

Unified Modeling Language

Modeling Java Applications using UML Language Mappings



Programmer's Reference Manual

How to use this Reference Card: The UML consists of a set of fundamental modeling elements which appear on many different diagrams. In this manual, we first provide a brief overview of the diagrams which compose the UML, and the category into which each fall. After this, we provide an illustration of the more common fundamental elements, providing a description of each, and where applicable, how they can be mapped to the Java programming language. These fundamental elements are broken out into two categories: those which represent some abstraction in our model, and those which specify the relationships that exist between our abstractions. The first we call entities, and the second we call relationships. This reference manual does not illustrate all of the elements which compose the UML. It focuses only on the more commonly used. Of the diagrams discussed, the Class and Interaction diagrams are used most often when modeling our systems, and this is the focus of our discussion. For a more detailed discussion on these fundamental elements, or individual diagrams, please refer to the UML User Guide.

UML and Java: Because the UML is a language, just as Java is a language, there is a mapping that must take place when converting our UML models to Java applications. Because UML is a precise and unambiguous language, little is left open for interpretation. Elements on diagrams can be accurately represented in Java. This card documents these mappings, helping to make our transition to the world of modeling more seamless. This will benefit us not only in interpreting our models in terms of Java code, but also in creating our own models. Throughout our discussion, we will elaborate further on how some of the fundamental elements are represented in Java.

Diagrams		
Diagram Name	Category	Description
Class	Structural	Illustrates a set of classes, packages, and relationships detailing a particular aspect of a system. This is likely the most common diagram used on modeling.
Object	Structural	A snapshot of the system illustrating the static relationships which exist between objects.
Component	Structural	Addresses the static relationships existing between the deployable software components. Examples of components may be .exe, .dll, .ocx, and/or beans.
Deployment	Structural	Describes the physical topology of a system. Typically include various processing nodes, realized in the form of a device (printer, modem) or a processor (server).
Composite Structure	Structural	Explore the run-time of interconnected instances collaborating over communication links.
Package	Structural	Enable organization of modeling elements into groups and represent relationships between those groups.
Use Case	Behavioral	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	Behavioral	Model the flow of activity between processes. Most useful in detailing Use Case behavior. Do not show collaboration amongst objects.
State	Behavioral	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	Behavioral	A type of Interaction diagram which describes time-ordering of messages sent between objects. Semantically equivalent to a Collaboration diagram.
Communication	Behavioral	A type of interaction diagram which describes the organizational layout of the objects which send and receive messages. Semantically equivalent to a Sequence diagram. Formerly Collaboration diagrams in UML 1.x.
Timing	Behavioral	Explore the behavior of one or more objects throughout a given period of time.
Interaction	Behavioral	Variants of Activity diagrams which overview control



While a diagram has a specific purpose, as identified by the description in the above table, the category into which a diagram falls is of utmost importance. Typically, when we model, diagrams will be used in conjunction with each other to create different ways of looking at the same system. These are commonly called views. Creating different views of a system to satisfy different individuals associated with a software project is typically a good thing to do. For instance, using both Class and Sequence diagrams together, provides the ability to not only show the structural relationships that exist amongst classes, but also to illustrate the dynamics associated with the messages sent between the instances of those classes.

Structural Entities

The following items represent structural entities. For our discussion, we have broken this out into two sections. The first section discusses elements which do not have a Java language mapping. These we will call Java Independent entities. The second section are those elements that do have a straightforward Java mapping, and are referred to as Java Dependent entities. There are other structural elements which compose the UML, but which are beyond the scope of our discussion.

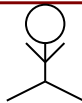
In our discussion, we will use the same template for each of the fundamental elements. At the top left will be the name of the element. This will be followed by a picture representing what that element would look like on a diagram. Farthest to the right on the first row will be the diagram(s) on which this element most often appears. Syntactically, placing these elements on diagrams not documented here might be correct. However, since our approach is one of simplicity, our discussion will be focused upon the diagram on which this element appears most often. In some cases, elements will consistently appear on more than one diagram, and in these special cases, the documentation will reflect that. This column 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 above can be used.

