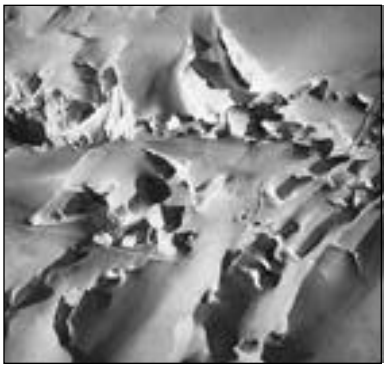
7



Object Design

Pooh: *Didn't you promise Christopher Robin not to touch the train?*
Tigger: *Yes, I did but I am only touching the controls.*
Pooh: *But aren't the controls part of the train?*
Tigger: *No, they don't even know each other.*
Pooh: *Well, isn't it time to introduce them to each other?*
—Winnie-the-Pooh

During analysis, we describe the purpose of the system. This results in the identification of application objects that represent user concepts. During system design, we describe the system in terms of its architecture, such as its subsystem decomposition, its global control flow, and its persistency management. During system design, we also define the hardware/software platform on which we build the system. This results in the selection of off-the-shelf components that provide a higher level of abstraction than the hardware. During object design, we close the gap between the application objects and the off-the-shelf components by identifying additional solution objects and refining existing objects.

Object design includes:

- *service specification*, during which we precisely describe each class interface
- *component selection*, during which we identify additional off-the-shelf components and solution objects
- *object model restructuring*, during which we transform the object design model to improve its understandability and extensibility
- *object model optimization*, during which we transform the object design model to address performance criteria such as response time or memory utilization

Object design, like system design, is not algorithmic. In this chapter, we show how to apply existing patterns and concrete components in the problem-solving process. We discuss these building blocks and the activities related to them. We conclude with the discussion of management issues associated with object design. We use Java and Java-based technologies in this chapter. The techniques we describe, however, are also applicable to other languages.

7.1 Introduction: Movie examples

Consider the following examples:

***Speed* (1994)**

Harry, an LAPD cop, is taken hostage by Howard, a mad bomber. Jack, Harry's partner, shoots Harry in the leg to slow down Howard's advance. Harry is shot in the right leg. Throughout the movie, Harry limps on the left leg.

***Star Wars* Trilogy (1977, 1980, & 1983)**

At the end of episode V: *The Empire Strikes Back* (1980), Han Solo is captured and frozen into carbonite for delivery to Jaba. At the beginning of episode VI, *The Return of the Jedi* (1983), the frozen Han Solo is recovered by his friends and thawed back to life. When being frozen, Solo is wearing a jacket. When thawed, he is wearing a white shirt.

***Titanic* (1997)**

Jack, a drifter, is teaching Rose, a lady from the high society, to spit. He demonstrates by example and encourages Rose to practice as well. During the lesson, Rose's mother arrives impromptu. As Jack starts to turn to face Rose's mother, there is no spit on his face. As he completes his turn, he has spit on his chin.

The budgets for *Speed*, *The Empire Strikes Back*, *The Return of the Jedi*, and *Titanic* were 30, 18, 32.5, and 200 millions dollars, respectively.

Movies, like software systems, are complex systems that contain (often many) bugs when delivered to the client. It is surprising, considering their cost of production, that any obvious mistakes should remain in the final product. Movies, however, are like software systems: They are more complex than they seem.

Many factors conspire to introduce mistakes in a movie: Movies require the cooperation of many different people; scenes are shot out of sequence; some scenes are reshot out of schedule; details, such as props and costumes, are changed during production; the pressure of the release date is high during the editing process, when all the pieces are integrated together. When a scene is shot, the state of every object and actor in the scene needs to be consistent with the scenes preceding and following it. This can include the pose of each actor, the condition of his or her clothes, jewelry, makeup, and hair, if they are drinking, the content of their glasses (e.g., white wine vs. red wine), the level of their glasses (e.g., full or half full), and so on. When different segments are combined into a single scenes, an editor, called the continuity editor, needs to ensure that such details were restored correctly. When changes occur, such as the addition or removal of a prop, the change must not interfere with other scenes.

Software systems, like movies, are complex, subject to continuous change, integrated under time pressure, and developed nonlinearly. During object design, developers close the gap between the application objects identified during analysis and the hardware/software platform selected during system design. Developers identify and build custom solution objects whose purpose is to realize any remaining functionality and to bridge the gap between application objects and the selected hardware/software platform. During object design, developers realize

custom objects in a way similar to the shooting of movie scenes. They are implemented out of sequence, by different developers, and change several times before they reach their final form. Often, the caller of an operation has only an informal specification of the operation and makes assumptions about its side effects and its boundary cases. This results in mismatches between caller and callee, missing behavior, or incorrect behavior. To address these issues, developers construct precise specifications of the classes, attributes, and operations in terms of constraints. Similarly, developers adjust and reuse off-the-shelf components, annotated with interface specifications. Finally, developers restructure and optimize the object design model to address design goals, such as maintainability, extensibility, efficiency, response time, or timely delivery.

In Section 7.2, the next section, we provide an overview of the object design. In Section 7.3, we define the main object design concepts, such as constraints used to specify interfaces. In Section 7.4, we describe in more detail the activities of object design. In Section 7.5, we discuss management issues related with object design. We do not describe activities such as implementing algorithms and data structures or using specific programming languages. First, we assume the reader already has experience in those areas. Second, these activities become less critical as more and more off-the-shelf components become available and reused.

7.2 An overview of object design

Conceptually, we think of system development as filling a gap between the problem and the machine. The activities of system development incrementally close this gap by identifying and defining objects that realize part of the system (Figure 7-1).

Analysis reduces the gap between the problem and the machine by identifying objects representing user-visible concepts. During analysis, we describe the system in terms of external behavior, such as its functionality (use case model), the application domain concepts it manipulates (object model), its behavior in terms of interactions (dynamic model), and its nonfunctional requirements.

System design reduces the gap between the problem and the machine by defining a hardware/software platform that provides a higher level of abstraction than the computer hardware. This is done by selecting off-the-shelf components for realizing standard services, such as middleware, user interface toolkits, application frameworks, and class libraries.

During object design, we refine the analysis and system design models, identify new objects, and close the gap between the application objects and off-the-shelf components. This includes the identification of custom objects, the adjustment of off-the-shelf components, and the precise specification of each subsystem interface and class. As a result, the object design model can be partitioned into sets of classes such that they can be implemented by individual developers.

