 describes them. As we model our applications, we can create complete models, each containing multiple views. While each of these views represents the system from a different perspective, each represents it from a perspective that is significant to different individuals associated with the development initiative.

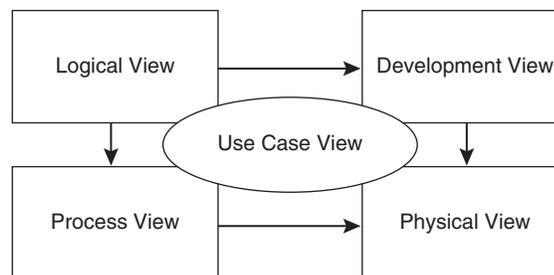
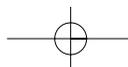
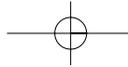


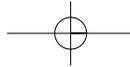
Figure 3.2 The 4+1 View Model of Software Architecture

Adapted from Kruchten, Philippe, "The 4+1 View Model of Software Architecture," IEEE Software, November 1995.



**Table 3.1** View Descriptions

View	Description
Use case	This view documents the system from the customer's perspective. Terminology used in this view should be domain specific. Depending on the technical nature of our audience, we should avoid obscure technical terms. Diagrams most common in this view are the use case diagrams and, less common, activity diagrams. Organizations transitioning to the UML may wish to work only with use case diagrams early and experiment with activity diagrams over time.
Design	This view documents the system from designer's and architect's perspective. Diagrams most common in this view are class and interaction diagrams (either sequence or collaboration), as well as package diagrams illustrating the package structure of our Java application.
Development	This view documents the components that the system is composed of. This view typically contains component diagrams. Except for the most complex Java applications, this view is optional.
Process	This view documents the processes and threads that compose our application. These processes and threads typically are captured on class diagrams using an active class. Because of the advanced nature of active classes, coupled with the volume of use, active classes are beyond the scope of this discussion. For information, refer to [BOOCH99].
Physical	This view documents the system topology. Deployment diagrams that compose this view illustrate the physical nodes and devices that make up the application, as well as the connections that exist between them.



These different views are extremely important because end users, developers, and project stakeholders most likely have different agendas, and each looks at the system quite differently. While each of these perspectives might be different, they represent the same system and should be consistent in the information they convey. In addition, our views can be used to validate each other. The specification contained within one view is consistent with the specification within another. Because of this consistency, we can trace the specification in one view through the realization of that specification in another. The result is an excellent way to ensure that when requirements in our use case view change, they can be traced through the other views, enabling us to make the appropriate changes. This concept of traceability is further discussed in Chapter 4.

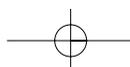
The views in Figure 3.2 may not be the only views from which we look at the system. We might consider creating a view that is responsible for representing the architecturally significant elements within an application. In fact, a view's intent is to model architecturally significant elements that are relevant to the perspective the view represents. In some situations, we may be interested in representing the architecturally significant elements of our system fulfilling a set of security requirements. In this case, we might create a security view. Architectural modeling is discussed in detail in Chapter 10. The point is that whenever we need to communicate about the system from a particular perspective, we can create a view into the system from that perspective, which is consistent with all other views we have created.

3.1 Core Diagrams

As we've seen, we can combine diagrams that form models and that can serve as views into our system. This capability is illustrated at a higher level in Figure 3.3. If an advantage in modeling is to combine diagrams to form views into our system, then it only makes sense that each diagram has a different focus on what it communicates.

Examining the intent of these diagrams, we see that each falls into one of two categories. Behavioral diagrams depict the dynamic aspects of our system. They are most useful for specifying the collaborations among elements that satisfy the behavior of our system's requirements. Structural diagrams depict the static aspect of our system. These diagrams are most useful for illustrating the relationships that exist among physical elements within our system, such as classes.