Following this will be a description of the element which we are discussing. Then, a table can be found with two columns. The first column shows a Java code snippet. The second column shows the UML representation of this code. This table will be shown in the second section only.

Java Independent Entities

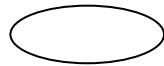
Actor



Use Case Diagram

An Actor represents a role a user of the system plays. An Actor is always 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. This will vary among systems. The one constant is that a Use Case should always provide a result of an observable value to an actor. Typically, primary business processes are good candidates for Use Cases. This way, these individual Use Cases can drive the development effort, which will be focused on core business processes. This will allow us to trace our results throughout the development lifecycle. A Use Case should be focused on the system from a customer's perspective.

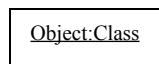
Collaboration



Use Case Diagram

A Collaboration is somewhat beyond the scope of our discussion here. However, because we use examples in later chapters, it needs to be introduced here for completeness. Collaborations are most often used to bring structure to our design model. They allow 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. Typically, a Collaboration will have 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



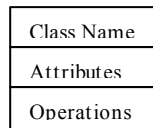
Interaction & Object Diagrams

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 which this Object is an instance of. This is done by specifying the class name in the Object rectangle, preceded by a semicolon, and underlining it. Second, we can specify only the Object name, neglecting the class name. This is done by omitting the class name and semicolon, and simply typing the Object name, then underlining it. In this naming scenario, we do not know the class which this Object is an instance of. We will see an example of this later in the chapter. The third way to represent an Object is to combine the two approaches mentioned above.

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 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 run-time, and classes existing at design time. Developers create their code, and map UML elements to Java, at design time, not run-time. Therefore, while a UML Object maps directly to an Object in Java, there is no Java language mapping representing a UML Object.

Java Dependent Entities

Class

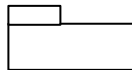


Class Diagram

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. A minus sign (-) denotes private visibility. A pound sign (#) denotes protected visibility. Omission of this visibility adornment denotes package level visibility. If an attribute or operation is underlined, this indicates that it is static. An operation may also list the parameters it accepts, as well as the return type, as seen below.

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

Package



Class Diagram

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

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

Interface



Class Diagram

An Interface is a collection of operations that specify a service of a Class. This translates directly to an interface type in Java. An Interface can be either represented by the above icon, or by a regular class with a Stereotype attachment of <<interface>>. Typically, an Interface will be shown on a class diagram as having Realization relationships with other classes.

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

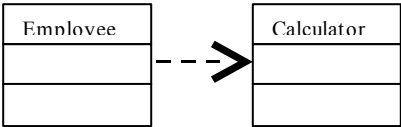
Relationships

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

Each of these relationships below appear on diagrams in the structural category, most likely class diagrams. Though some, such as Association, also appear on Use Case Diagrams. This is beyond the scope of our discussion here.

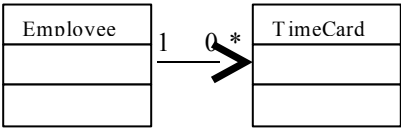
Dependency Structural Diagram

A “using” relationship between entities which implies a change in specification of one entity may affect the entities which are dependent upon it. More concretely, this translates to any type of reference to a class or object that does not exist at the instance scope. This includes a local variable, reference to an object obtained via a method call, as in the example below, or reference to a class's static method, where an instance of that class does not exist. A dependency is also 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 amongst the classes within those packages.

Java	UML
<pre>public class Employee { public void calcSalary(CalculatorStrategy cs) { ... } }</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 bi-directional 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 example below. Additional adornments can also be attached to an association. Multiplicity adornments imply relationships between the instances. In the example below, an Employee can have 0 or more TimeCard objects. However, a TimeCard belongs to a single Employee (i.e. Employees do not share TimeCards).

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

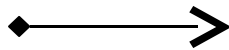
Aggregation Class Diagram

A form of Association representing a “whole/part” relationship between two classes. 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. Translates to an instance scope variable in Java. Difference between Association

and Aggregation is entirely conceptual, and is focused strictly on semantics. An aggregation also implies that there be no cycles in the instance graph. In other words, it must be a unidirectional relationship. The difference in Java between an Association and Aggregation is not noticeable. It is purely semantics. If you are unsure as to when to use Association or Aggregation, use Association. Aggregation need not be used often.