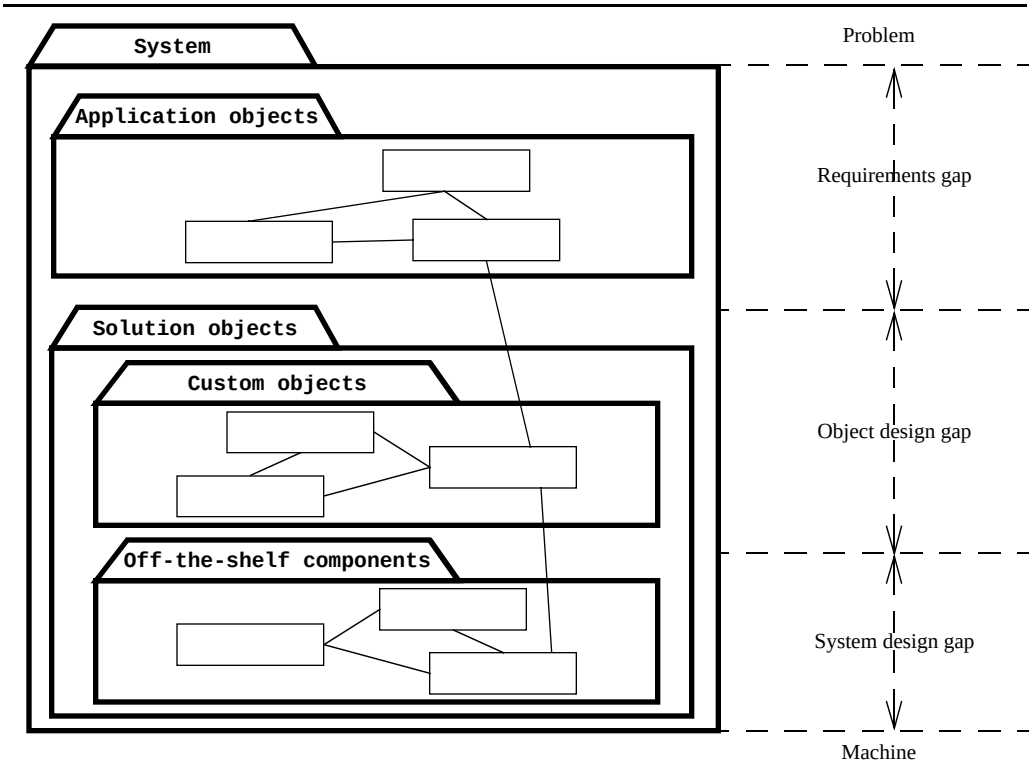


Figure 7-1 Object design closes the gap between application objects identified during requirements and off-the-shelf components selected during system design (stylized UML class diagram).

Object design includes four groups of activities (see Figure 7-2):

- *Service specification.* During object design, we specify the subsystem services (identified during system design) in terms of class interfaces, including operations, arguments, type signatures, and exceptions. During this activity, we also find missing operations and objects needed to transfer data among subsystems. The result of service specification is a complete interface specification for each subsystem. The subsystem service specification is often called subsystem **API** (Application Programmer Interface).
- *Component selection.* During object design, we use and adapt the off-the-shelf components identified during system design to realize each subsystem. We select class libraries and additional components for basic data structures and services. Often, we need to adjust the components we selected before we can use them, by wrapping custom objects around them or by refining them using inheritance. During these activities, we face the same buy versus build trade-off that we faced during system design.

- *Restructuring.* Restructuring activities manipulate the system model to increase code reuse or meet other design goals. Each restructuring activity can be seen as a graph transformation on subsets of a particular model. Typical activities include transforming n-ary associations into binary associations, implementing binary associations as references, merging two similar classes from two different subsystems into a single class, collapsing classes with no significant behavior into attributes, splitting complex classes into simpler ones, rearranging classes and operations to increase the inheritance and packaging. During restructuring, we address design goals such as maintainability, readability, and understandability of the system model.
- *Optimization.* Optimization activities address performance requirements of the system model. This includes changing algorithms to respond to speed or memory requirements, reducing multiplicities in associations to speed up queries, adding redundant associations for efficiency, rearranging execution orders, adding derived attributes to improve the access time to objects and opening up the architecture, that is, adding access to lower layers because of performance requirements.

Object design is nonlinear. Although the groups of activities we describe above each addresses a specific object design issue, they need to occur concurrently. A specific off-the-shelf component may constrain the number of types of exceptions mentioned in the specification of an operation and thus may impact the subsystem interface. The selection of a component may reduce the implementation work while introducing new "glue" objects, which also need to be specified. Finally, restructuring and optimizing may reduce the number of objects to be implemented by increasing the amount of reuse in the system.

The larger number of objects and developers, the high rate of change, and the concurrent number of decisions made during object design make this activity much more complex than analysis or system design is. This represents a management challenge, as many important decisions tend to be resolved independently and are not communicated to the rest of the project. Object design requires much information to be made available among the developers so that decisions can be made consistently with decisions made by other developers and with design goals. The *Object Design Document*, a live document describing the specification of each class, supports this information exchange.

7.3 Object design concepts

In this section, we present the principal object design concepts:

- application objects versus solution objects (Section 7.3.1)
- types, signatures, and visibility (Section 7.3.2)
- preconditions, postconditions, and invariants (Section 7.3.3)
- UML's Object Constraint Language (Section 7.3.4)

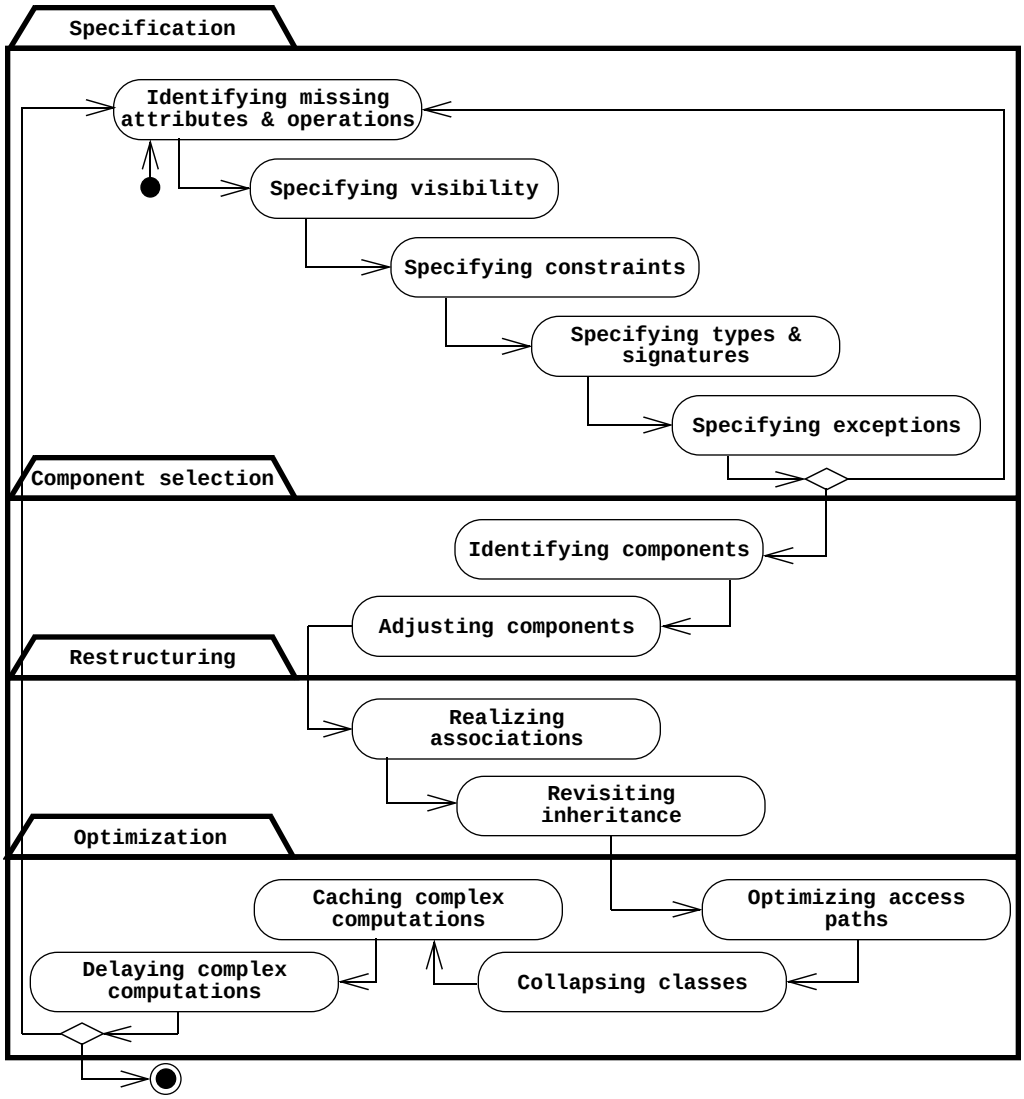


Figure 7-2 Activities of object design (UML activity diagram).