Of these diagrams, the three most commonly used are use case, sequence, and class diagrams. These three typically are used on all software development projects taking advantage of the UML. While still important, the remaining diagrams have a more specialized focus, which isn't to say that these remaining dia-



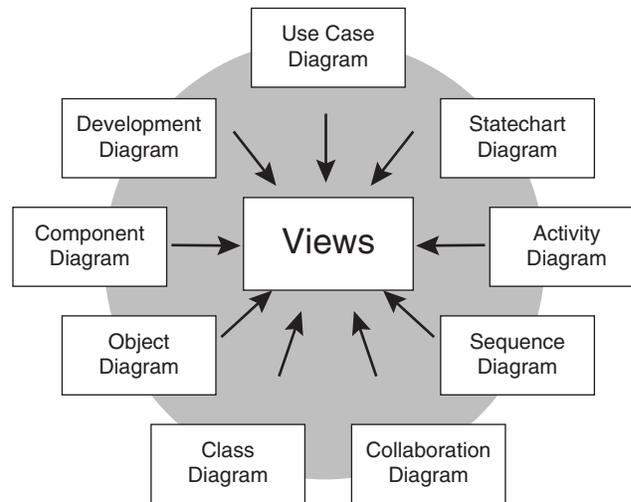
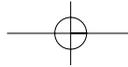


Figure 3.3 Diagrams Composing the UML

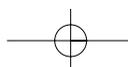
grams don't serve an important purpose. They definitely do—usually in cases where a specific, complex portion of our system must be communicated or more fully understood. Our discussion of using the UML with Java begins with a focus on these three diagrams.

3.1.1 Behavioral Diagrams

Behavioral diagrams communicate the aspects of the system that contribute to satisfying the system's requirements, typically captured in the form of use cases. Table 3.2 describes the five diagrams that fall into this category. Of these diagrams, the most commonly used are use case, sequence, and collaboration diagrams. While still useful, activity and state diagrams typically are used on an as-needed basis. Activity diagrams visually represent behaviors captured by use cases. State diagrams, on the other hand, are used to illustrate complex transitions in behavior for a single class.

Use case diagrams are centered around the business processes that our application must support. Most simply, use case diagrams enable us to structure our entire application around the core processes that it must support. Doing so enables us to use these use cases to drive the remainder of the modeling and development effort.

Sequence and collaboration diagrams are forms of interaction diagrams, which model the interactions among a set of objects. Interaction diagrams are



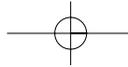
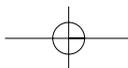
**Table 3.2 Behavioral Diagrams**

Diagram	Description
Use case	Shows a set of actors and use cases, and the relationships between them. Use case diagrams contribute to effective model organization, as well as modeling the core behaviors of a system.
Activity	Models the flow of activity between processes. These diagrams are most useful in detailing use case behavior. An activity diagram doesn't show collaboration among objects.
State	Illustrates internal state-related behavior of an object. Transitions between states help identify, and validate, complex behavior. A class can have at most a single state diagram.
Sequence	Semantically equivalent to a collaboration diagram, a sequence diagram is a type of interaction diagram that describes time ordering of messages sent between objects.
Collaboration	A type of interaction diagram that describes the organizational layout of the objects that send and receive messages. Semantically equivalent to a sequence diagram.

used often, primarily because they capture the messages sent between objects, which is of utmost importance to architects, designers, and developers.

Both sequence and collaboration diagrams fall into the interaction diagrams category because these diagrams are semantically equivalent—that is, they specify the same behaviors. The difference is the vantage point from which they express them. Collaboration diagrams focus on the spatial layout of the object interactions. They're useful in identifying structure among the classes, because the format of a collaboration diagram is similar in layout to that of a class diagram. Sequence diagrams, on the other hand, are focused on communicating the time ordering of messages sent among the objects. Because of their similar nature, it's quite common for a development team to standardize on the usage of either a sequence or collaboration diagram but not on both.



3.1.2 Structural Diagrams

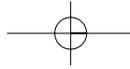
Diagrams in this category are focused on specifying the static aspects of our system. Table 3.3 describes the structural diagrams. Of these four diagrams, the class diagram is most often used. In fact, when transitioning to the UML, most organizations tend to use class diagrams first because they are excellent mechanisms for communication among developers, as well as tools that can be used for problem solving.

There are two forms of class diagrams. The first is the most commonly understood and consists of the classes that compose our system and of the structure among these classes. Unfortunately, the second is not often used but is of equal importance and can be most effective in helping developers understand our system from a high level. A type of class diagram, called a *package diagram*, often represents the Java packages and the dependencies between them that our application consists of. An example of a package diagram is provided in Figure 3.6.