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

Composition



Class Diagram

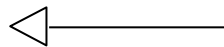
A special form of Aggregation, which implies lifetime responsibility of the part within the whole. Composition is also non-shared. Therefore, while the part does not necessarily need to be destroyed when the whole is destroyed, the whole is responsible for either keeping alive or destroying the part. The part cannot be shared with other wholes. The whole, however, can transfer ownership to another object, which then assumes lifetime responsibility. The UML 1.3 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, although the owner may be changed over time."

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

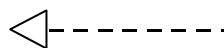


Class & Use Case Diagrams

Illustrates a relationship between a more general element and a more specific element. Generalization is the UML element to model Inheritance. In Java, this directly translates into use of the extends keyword. Generalization can also 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 which 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. Realization can also be used to obtain traceability between Use Cases, which define the behavior of the system, to the set of classes which guarantee to realize this behavior. These set of classes which realize a Use Case are typically structured around a Collaboration, formerly know as a Use Case Realization.

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

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.

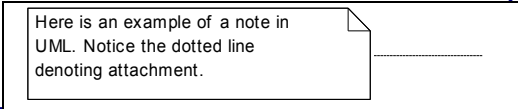
Notes



Any Diagram

Notes in 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 would translate to in Java would be a comment. However, it is not 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 in more detail our code.

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>	 <p>Here is an example of a note in UML. Notice the dotted line denoting attachment.</p>

Extensibility Mechanisms

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

We also have the ability to create our own mechanisms, which allow us to customize the UML for our development environment. We should use caution in doing this. The UML is a robust language. Before defining our own extension mechanisms, we should be sure it does not already exist within the language.

Stereotype

`<<stereotypename>>`

Used to create a new fundamental element within the UML with it's own set of special properties, semantics, and notation. UML Profiles can be created which define a set of stereotypes for language specific features. For instance, Sun is currently working on a UML Profile which defines a mapping between UML and EJB.

Tagged Values

`{tagged value = value}`

Tagged values allow us to extend the UML by create properties which 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 can also be associated with Stereotypes, at which point, attachment of the Stereotype to an element implies the Tagged Value.