7.3.1 Application objects versus solution objects revisited

As we saw in Chapter 2, *Modeling with UML*, class diagrams can be used to model both the application domain and the solution domain. **Application objects**, also called domain objects, represent concepts of the domain that the system manipulates. **Solution objects** represent support components that do not have a counterpart in the application domain, such as persistent data stores, user interface objects, or middleware.

During analysis, we identify application objects, their relationships, and attributes and operations. During system design, we identify subsystems and most important solution objects. During object design, we refine and detail both sets of objects and identify any remaining solution objects needed to complete the system.

7.3.2 Types, signatures, and visibility revisited

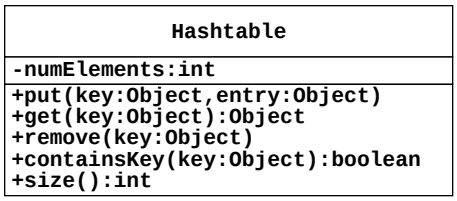
During analysis, we identified attributes and operations without specifying their types or their parameters. During object design, we refine the analysis and system design models by adding type and visibility information. The **type** of an attribute specifies the range of values the attribute can take and the operations that can be applied to the attribute. For example, consider the attribute `numElements` of a `Hashtable` class (see Figure 7-3). `numElements` represent the current number of entries in a given `Hashtable`. Its type is `int`, denoting that it is an integer number. The type of the `numElements` attribute also denotes the operations that can be applied to this attribute: We can add or subtract other integers to `numElements`.

Operation parameters and return values are typed in the same way as attributes are. The type constrains the range of values the parameter or the return value can take. Given an operation, the tuple made out of the types of its parameters and the type of the return value is called the **signature** of the operation. For example, the `put()` operation of `Hashtable` takes two parameters of type `Object` and does not have a return value. The type signature for `put()` is then `(Object, Object):void`. Similarly, the `get()` operation of `Hashtable` takes one `Object` parameter and returns an `Object`. The type signature of `get()` is then `(Object):Object`.

The **visibility** of an attribute or an operation specifies whether it can be used by other classes or not. UML defines three levels of visibility:

- **Private**. A private attribute can be accessed only by the class in which it is defined. Similarly, a private operation can be invoked only by the class in which it is defined. Private attributes and operations cannot be accessed by subclasses or other classes.
- **Protected**. A protected attribute or operation can be accessed by the class in which it is defined and on any descendant of the class.
- **Public**. A public attribute or operation can be accessed by any class.

Visibility is denoted in UML by prefixing the symbol: – (private), # (protected), or + (public) to the name of the attribute or the operation. For example, in Figure 7-3, we specify that the `numElements` attribute of `Hashtable` is private, whereas all operations are public.




```
class Hashtable {
    private int numElements;

    /* Constructors omitted */
    public void put (Object key, Object entry) {…};
    public Object get(Object key) {…};
    public void remove(Object key) {…};
    public boolean containsKey(Object key) {…};
    public int size() {…};

    /* Other methods omitted */
}
```

Figure 7-3 Declarations for the `Hashtable` class (UML class model and Java excerpts).

Type information alone is often not sufficient to specify the range of legitimate values. In the `Hashtable`example, the `int` type allows `numElements`to take negative values, which are not valid for this attribute. We address this issue next with contracts.

7.3.3 Contracts: Invariants, preconditions, and postconditions

Contracts are constraints on a class that enable caller and callee to share the same assumptions about the class [Meyer, 1997]. A contract specifies constraints that the caller must meet before using the class as well as constraints that are ensured by the callee when used. Contracts include three types of constraints:

- An **invariant** is a predicate that is always true for all instances of a class. Invariants are constraints associated with classes or interfaces. Invariants are used to specify consistency constraints among class attributes.
- A **precondition** is a predicate that must be true before an operation is invoked. Preconditions are associated with a specific operation. Preconditions are used to specify constraints that a caller must meet before calling an operation.
- A **postcondition** is a predicate that must be true after an operation is invoked. Postconditions are associated with a specific operation. Postconditions are used to specify constraints that the object must ensure after the invocation of the operation.

For example, consider the Java interface for a `Hashtable`depicted in Figure 7-3. This class provides a `put()` method to create an entry in the table, associating a key with a value, a `get()` method to lookup a value given a key, a `remove()`method to destroy an entry from the `Hashtable`, and a `hashKey()`method which returns a boolean indicating whether or not an entry exists.

An example of an invariant for the `Hashtable`class is that the number of entries in the `Hashtable`is nonnegative at all times. For example, if the `remove()` method results in a negative value of `numElements`, the `Hashtable`implementation is incorrect. An example of a precondition for the `remove()`method is that an entry must be associated with the key passed as a parameter. An example of a postcondition for the `remove()`method is that the removed entry should no long exist in the `Hashtable`after the `remove()`method returns. Figure 7-4 depicts the `Hashtable`class annotated with invariants, preconditions, and postconditions.

```
/* Hashtable class. Maintains mappings from unique keys to arbitrary objects */
class Hashtable {

    /* The number of elements in the Hashtable is nonnegative at all times.
     * @inv numElements >= 0 */
    private int numElements;

    /* Constructors omitted */

    /* The put operation assumes that the specified key is not used.
     * After the put operation is completed, the specified key can be used
     * to recover the entry with the get(key) operation:
     * @pre !containsKey(key)
     * @post containsKey(key)
     * @post get(key) == entry */
    public void put (Object key, Object entry) {…};


    /* The get operation assumes that the specified key corresponds to an
     * entry in the Hashtable.
     * @pre containsKey(key) */
    public Object get(Object key) {…};

    /* The remove operation assumes that the specified key exists in the
     * Hashtable.
     * @pre containsKey(key)
     * @post !containsKey(key) */
    public void remove(Object key) {…};

    /* Other methods omitted */
}
```

Figure 7-4 Method declarations for the `Hashtable` class annotated with preconditions, postconditions, and invariants (Java, constraints in the iContract syntax [iContract]).

We use invariants, preconditions, and postconditions to specify special or exceptional cases unambiguously. For example, the constraints in Figure 7-4 indicate that the `remove()` method should be invoked only for entries that exist in the table. Similarly, the `put()` method should be invoked only if the key is not already in use. In most cases, it is also possible to use constraints to completely specify the behavior of an operation. Such a use of constraints, called *constraint-based specification*, however, is difficult and can be more complicated than implementing the operation itself [Horn, 1992]. We will not describe pure constraint-based specification in the scope of this chapter. Instead, we will focus on specifying operations using both constraints and natural language, emphasizing boundary cases.

7.3.4 UML's Object Constraint Language

In UML, constraints are expressed using OCL [OMG, 1998]. OCL is a language that allows constraints to be formally specified on single model elements (e.g., attributes, operations, classes) or groups of model elements (e.g., associations and participating classes). A constraint is expressed as an OCL expression returning the value True or False. OCL is not a procedural language and thus cannot be used to denote control flow. In this chapter, we focus exclusively on the aspects of OCL related to invariants, preconditions, and postconditions.

A constraint can be depicted as a note attached to the constrained UML element by a dependency relationship. A constraint can be expressed in natural language or in a formal language such as OCL. Figure 7-5 depicts a class diagram of `Hashtable` example of Figure 7-4 using UML and OCL.

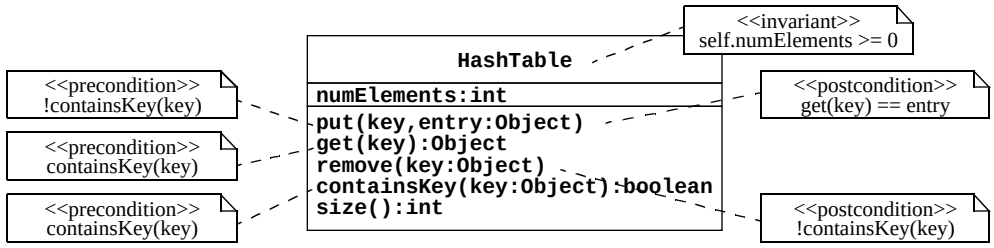


Figure 7-5 Examples of invariants, preconditions, and postconditions in OCL (UML class diagram).