The remaining structural diagrams, while still useful, have niches in modeling that are focused on certain types of applications. Without undermining the

Table 3.3 Structural Diagrams

Diagram	Description
Class	Illustrates a set of classes, packages, and relationships detailing a particular aspect of a system. This diagram is likely the most common one used in modeling.
Object	Provides a snapshot of the system illustrating the static relationships that exist between objects.
Component	Addresses the static relationships existing between the deployable software components. Examples of components may be .exe, .dll, .ocx, jar files, and/or Enterprise JavaBeans.
Deployment	Describes the physical topology of a system. Typically includes various processing nodes, realized in the form of a device (for example, a printer or modem) or a processor (for example, a server).



importance of these diagrams, suffice it to say that use of these diagrams should be dictated by the complexity of our system. Component diagrams might be used to show the software components within our application. Components aren't equivalent to classes. A component might be composed of multiple classes. Deployment diagrams, which illustrate the physical topology of our system, are most useful when we have a complex configuration environment. If our application is to be deployed to multiple servers, across locations, a deployment diagram might be useful. Object diagrams depict the structural relationship that exists among the objects within our running application at a given point in time. When we think of the runtime version of our system, we typically think of behavior. Many people have found that object diagrams are most useful in fleshing out the instance relationships among objects, which in turn can help verify our class diagrams. Beyond this, object diagrams are not often used.

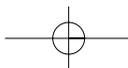
3.2 Fundamental Elements

Our discussion on the fundamental elements is in two sections. Section 3.2.1 discusses the structural elements that represent abstractions in our system. Structural elements typically are elements that encapsulate the system's set of behaviors. In section 3.2.2, we discuss the relationships that define how the structural elements relate to each other. Throughout our discussions of these elements, we provide mappings to the Java language.

3.2.1 Structural Elements

We have broken our discussion of structural elements into two sections. In section 3.2.2, we discuss Java-independent entities, which are elements that don't have a Java language mapping. Then, in section 3.2.3, we turn our attention to the Java-dependent entities, which are elements that have a straightforward Java mapping. The UML includes other structural elements that are beyond the scope of our discussion.

In our discussion of the structural elements, we use a template. At the top left is the name of the element. In the top center of the template is a graphic representing the element as it appears on a diagram. On the top right is the diagram(s) on which this element most often appears. Syntactically, placing these elements on diagrams not documented here might be correct. However, because we're using a simple approach, our discussion is focused on the diagram on which this element appears most often. In some cases, elements appear on more than one diagram, and in these special cases, the documenta-



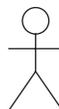
tion reflects that. The top right section of the template is broken down into the following categories:

- **Specific diagram:** This element can appear on any of the diagrams mentioned at the beginning of this chapter. Replace *specific* with the diagram name.
- **Structural diagram:** This element can appear on any of the structural diagrams, as categorized at the beginning of this chapter.
- **Behavioral diagram:** This element can appear on any of the behavioral diagrams, as categorized at the beginning of this chapter.
- **Combinatorial:** Any combination of the preceding can be used.

Following the top section of each element is a description of that element. In section 3.2.3, a table with two columns is included. The left column shows a Java code snippet. The right column shows the UML representation of this code.

3.2.2 Java-Independent Entities

Actor



Use Case Diagram

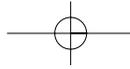
An actor represents a role that a user of the system plays. An actor always is external to the system under development. An actor need not always be a person. An actor might be another external system, such as a legacy mainframe from which we are obtaining data, or possibly a device, which we must obtain data from, such as a keypad on an ATM machine.

Use Case



Use Case Diagram

A use case represents a sequence of actions that a software system guarantees to carry out on behalf of an actor. When defining use cases, the level of granularity becomes important. The level of granularity varies among systems. The one constant is that a use case should always provide a result of an observable value to an actor. Primary business processes typically are good candidates for use cases. This way, these individual use cases can drive the development effort, which is focused on core business processes. This enables us to trace our results throughout the development lifecycle. A use case should be focused on the system from a customer's perspective.



Collaboration



Use Case Diagram

A collaboration is somewhat beyond the scope of our discussion in this book. However, because we use examples in later chapters, it should be introduced for completeness. Collaborations most often are used to bring structure to our design model. They enable us to create sequence and class diagrams that work in conjunction with each other, to provide an object-oriented view into the requirements that our system satisfies. A collaboration typically has a one-to-one mapping to a use case. This way, while use cases represent requirements from a customer's vantage point in the use case view, a collaboration models these same set of requirements from a developer's perspective.