Constraint

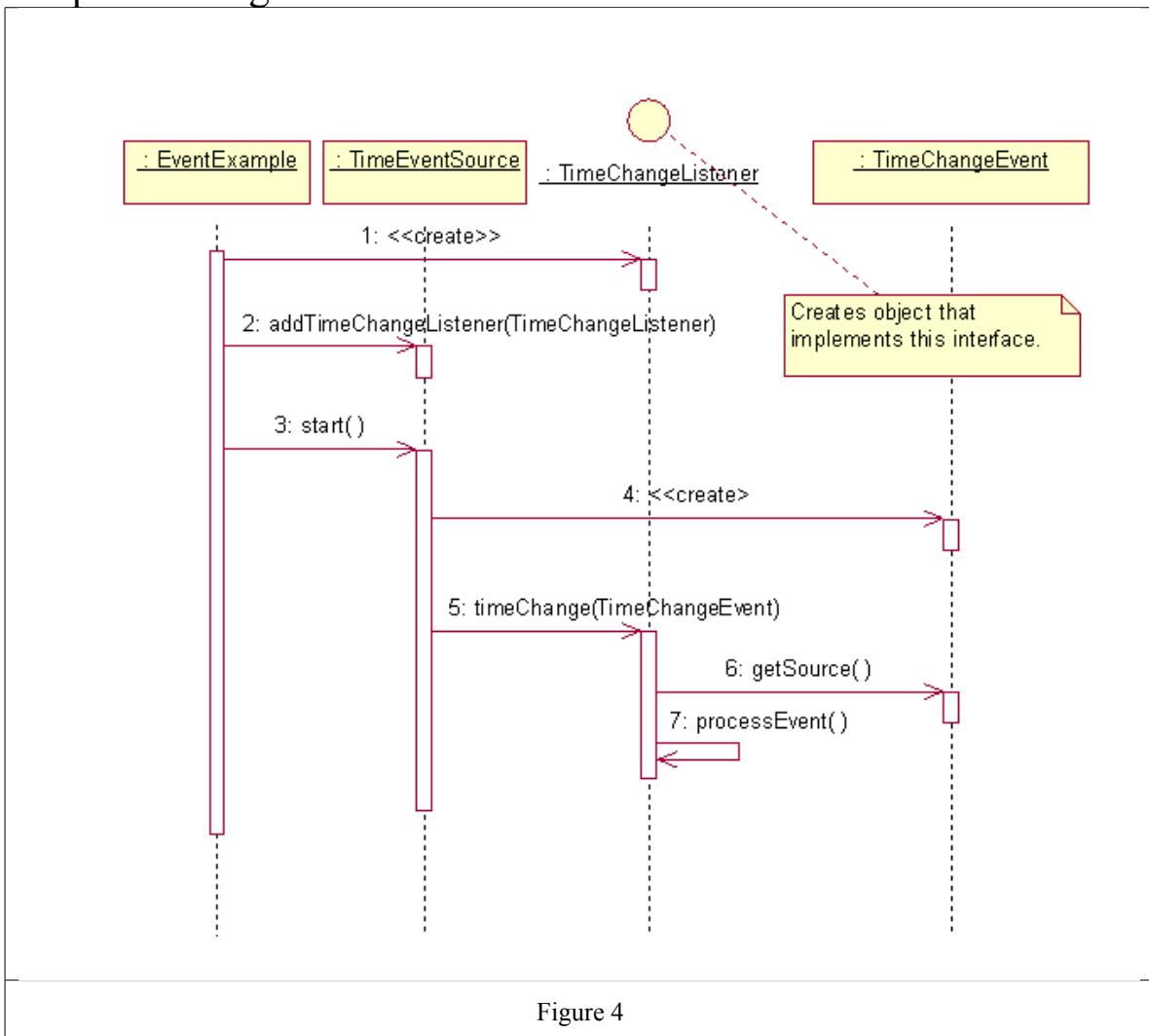
`{constraint}`

Constraints allow us to modify or add various rules to the UML. Essentially, Constraints allow 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.

Diagram Samples

These sample diagrams provide illustration on how we interpret various relationships on individual diagrams. As our discussion permits, we will 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 AWT. It is quite common when developing Java applications to use a pattern similar to this to handle events within our application. In fact, this event handling mechanism within Java is an implementation of the Observer and Command design pattern.

Sequence Diagram



The rectangles at the top of our sequence diagram in Figure 4 denote objects. Objects have the same rectangular iconic representation as Classes. There are three primary ways to name objects. The first, as seen above (farthest left object), is to provide an object name. Here, we do not necessarily know the name of the class which this object is an instance of. This 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 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 above two approaches. This is not shown on this diagram. 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 which is triggered as the result of the object's communication. On the above diagram, which simulates Java's AWT event handling mechanism, I can see that 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 is typically read left to right, though reading left to right is not a requirement. Also, be sure to notice the notes used on the above diagram to enhance the understanding.

Now, let's walk through the sequence diagram, in further detail.

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

Also, in the above diagram, we notice that when the `TimeEventSource` object notifies its listeners of a time change, that we have attached a note to the message indicating that it may perform this multiple times. This aids in helping us understand more fully the details associated with this sequence of events. Notes appear on most diagrams, and should be used often.

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

Sequence diagrams, when used in conjunction with class diagrams, are an extremely effective communication mechanism. This is because we can use a class diagram to illustrate the relationships between the classes. The sequence diagram can then be used to show the messages sent amongst 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. Essentially, if an object sends a message to another object, that must imply that the two classes have a relationship to each other which must be shown on a class diagram.