OCL's syntax is similar to object-oriented languages such as C++ or Java. In the case of an invariant, the context for the expression is the class associated with the invariant. The keyword `self` (e.g., `self.numElements` in Figure 7-5) denotes any instance of the class. Attributes and operations are accessed using the dot notation (e.g., `self.attribute` or `self.operation(params)`). The `self` keyword can be omitted if no ambiguity is introduced. (Note that OCL uses the keyword `self` to represent the same concept as the Java and C++ keyword `this`.) For a precondition or a postcondition, the parameters passed to the associated operation can be used in the OCL expression. For postconditions, the suffix `@pre` denotes the

value of a parameter or an attribute before the execution of the operation. For example, a postcondition for the `put(key, entry)` operation expressing that the number of entries in the table increased by one can be represented as `numElements= numElements@pre+ 1`.

Attaching OCL expressions to diagrams can lead to clutter. For this reason, OCL expressions can be alternatively expressed textually. The context keyword introduces a new context for an OCL expression. The word following the context keyword refers to a class, an attribute, or an operation. Then follows one of the keywords inv, pre, and post, which correspond to the UML stereotypes `<<invariant>>`, `<<precondition>>` and `<<postcondition>>` respectively. Then follows the actual OCL expression. For example, the invariant for the `Hashtable` class and the constraints for the `Hashtable` `put ()` operation are written as follows:

```
context Hashtable inv:
numElements >= 0

context Hashtable::put(key, entry) pre:
!containsKey(key)

context Hashtable::put(key, entry) post:
containsKey(key) and get(key) = entry
```

7.4 Object design activities

As we have already mentioned in the introduction, object design includes four groups of activities: specification, component selection, restructuring, and optimization.

Specification activities include:

- identifying missing attributes and operations (Section 7.4.1)
- specifying type signatures and visibility (Section 7.4.2)
- specifying constraints (Section 7.4.3)
- specifying exceptions (Section 7.4.4)

Component selection activities include:

- identifying and adjusting class libraries (Section 7.4.5)
- identifying and adjusting application frameworks (Section 7.4.6)
- a framework example: WebObjects (Section 7.4.7)

Restructuring activities include:

- realizing associations (Section 7.4.8)
- increasing reuse (Section 7.4.9)
- removing implementation dependencies (Section 7.4.10)

Optimization activities include:

- revisiting access paths (Section 7.4.11)
- collapsing objects: turning objects into attributes (Section 7.4.12),
- caching the result of expensive computations (Section 7.4.13)
- delaying expensive computations (Section 7.4.14)

To illustrate these four groups of activities in more detail, we use as example an emissions modeling system called JEWEL (Joint Environmental Workshop and Emissions Laboratory, [Bruegge et al., 1995], [Kompanek et al., 1996]). JEWEL enables end users to simulate and visualize air pollution as a function of point sources (e.g., factories, powerplants), area sources (e.g., cities), and mobile sources (e.g., cars, trucks, trains). The area under study is divided into grid cells. Emissions are then estimated for each grid cell and each hour of the day. Once the simulation is completed, the end user can visualize the concentration of various pollutants on a map along with the emission sources (see Figure 7-6). JEWEL is targeted for government agencies that regulate air quality and attempt to bring troubled populated areas into compliance with regulation.

Given its focus on visualization of geographical data, JEWEL includes a Geographical Information Subsystem (GIS), which is responsible for storing and manipulating maps. The JEWEL `GIS` manages geographical information as sets of polygons and segments. Different types of information, such as roads, rivers, and political boundaries, are organized into different layers that can be displayed independently. Moreover, data is organized such that it can be seen at different levels of abstraction. For example, a high-level view of a map only contains main roads, whereas a detailed view also includes secondary roads. `GIS` is an ideal example for object design, given its rich application domain and complex solution domain. First, we start with the specification of `GIS`.

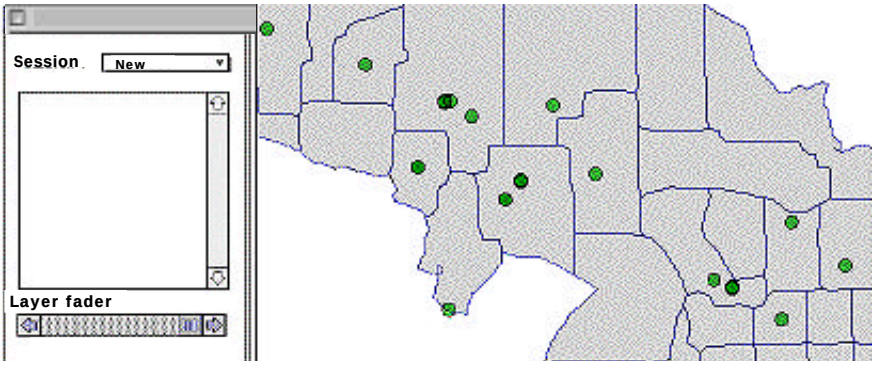


Figure 7-6 Map with political boundaries and emission sources (JEWEL, mock-up).

Specification activities

In Figure 7-7, the object model for the GIS for JEWEL describes an organization in three layers (i.e., the road layer, the water layer, and the political layer). Each layer is composed of elements. Some of these elements, such as highways, secondary roads, and rivers, are displayed with lines composed of multiple segments. Others, such as lakes, states, and counties, are displayed as polygons, which are also represented as collections of line segments.

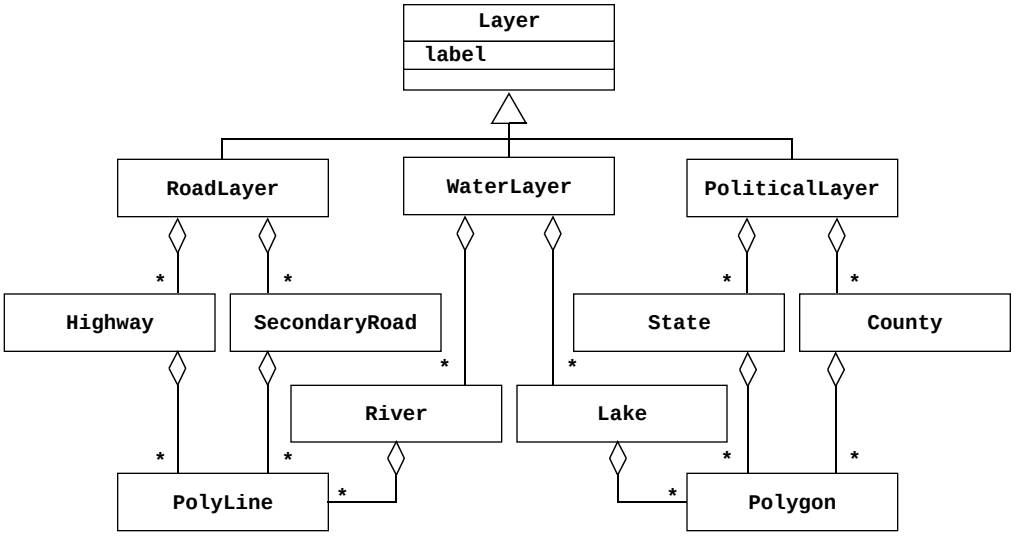


Figure 7-7 Object model for the GIS of JEWEL (UML class diagram).