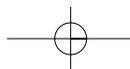
Object

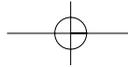


Interaction and Object Diagrams

An object is an instance of a class. It is represented by a rectangle. An object can be named in three different ways. First, and probably most common, we can specify the class that this object is an instance of. This is done by specifying the class name in the object rectangle. The class name is preceded by a semicolon and underlined. Second, we can specify only the object name, neglecting the class name, which is done by omitting the class name and semicolon and simply typing the object name and underlining it. In this naming scenario, we don't know the class that this object is an instance of. The third way to represent an object is to combine the two previously mentioned approaches.

An object can also be thought of as a physical entity, whereas a class is a conceptual entity. At first glance, it may seem odd that an object doesn't have a Java language mapping. In fact, it does. An object in the UML maps directly to an object in Java. However, when developers create Java applications, they are creating Java classes, not Java objects. Developers never write their code inside a Java object. Thinking about this a little differently, we think of objects as existing at runtime and classes as existing at design time. Developers create their code and map the UML elements to Java at design time, not runtime. Therefore, while a UML object maps directly to an object in Java, no Java language mapping represents a UML object.





3.2.3 Java-Dependent Entities

Class

Class Name
Attributes
Operations

Class Diagram

A class is a blueprint for an object. A class has three compartments. The first represents the name of the class as defined in Java. The second represents the attributes. Attributes correspond to instance variables within the class. An attribute defined in this second compartment is the same as a composition relationship. The third compartment represents methods on the class. Attributes and operations can be preceded with a visibility adornment. A plus sign (+) indicates public visibility, and a minus sign (-) denotes private visibility. A pound sign (#) denotes protected visibility. Omission of this visibility adornment denotes package-level visibility. When an attribute or operation is underlined, it indicates that it's static. An operation may also list the parameters it accepts, as well as the return type, as follows:

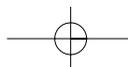
Java	UML
<pre>public class Employee { private int empID; public double calcSalary() { ... } }</pre>	

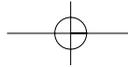
Package



Class Diagram

A general purpose grouping mechanism, packages can contain any other type of element. A package in the UML translates directly into a package in Java. In Java, a package can contain other packages, classes, or both. When modeling, we typically have packages that are logical, implying they serve only to organize our model. We also have packages that are physical, implying these packages translate directly into Java packages in our system. A package has a name that uniquely identifies it.





Java	UML
<pre>package BusinessObjects; public class Employee { }</pre>	

Interface



Class Diagram

An interface is a collection of operations that specify a service of a class. An interface translates directly to an interface type in Java. An interface can be represented either by the previously shown icon or by a regular class with a stereotype attachment of <<interface>>. An interface typically is shown on a class diagram as having realization relationships with other classes.

Java	UML
<pre>public interface CollegePerson { public Schedule getSchedule(); }</pre>	

3.2.4 Java-Dependent Relationships

The following examples illustrate the relationships in isolation based on the intent. Though syntactically correct, these samples could be further refined to include additional semantic meaning within the domain with which they are associated.

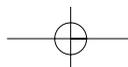
Each of the following relationships appear on diagrams in the structural category, most likely class diagrams. Though some, such as association, also appear on use case diagrams, their discussion is beyond the scope of this book.

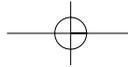
Dependency



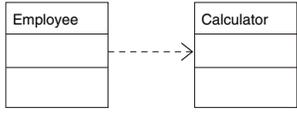
Structural Diagram

A “using” relationship between entities that implies a change in specification of one entity may affect the entities that are dependent upon it. More concretely, a





dependency translates to any type of reference to a class or object that doesn't exist at the instance scope, including a local variable, reference to an object obtained via a method call, as in the following example, or reference to a class' static method, where an instance of that class does not exist. A dependency also is used to represent the relationship between packages. Because a package contains classes, we can illustrate that various packages have relationships between them based upon the relationships among the classes within those packages.

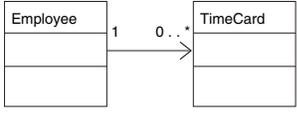
Java	UML
<pre>public class Employee { public void calcSalary(Calculator c) { ... } }</pre>	

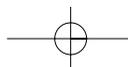
Association

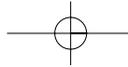


Structural & Use Case Diagrams

A structural relationship between entities specifying that objects are connected. The arrow is optional and specifies navigability. No arrow implies bidirectional navigability, resulting in tighter coupling. An instance of an association is a link, which is used on interaction diagrams to model messages sent between objects. In Java, an association translates to an instance scope variable, as in the following example. Additional adornments also can be attached to an association. Multiplicity adornments imply relationships between the instances. In the following example, an `Employee` can have 0 or more `TimeCard` objects. However, a `TimeCard` belongs to a single `Employee` (that is, `Employees` do not share `TimeCards`).