Class Diagram

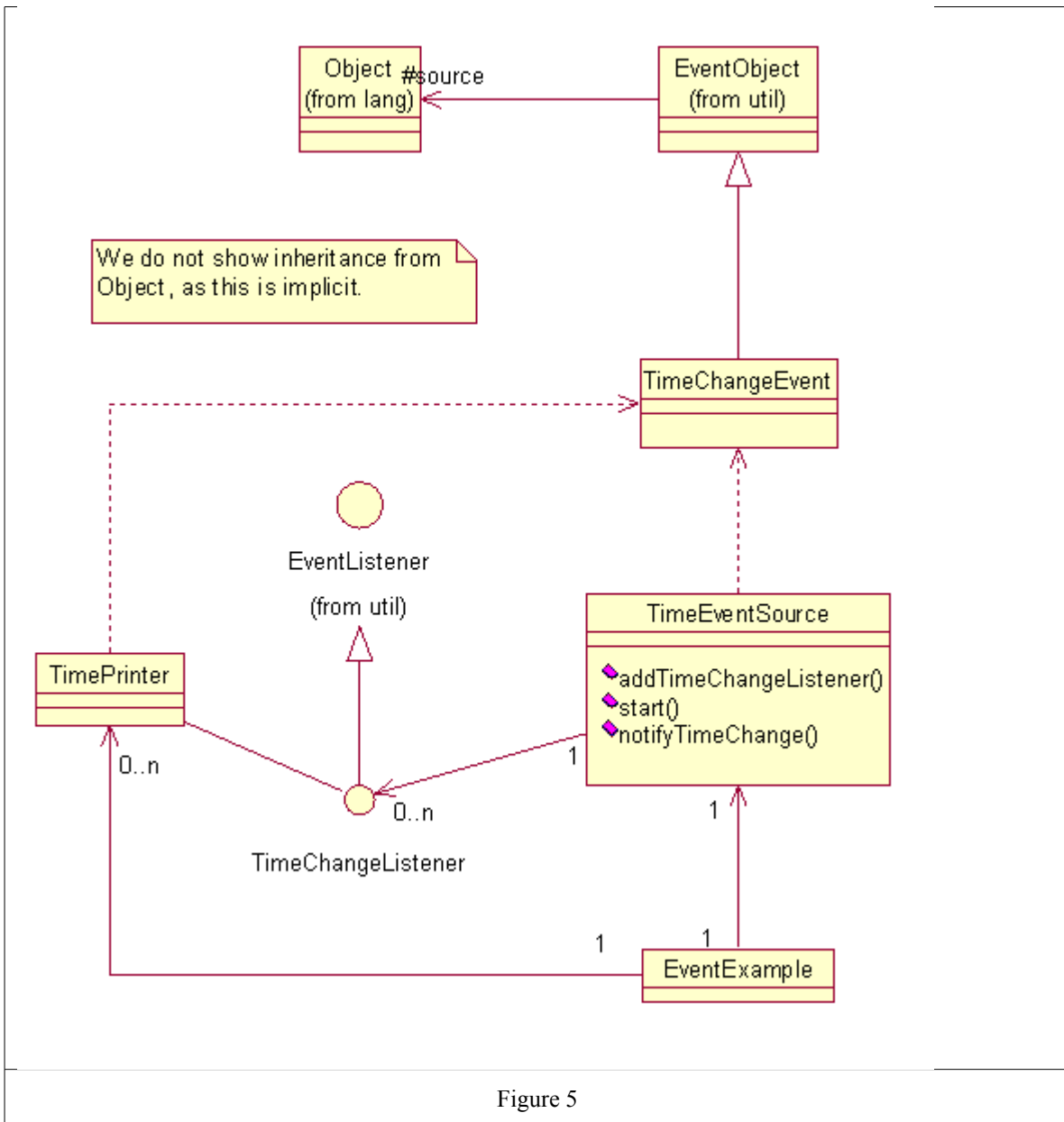


Figure 5

The class diagram in Figure 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. This corresponds to the messages an EventExample object would need to send to instances of these classes. On our sequence diagram, however, we did not see a TimePrinter object. In fact, the sequence diagram did contain a TimePrinter, but it was in the form of the TimeChangeListener interface. As we can see on the above diagram, the TimePrinter implements the TimeChangeListener interface. Also, notice the structural inheritance relationships depicted on this diagram which were not apparent on the Sequence diagram.