The ZoomMap use case (Figure 7-8) describes how users can zoom in or out around a selected point on the map. The analysis model is still abstract. For example, it does not contain any information at this point about how zooming is implemented or how points are selected.

The system design model focuses on the subsystem decomposition and global system decisions such as hardware software mapping, persistent storage, or access control. We identify the top-level subsystems and define them in terms of the services they provide. In JEWEL, for example (Figures 7-10 and 7-9), we identified the GIS providing services for creating, storing, and deleting geographical elements, organizing them into layers, and retrieving their outline in terms of a series of points. These services are used by the Visualization subsystem, which retrieves geographical information for drawing maps. The geographical data is provided as a set of flat files and is treated by JEWEL as static data. Consequently, we do not need to support the interactive editing of geographical data. From the use cases of JEWEL, we also know that users need to see geographical data from different zooming criteria. During system design, we decide that the GIS provides zooming and clipping services. The Visualization subsystem specifies

Use case name	`ZoomMap`
Entry condition	The map is displayed in a window, and at least one layer is visible.
Flow of events	1. The end user selects the zoom tool from the tool bar. The system changes the cursor to a magnifying glass. 2. The end user selects a point on the map using the mouse by either clicking the left or the right mouse button. The point selected by the user will become the new center of the map. 3. The end user clicks the left mouse to request an increase in the level of detail (i.e., zoom in) or the right mouse button to request a decrease of the level of detail (i.e., zoom out). 4. The system computes the new bounding box and retrieves from the `GIS` the corresponding points and lines from each visible layer. 5. The system then displays each layer using a predefined color in the new bounding box.
Exit condition	The map is scrolled and scaled to the requested position and detail level.

Figure 7-8 `ZoomMap` use case for JEWEL.

the level of detail and the bounding box of the map, and the `GIS` carries out the zooming and clipping and returns only the points that need to be drawn. This minimizes the amount of data that needs to be transferred between subsystems. Although the system design model is closer to the machine, we have yet to describe in detail the interface of the `GIS`.

Specification activities during object design include:

- identifying missing attributes and operations (Section 7.4.1)
- specifying type signatures and visibility (Section 7.4.2)
- specifying constraints (Section 7.4.3)
- specifying exceptions (Section 7.4.4)

JEWEL **GIS**

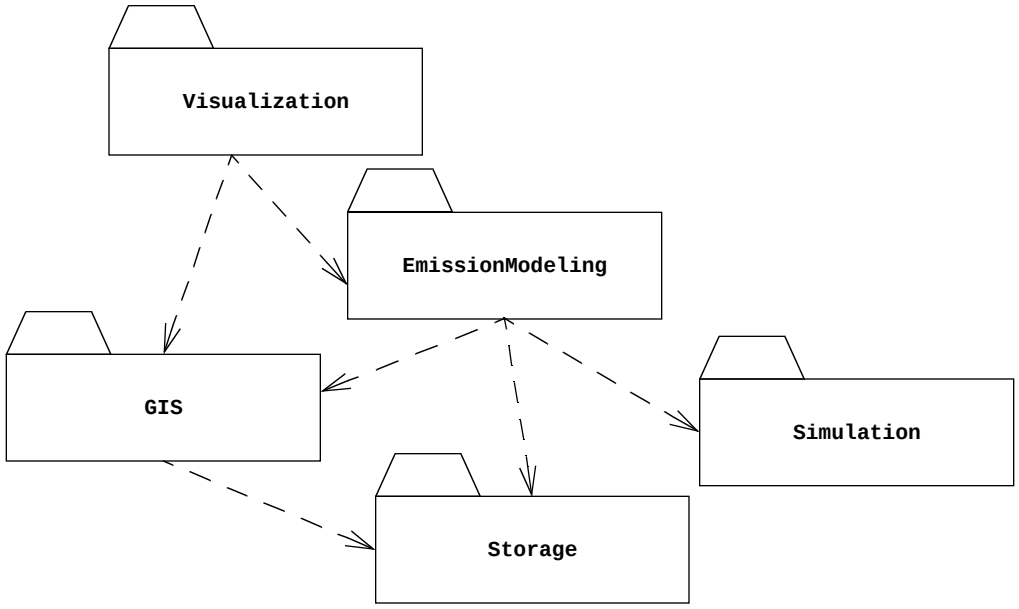
Purpose

- store and maintain the geographical information for JEWEL

Service

- creation and deletion of geographical elements (roads, rivers, lakes, and boundaries)
- organization of elements into layers
- zooming (selection of points given a level of detail)
- clipping (selection of points given a bounding box)

Figure 7-9 Subsystem description for the `GIS` of JEWEL.



EmissionModeling	The EmissionModeling subsystem is responsible for setting up simulations and managing its results.
GIS	The GIS maintains geographical information for Visualization and for EmissionModeling.
Simulation	The Simulation subsystem is responsible for the simulation of emissions.
Storage	The Storage subsystem is responsible for all the persistent data in the system, including geographical and emission data.
Visualization	The Visualization subsystem is responsible for displaying geographical and emissions data to the user.

Figure 7-10 Subsystem decomposition of JEWEL (UML class diagram).

7.4.1 Identifying missing attributes and operations

During this step, we examine the service description of the subsystem and identify missing attributes and operations. During analysis, we may have missed many attributes because we focused on the functionality of the system. Moreover, we described the functionality of the system primarily with the use case model (not the object model). We focused on the application domain when constructing the object model and therefore ignored details related to the system that are independent of the application domain.

In the JEWEL example, the creation, deletion, and organization of layers and layer elements are already supported by the Layer class. We need, however, to identify operations to

realize the clipping and zooming services. Clipping is not a concept that is related to the application domain, but rather is related to the user interface of the system and thus is part of the solution domain.

We draw a sequence diagram representing the control and data flow needed to realize the zoom operation (Figure 7-11). We focus especially on the `Layer` class. When drawing the sequence diagram, we realize that a Layer needs to access all its contained elements to gather their geometry for clipping and zooming. We observe that clipping can be realized independently of the kind of element being displayed, that is, clipping line segments associated with a road or a river can be done using the same operation. Consequently, we identify a new class, the `LayerElement` abstract class (see Figure 7-12), which provides operations for all elements that are part of a `Layer` (i.e., `Highway`, `SecondaryRoad`, `River`, `Lake`, `State`, and `County`).

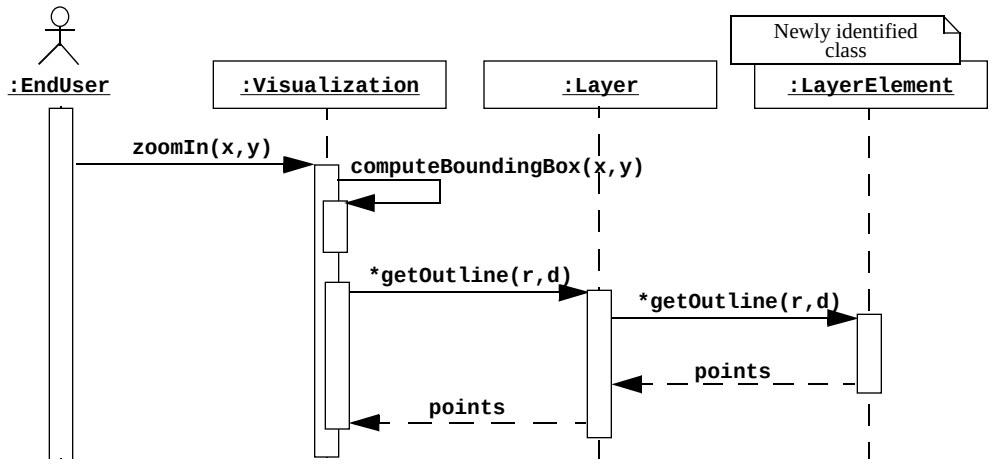


Figure 7-11 A sequence diagram for the `zoomIn()` operation (UML sequence diagram). This sequence diagram leads to the identification of a new class, `LayerElement`. Because we are focusing on the `GIS`, we treat the `Visualization` subsystem as a single object.