Java	UML
<pre>public class Employee { private TimeCard _tc[]; public void maintainTimeCard() { ... } }</pre>	



**Aggregation****Class Diagram**

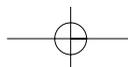
A form of association representing a whole/part relationship between two classes, an aggregation implies that the whole is at a conceptually higher level than the part, whereas an association implies both classes are at the same conceptual level. An aggregation translates to an instance scope variable in Java. The difference between an association and an aggregation is entirely conceptual and is focused strictly on semantics. An aggregation also implies that no cycles are in the instance graph. In other words, an aggregation must be a unidirectional relationship. The difference in Java between an association and aggregation is not noticeable. As stated previously, it's purely a matter of semantics. If you are unsure as to when to use an association or an aggregation, use an association. An aggregation need not be used often.

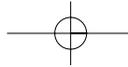
Java	UML
<pre>public class Employee { private EmpType et[]; public EmpType getEmpType() { ... } }</pre>	

Composition**Class Diagram**

A special form of aggregation, which implies lifetime responsibility of the part within the whole, composition is also nonshared. Therefore, while the part doesn't necessarily need to be destroyed when the whole is destroyed, the whole is responsible for either keeping alive or destroying the part. The part cannot be shared with other wholes. The whole, however, can transfer ownership to another object, which then assumes lifetime responsibility. The *UML Semantics* documentation states the following:

Composite aggregation is a strong form of aggregation which requires that a part instance be included in at most one composite at a time and that the composite object has sole responsibility for the disposition of its parts. [SEM01]





The relationship below between Employee and TimeCard might better be represented as a composition versus an association, as in the previous discussion.

Java	UML
<pre>public class Employee { private TimeCard tc[]; public void maintainTimeCard() { ... } }</pre>	

Generalization



Class and Use Case Diagrams

Illustrating a relationship between a more general element and a more specific element, a generalization is the UML element to model inheritance. In Java, a generalization directly translates into use of the extends keyword. A generalization also can be used to model relationships between actors and use cases.

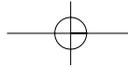
Java	UML
<pre>public abstract class Employee { } public class Professor extends Employee { }</pre>	

Realization



Class Diagram

A relationship that specifies a contract between two entities, in which one entity defines a contract that another entity guarantees to carry out. When modeling Java applications, a realization translates directly into the use of the implements keyword. A realization also can be used to obtain traceability between use cases, which define the behavior of the system, to the set of classes that guarantee to realize this behavior. This set of classes that realize a use case are typically structured around a collaboration, formally known as a *use case realization*.



Java	UML
<pre>public interface CollegePerson { } public class Professor implements CollegePerson { }</pre>	

3.3 Annotations

The only true annotational item in the UML is a note. Annotations simply provide further explanation on various aspects of UML elements and diagrams.

Notes



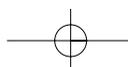
Any Diagram

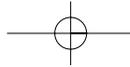
Notes in the UML are one of the least structured elements. They simply represent a comment about your model. Notes can be, though need not be, attached to any of the other elements and can be placed on any diagram. Attaching a note to another element is done via a simple dotted line. If the note is not attached to anything, we can omit the dotted line. The closest thing that notes translate to in Java is a comment. However, it isn't likely that we would copy the text from a note and place it in our Java code. Notes provide comments regarding our diagrams; comments describe code in detail.

Java	UML
<pre>/** Here is an example of a Java comment. Typically, this text will not be exactly the same as the text contained within the note. */</pre>	

3.4 Extensibility Mechanisms

The extensibility mechanisms don't necessarily have direct mappings to Java. However, they're still a critical element of the UML. These mechanisms are commonly used across diagrams, and understanding their intent is important.





We can create our own mechanisms, which enables us to customize the UML for our development environment. We should use caution in creating our own mechanisms. The UML is a robust language. Before defining our own extension mechanisms, we should be sure the mechanism does not already exist within the language.