Many times, when interpreting each of the diagrams, it is easiest to place both 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. This should be very helpful in understanding why the relationships on the class diagram exist. If you find that, after having read each of the sequence diagrams, you find a relationship on a class diagram that can't be mapped to a method call, the relationship should be questioned as to its validity.

Package Diagram

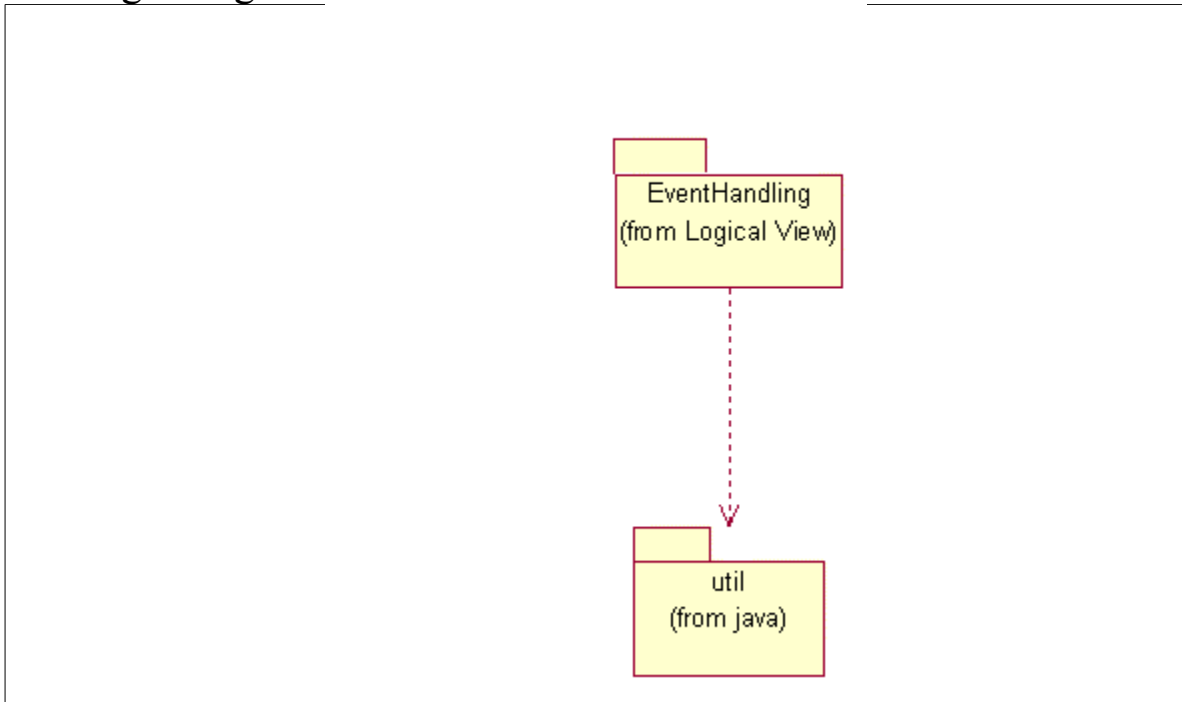


Figure 6

Package diagrams are a structural diagram showing the relationships between our individual packages. We can think of a package diagram as a higher level view into our system. This becomes important when understanding the system's architecture. The dependency relationships illustrated above give me an indication of the direction of the structural relationships amongst the classes within the various packages. Notice that the EventHandling package is dependent on the util package. This implies that classes within EventHandling can import classes within util, but not vice versa. This is an important distinction, as 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 above provides a high level glimpse into a system's overall structure. Based on this high level view, developers have the ability to make assumptions regarding the relationships between individual classes. This becomes especially helpful as new developers join the project and need to be brought up to speed quickly. Or when developers need to maintain a system they may have previously worked on, but have not interacted with, in quite some time. Either way, this form of "architectural modeling" is very beneficial.