We identify the `getOutline()` operation on the `LayerElement` class, which is responsible for scaling and clipping the lines and polygons of individual elements, given a bounding box and a detail level. The `getOutline()` operation uses the detail level to scale each line and polygon and to reduce the number of points for lower levels of detail. For example, when the user zooms out the map by a factor 2, the `GIS` returns only half the number of points for a given layer element, because less detail is needed. We then identify the `getOutline()` operation on the `Layer` class, which is responsible for invoking the `getOutline()` operation on each `LayerElement` and for collecting all lines and polygons of the layer into a single data structure. Both `getOutline()` operations return collections of lines and polygons. The `Visualization`

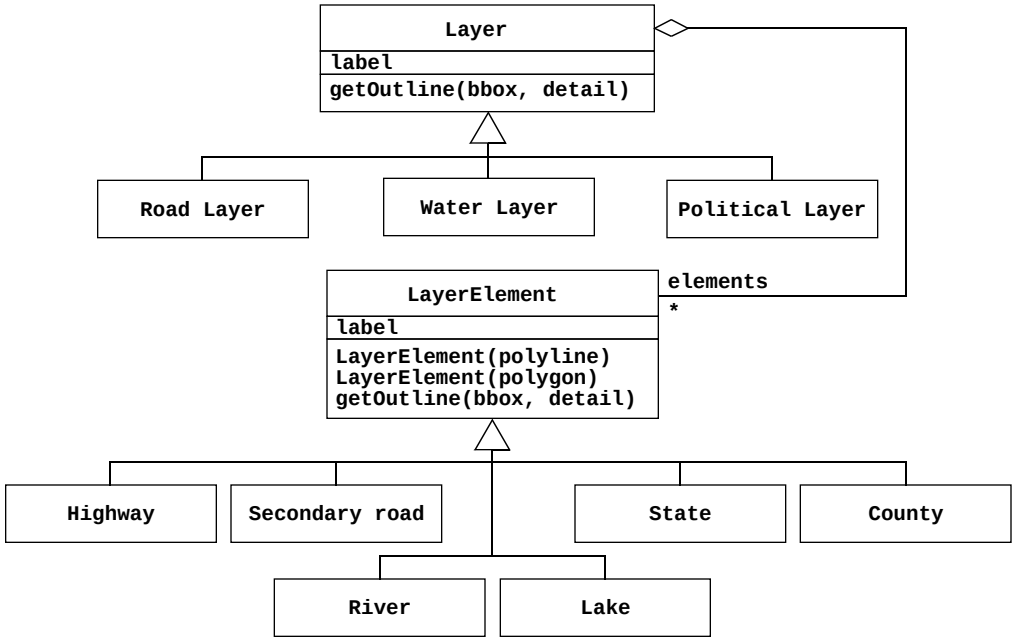


Figure 7-12 Adding operations to the object model of the JEWEL GIS to realize zooming and clipping (UML class diagram).

subsystem then translates the GIS coordinates into screen coordinates, adjusting for scale and scrolling, and draws the line segments on the screen.

During a review of the object model, we realize that the zooming algorithm of the `LayerElement.getOutline()` operation is not trivial: It is not sufficient to select a subset of points of the `LayerElement` and scale their coordinates. Because different `LayerElements` may share points (e.g., two connecting roads, two neighboring counties), the same set of points need to be selected to maintain a visually consistent picture for the end user.

For example, Figure 7-13 displays examples of a naive algorithm for selecting points applied to connecting roads and neighboring counties. The left column displays the points that are selected for a higher level of detail. The right column displays the points that are selected for a lower level of detail. In this case, the algorithm arbitrarily selected every other point, disregarding whether points were shared or not. This leads to elements that are not connected when displayed at lower levels of details.

To address this problem, we decide to include more intelligence in the `PolyLine`, `Polygon`, and `Point` classes (Figure 7-14). First, we decide to represent shared points by exactly one `Point` object; that is, if two lines share a point, both `Line` objects have a reference to the same `Point` object. This is handled by the `Point(x,y)` constructor, which checks if the

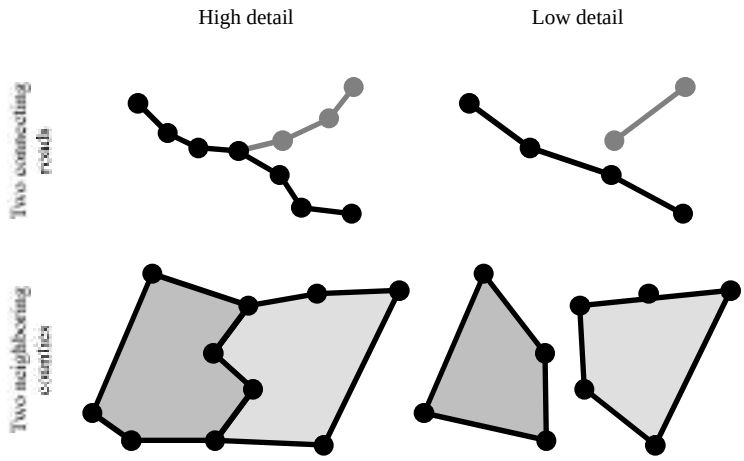


Figure 7-13 A naive point selection algorithm for the `GIS`. The left column represents a road crossing and two neighboring counties. The right column shows that road crossings and neighboring counties may be displayed incorrectly when points are not selected carefully.

specified coordinates correspond to an existing point. Second, we add attributes to `Point` objects to store the levels of details in which they participate. The `inDetailLevels` attribute is a set of all the detail levels in which this `Point` participates. The `notInDetailLevels` attribute is a set of all the detail levels from which this `Point` has been excluded. If a given detail level is not in either set, this means that this `Point` has not yet been considered for the given detail level. The `inDetailLevels` and `notInDetailLevels` attributes (and their associated operations) are then used by the `LayerElement.getOutline()` operation to select shared points and maintain connectivity.

At this point, we identified the missing attributes and operations necessary to support zooming and clipping of `LayerElement` Note that we will probably revisit some of these issues later when we select existing components or perform the object design of the dependent subsystems. Next, we proceed to specifying the interface of each of the classes using types, signatures, contracts, and visibility.

7.4.2 Specifying type signatures and visibility

During this step, we specify the types of the attributes, the signatures of the operations, and the visibility of attributes and operations. Specifying types refines the object design model in two ways. First, we add detail to the model by specifying the range of each attribute. For example, by determining the type of coordinates, we make decisions about the location of the origin (`Point` 0, 0) and the maximum and minimum values for all coordinates. By selecting a double-precision floating-point factor for detail levels, we compute coordinates at different detail levels by simply multiplying the detail level by the coordinates. Second, we map classes

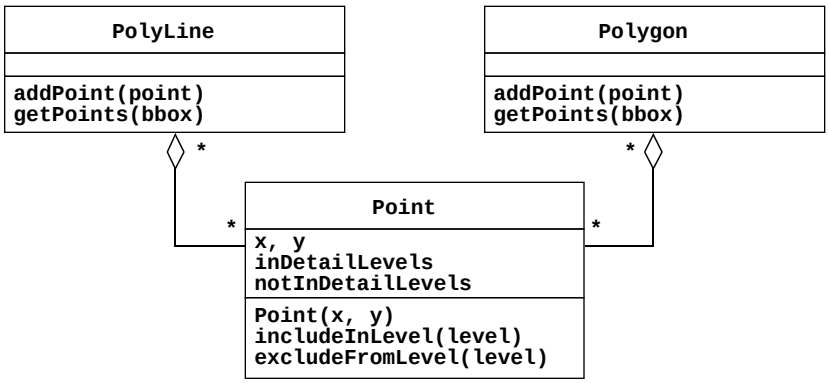


Figure 7-14 Additional attributes and methods for the `Point` class to support intelligent point selection and zooming (UML class diagram).

and attributes of the object model to built-in types provided by the development environment. For example, by selecting `String` to represent the `label` attributes of `Layer`s and `LayerElement`s, we can use all the operations provided by the `String` class to manipulate `label` values.