Stereotype

A stereotype is used to create a new fundamental element within the UML with its own set of special properties, semantics, and notation. UML profiles can be created that define a set of stereotypes for language-specific features. For instance, Sun is currently working on a UML profile that defines a mapping between the UML and Enterprise JavaBeans (EJB).

Tagged Values

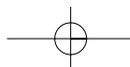
Tagged values enable us to extend the UML by creating properties that can be attached to other fundamental elements. For instance, a tagged value may be used to specify the author and version of a particular component. Tagged values also can be associated with stereotypes, at which point attachment of the stereotype to an element implies the tagged value.

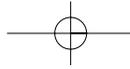
Constraint

Constraints enable us to modify or add various rules to the UML. Essentially, constraints enable us to add new semantics. For example, across a set of associations, we may use a constraint to specify that only a single instance is manifest at any point in time.

3.5 Introduction to Diagrams

These sample diagrams provide illustrations of how we interpret various relationships on individual diagrams. We also discuss how these different diagrams can be used in conjunction with each other to further enhance communication. Our example here depicts the Java event-handling mechanism used within the Abstract Windowing Toolkit (AWT). When developing Java applications, it's quite common to use a pattern similar to that of AWT to handle events within our application. In fact, this event-handling mechanism within Java is an implementation of the observer [GOF95] and command design pattern [GOF95].





3.5.1 Sequence Diagram

The rectangles at the top of our sequence diagram in Figure 3.4 denote objects. Objects have the same rectangular iconic representation as classes. There are three primary ways to name objects. The first, seen at the far left in Figure 3.4, is to provide an object name. When using this form, we don't necessarily know the name of the class that this object is an instance of. This representation, not seen in Figure 3.4, is commonly used to represent the fact that any object, regardless of its type, can invoke the flow of events that this sequence diagram is modeling. The second, represented by the remaining objects, denotes that an object is an instance of a specific class. This form is illustrated by preceding the name of the class with a colon. The third way of naming an object is to use a combination of the previously described two approaches. This way is not shown on Figure 3.4. Of the three ways to name objects on sequence diagrams, the second is most common. In each instance, the object's name is underlined.

A directed arrow represents messages on sequence diagrams. Associated with this directed arrow is typically a method name denoting the method that is

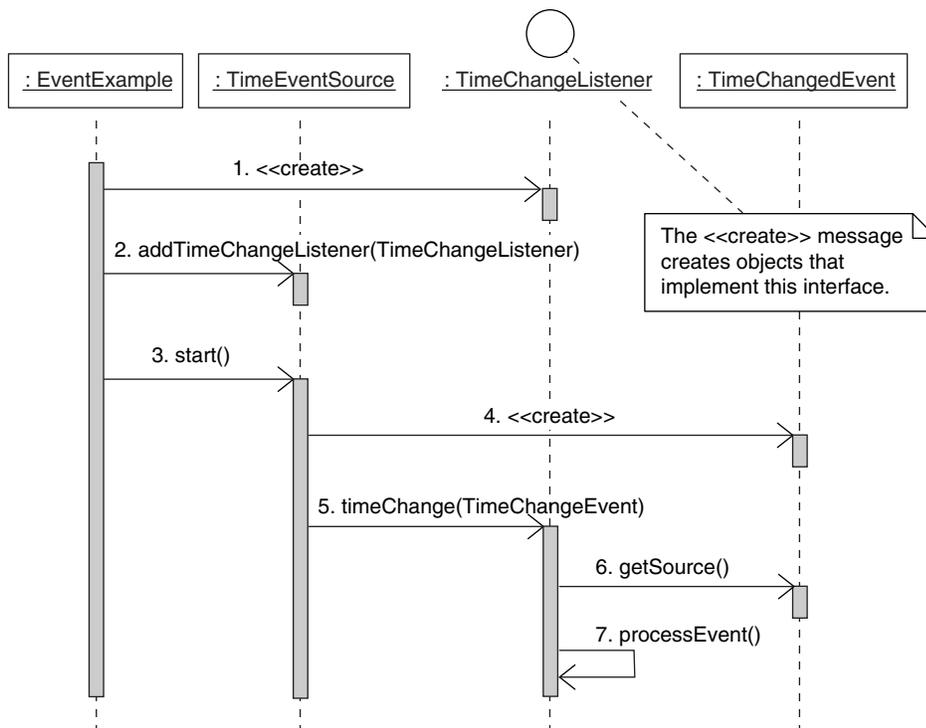
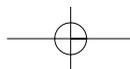
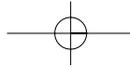


Figure 3.4 Sequence Diagram for Event-Handling Simulation



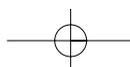


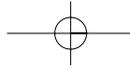
triggered as the result of the object's communication. In Figure 3.4, which simulates Java's AWT event-handling mechanism, my `EventExample` object sends a method to the `TimeEventSource` object. This message results in the `addTimeChangeListener()` method being triggered. The ordering of the messages sent between objects is always read top to bottom and typically is read left to right, although reading left to right is not a requirement. Be sure to notice the notes used in Figure 3.4 to enhance understanding.

Let's walk through the sequence diagram in further detail. The numbers that follow correspond to the numbers in Figure 3.4.

1. Some `EventExample`, which can be any object in our system, begins our event simulation by creating a `TimeChangeListener` object. Obviously, we can't have a true instance of an interface, and we certainly won't. However, it's important to communicate that the `TimeEventSource` isn't coupled to the implementation of the `TimeChangeListener` but to the listener itself. That is clearly communicated in Figure 3.4.
2. The `EventExample` object now registers the `TimeChangeListener` with the `TimeEventSource` object.
3. `EventExample` calls `start` on the `TimeEventSource`, which begins sending events to the listeners that have been registered with it.
4. The `TimeEventSource` creates a `TimeChangeEvent` object. This `TimeChangeEvent` object encapsulates information regarding this event.
5. The `TimeEventSource` loops through each of its listeners, calling the `timeChange` method on each.
6. Optionally, the `TimeChangeListener` can obtain a reference to the object that caused the event notification. This reference is returned a generic reference to `java.lang.Object`.
7. `TimeChangeListener` calls its `processEvent()` method to handle processing the event.

In Figure 3.4, notice that we have attached a note to the `TimeChangeListener` interface specifying that we will actually create a class that implements this interface. This note allows for a great deal of flexibility in our diagram, because the entire message sequence holds true regardless of what class we use in place of `TimeChangeListener`. We also may wish to use a note to specify that when the `TimeEventSource` object notifies its listeners of a time change; it may notify multiple listeners, resulting in a loop. The point is that developers who need to interpret this diagram, and use it to construct code, must have the information provided in the notes to effectively construct code. Notes are a great aid





in helping us understand more fully the details associated with this sequence of events. Notes appear on most diagrams and should be used often.

We typically have many sequence diagrams for a single class diagram because any society of classes most likely interacts in many different ways. The intent of a sequence diagram is to model one way in which the society interacts. Therefore, to represent multiple interactions, we have many sequence diagrams.

Sequence diagrams, when used in conjunction with class diagrams, are an extremely effective communication mechanism because we can use a class diagram to illustrate the relationships between the classes. The sequence diagram then can be used to show the messages sent among the instances of these classes, as well as the order in which they are sent, which ultimately contribute to the relationships on a class diagram. When an object sends a message to another object, that essentially implies that the two classes have a relationship that must be shown on a class diagram.

3.5.2 Class Diagram

The class diagram in Figure 3.5 is a structural representation of the Java AWT event simulation. Note each of the relationships that appear on this diagram. First, our `EventExample` class has relationships to `TimeEventSource` and `TimePrinter`, which corresponds to the messages an `EventExample` object must send to instances of these classes. On our sequence diagram in Figure 3.4, however, we didn't see a `TimePrinter` object. In fact, the sequence diagram did contain a `TimePrinter`, but it was in the form of the `TimeChangeListener` interface, which the note on the diagram in Figure 3.4 didn't clarify. As we can see in Figure 3.5, the `TimePrinter` implements the `TimeChangeListener` interface. Also, notice the structural inheritance relationships depicted on this diagram that weren't apparent in Figure 3.4.

When interpreting the diagrams, it often is easiest to place the class diagram and sequence diagram side by side. Begin reading the sequence diagram, and as the messages are sent between the objects, trace these messages back to the relationships on the class diagram. Doing so should be helpful in understanding why the relationships on the class diagram exist. After having read each of the sequence diagrams, if you find a relationship on a class diagram that can't be mapped to a method call, you should question the validity of the relationship.