During this step, we also consider the relationship between the classes we identified and the classes from off-the-shelf components. For example, a number of classes implementing collections are provided in the `java.util` package. The `Enumeration` interface provides a way to access an ordered collection of objects. The `Set` interface provides a mathematical set abstraction implemented by several classes, such as `HashSet`. We select the `Enumeration` interface for returning outlines and the `Set` interface from the `java.util` package for representing the `inDetailLevels` and `notInDetailLevels` attributes.

Finally, we determine the visibility of each attribute and operation during this step. By doing so, we determine which attributes are completely managed by a class, which should only be accessible indirectly via the class's operations, and which attributes are public and can be modified by any other class. Similarly, the visibility of operations allows us to distinguish between operations that are part of the class interface and those that are utility methods that can only be accessed by the class. In the case of abstract classes and classes that are intended to be refined, we also define protected attributes and methods for the use of subclasses only. Figure 7-15 depicts the refined specification of the `Layer`, `LayerElement`, `PolyLine`, and `Point` classes after types, signatures, and visibility have been assigned.

Once we specified the types of each attribute, the signature of each operation, and their visibility, we focus on specifying the behavior and boundary cases of each class by using contracts.

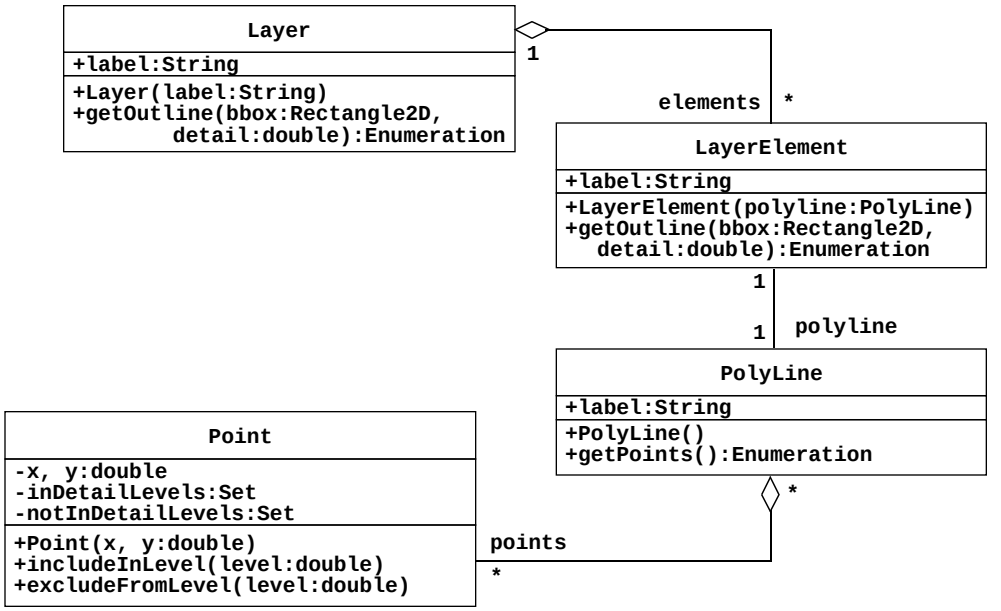


Figure 7-15 Adding type information to the object model of the GIS (UML class diagram). Only selected classes shown for brevity.

7.4.3 Specifying constraints

During this step, we attach constraints to classes and operations to more precisely specify their behavior and boundary cases. Our main goal is to remove as much ambiguity from the model as possible. We specify class contracts using three types of constraints. Invariants represent conditions on the attributes of a class that are always True. Preconditions represent conditions that must be satisfied (usually by the caller) prior to invoking a given operation. Postconditions represent conditions that are guaranteed by the callee after the operation is completed. As described in Section 7.3.3 and Section 7.3.4, we can use OCL [OMG, 1998] to attach constraints to UML models.

In the JEWEL example, the most complex behavior is associated with clipping and zooming; in particular, the `getOutline()` operations on the `Layer` and `LayerElement` classes. In the following, we develop constraints to clarify the `getOutline()` operations, focusing on the behavior associated with shared points. More specifically, we are interested in specifying the following constraints.

1. All points returned by `Layer.getOutline()` are within the specified bounding box.
2. The result of `Layer.getOutline()` is the concatenation of the invocation of `LayerElement.getOutline()` on its elements.

3. At most, one `Point` in the system represents a given (x, y) coordinate.
4. A `detail` level cannot be part of both the `inDetailLevels` and the `notInDetailLevels` sets.
5. For a given `detail` level, `LayerElement.getOutline()` can only return `Point`s which contain the `detail` level in their `inDetailLevels` set attribute.
6. The `inDetailLevels` and `notInDetailLevels` set can only grow as a consequence of `LayerElement.getOutline()`. In other words, once a detail level is in one of these sets, it cannot be removed.

First, we focus on clipping. Given a `LayerElement`, the enumeration of points returned by the `getOutline(bbox,detail)` operation must be inside the specified rectangle `bbox`. Moreover, any point returned by `getOutline()` must be associated with the `LayerElement`. We represent this using a postcondition on the `getOutline()` operation of `LayerElement`. Note that because we currently focus on clipping, we ignore the `detail` parameter.

```
/* Constraint 1 */
context LayerElement::getOutline(bbox, detail) post:
result->forAll(p:Point|bbox.contains(p) and points->includes(p))
```

The `result` field represents the result of the `getOutline()` operation. The `forAll` OCL construct applies the constraint to all points of `result`. Finally, the constraint expresses that all points in the result must be contained in the rectangle `bbox` passed as parameter and must be included in the `points` aggregation association of Figure 7-15.

We then define the `getOutline()` operation on a `Layer` as the concatenation of the enumerations returned by the `getOutline()` operation of the `LayerElement`s. In OCL, we use the `iterate` construct on collections to go through each `LayerElement` and collect its outline into a single enumeration. The `including` construct appends its parameter to the collection. OCL automatically flattens the resulting collection.

```
/* Constraint 2 */
context Layer::getOutline(bbox, detail) post:
elements->iterate(le:LayerElement; result:Enumeration|
    result->including(le.getOutline(bbox,detail))
```

We then focus on constraints related with zooming. Recall that we added attributes and operations to the `Point` class to represent shared points. First, we specify `Point` uniqueness with an invariant applied to all `Point` instances:

```
/* Constraint 3 */
context Point inv:
Point.allInstances->forAll(p1, p2:Point |
    (p1.x = p2.x and p1.y = p2.y) implies p1 = p2)
```

We leave the derivation of the last three constraints as an exercise for the reader (see Exercise 2).

With these six constraints, we describe more precisely the behavior of the `getOutline()` operations and their relationship with the attributes and operations of the `Point` class. Note that we have not described in any way the algorithm by which the `LayerElement` selects `Point`s, given a detail level. We leave this decision to the implementation activity of object design.

Next, we describe the exceptions that can be raised by each operation.

7.4.4 Specifying exceptions

During this step, we specify constraints that the caller needs to satisfy before invoking an operation. In other words, we specify conditions that operations detect and treat as errors by raising an exception. Languages such as Java and Ada have built-in mechanisms for exception handling. Other languages, such as C and early versions of C++ do not support explicit exception handling, so the developers need to establish conventions and mechanisms for handling exceptions (e.g., return values or a specialized subsystem). Exceptional conditions are usually associated with the violation of preconditions. In UML, we attach OCL preconditions to operations and associate the precondition with a dependency to an exception object.