3.5.3 Package Diagram

Package diagrams are really a form of a class diagram. The difference is that a package diagram shows the relationships between our individual packages. We can think of a package diagram as a higher-level view into our system. This

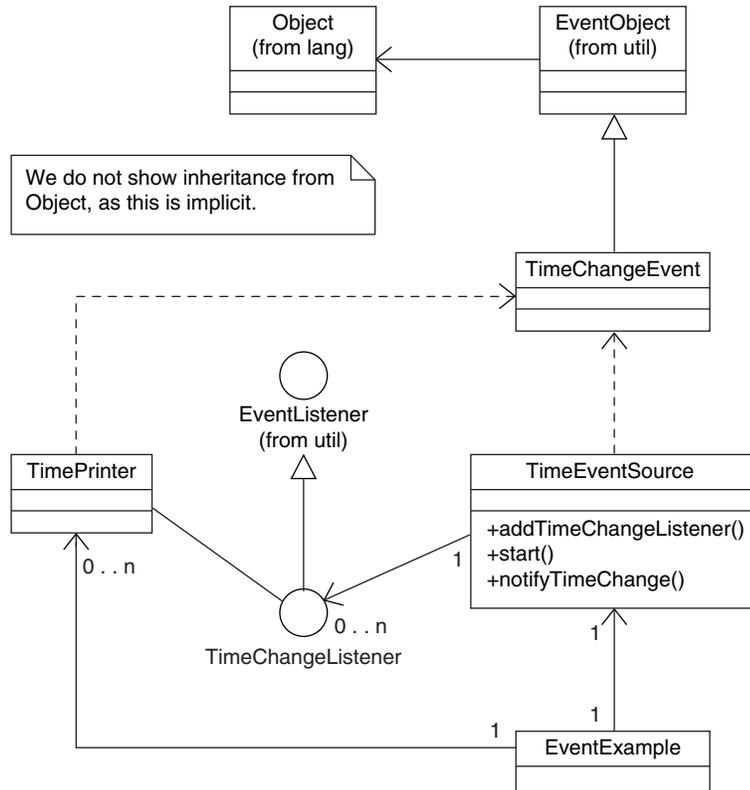
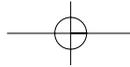
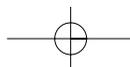


Figure 3.5 Class Diagram for Event-Handling Simulation

becomes important when understanding the system’s architecture. The dependency relationships illustrated in Figure 3.6 provide an indication of the direction of the structural relationships among the classes within the various packages. Notice that the `eventhandling` package is dependent on the `util` package, which implies that classes within `eventhandling` can import classes within `util` but not vice versa. This distinction is an important one because our package dependencies must be consistent with the relationships expressed on the corresponding class diagrams.

Package diagrams, when combined with class diagrams, are an extremely effective mechanism to communicate a system’s architecture. A diagram such as the one in Figure 3.6 provides a high-level glimpse into a system’s overall structure. Based on this high-level view, developers can make assumptions regarding the relationships between individual classes, which becomes especially helpful as new developers join the project and need to be quickly brought up to speed—



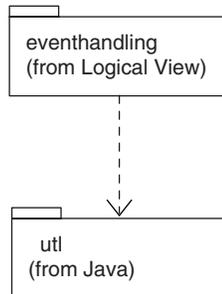
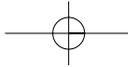


Figure 3.6 Package Diagram for Event-Handling Simulation

or when developers need to maintain a system they may have previously worked on but haven't interacted with in some time. Either way, this form of "architectural modeling" is beneficial. The code for this example can be found in Appendix C.

3.6 Conclusion

As we've seen, many of the elements that compose the UML have a precise mapping to the Java programming language, which is consistent with the claims that the UML is a precise and unambiguous communication mechanism. As individuals associated with software development, we must interpret each of these elements in a fashion that is faithful to this claim. Different interpretations of these elements can result in miscommunication, which is one of the challenges the UML attempts to resolve.

Though the UML is precise and unambiguous, we also must make sure that we don't derive more meaning than our diagrams actually convey. The UML is not a visual programming language but a modeling language. As such, the intent is not to model at the same level of detail at which code exists. For instance, an association has a multiplicity adornment associated with it at each end. If at one end the multiplicity is $0..*$, we must interpret this as an optional collection. Therefore, we must accommodate for the potential of a null object reference in our code. The way in which we accommodate this is typically left up to the implementor. It may be implemented as an array, a vector, or a custom collection class. The diagram typically doesn't state how to implement something, communicating instead only that we must accommodate that need.