In the JEWEL example (Figure 7-16), we specify that the bbox parameter of the `Layer.getOutline()` operation should have a positive width and height and that the `detail` parameter should be positive. We associate the `ZeroBoundingBox` and the `ZeroDetail` exceptions with each condition.

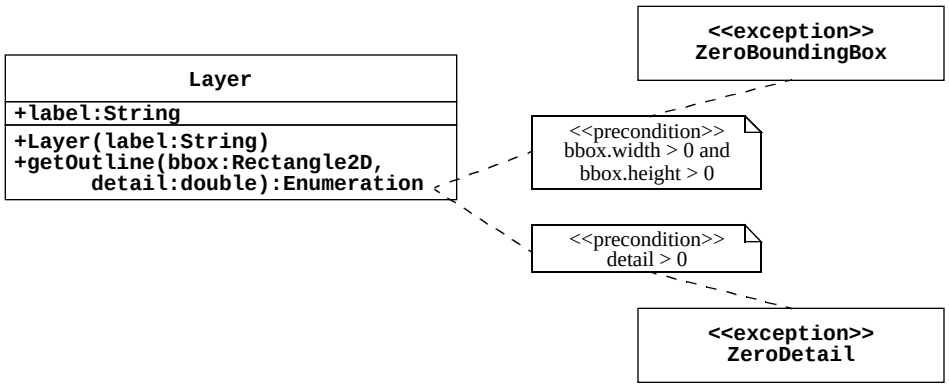


Figure 7-16 Examples of preconditions and exceptions for the `Layer` class of the JEWEL `GIS`.

Exceptions can be found systematically by examining each parameter of the operation and by examining the states in which the operation may be invoked. For each parameter and set of parameters, we identify values or combinations of values that should not be accepted. For example, we reject nonpositive values for the detail level, because the detail parameter

represents a multiplication factor. A zero value for the detail parameters would result in the collapse of all coordinates onto the origin. A negative detail level would result in an inverted picture. Note that a systematic discovery of all exceptions for all operations is a time-consuming exercise, albeit useful. For systems in which reliability is not a primary design goal, the specification of exceptions can be limited to the public interface of subsystems.

Component selection activities

At this point in object design, we selected the software/hardware platform on which the system runs. This platform includes off-the-shelf components such as database management systems, middleware frameworks, infrastructure frameworks, or enterprise application frameworks. The main objective in selecting off-the-shelf components is to reuse as many objects as possible, thus minimizing the number of custom objects that need to be developed. Moreover, an off-the-shelf component often provides a more reliable and efficient solution than any developer could hope to produce in the context of a single system. A user interface class library, for example, pays close attention to an efficient display algorithm or to good response times. An off-the-shelf component also has been used by many more systems and users and therefore is more robust. Off-the-shelf components, however, have a cost. Their purpose is to support a wide variety of systems, and thus they are usually complex. Using an off-the-shelf component requires an investment in learning and often requires a degree of customization. Using off-the-shelf components is usually a better alternative than building the complete system from scratch.

7.4.5 Identifying and adjusting class libraries

Assume that we select the Java Foundation Classes (JFC) [JFC, 1999] as an off-the-shelf component for realizing the `Visualization` subsystem. We need to display the map as a series of polylines and polygons returned by the `Layer.getOutline()` operation. This is usually not straightforward, as we have to reconcile functionality provided by the `GIS` and JFC to realize the `Visualization` services. For example, this can introduce the need for custom objects whose only function is to convert data from one subsystem to another.

For displaying graphics, JFC provides a number of reusable components for composing a user interface. JFC arranges components in a containing hierarchy that constrains the order in which the components are painted. JFC paints last the components closer to the bottom of the hierarchy (usually atomic components) such that they appear on the top of all the other components. A `JFrame`, also known as main window, defines an area that is exclusively owned by the application. A `JFrame` is often made out of several `JPanel`s, each responsible for the layer of several atomic components. A `JScrollPane` provides a scrollable view of a component. It allows a user to view a subset of a component that is too large to display completely.

For the JEWEL `Visualization` subsystem (see Figure 7-17), we select a `JFrame` as a top-level container, a `JToolbar` and two `JButton`s for zooming in and out, and a `JScrollPane` for scrolling the map. We realize the map proper with the `MapArea` class, which refines a `JPanel` and overwrites the `paintContents()` operation. The new `paintContents()` operation

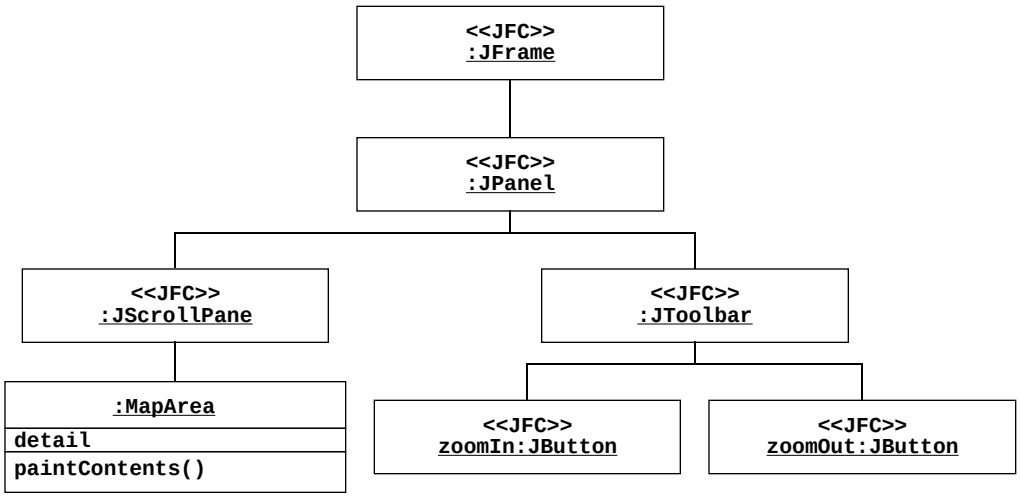


Figure 7-17 JFC components for the JEWEL `Visualization` subsystem (UML object diagram). Associations denote the containment hierarchy used for ordering the painting of components. We use stereotypes to distinguish JEWEL classes from classes provided by JFC.

computes the visible bounding box from attributes of the `JScrollPane`, retrieves lists of points from the `Layer` classes, scales them, and draws them. The `MapArea` class also maintains the current detail level. Actions associated with the `zoomIn:JButton` and `zoomOut:JButton` access operations on the `MapArea` to increase and decrease the detail level, respectively. This triggers the `repaint()` operation on the `MapArea` which refreshes the map display.

When examining the drawing primitives provided by JFC, we realize that JFC and the `GIS` represent lines differently. On the one hand, the `drawPolygon()` and `drawPolyline()` operations of the `Graphics` class accept two arrays of coordinates (one for the x coordinates of the points, the other for the y coordinates; see Figure 7-18). On the other hand, the `getOutline()` operation of the `GIS` returns a `Enumeration` of `Point`s (see Figure 7-16). There are two approaches to resolve this mismatch. We can write a utility method on the `MapArea` class to translate between the two different data structures or we can ask the developers responsible for the `GIS` to change the interface of the `Layer` class.

```
// from java.awt package
class Graphics
//...
    void drawPolyline(int[] xPoints, int[] yPoints, int nPoints) {…};
    void drawPolygon(int[] xPoints, int[] yPoints, int nPoints) {…};
```

Figure 7-18 Declaration for `drawPolyline()` and `drawPolygon()` operations [JFC, 1999].

We should change the API of the `Layer` class if we have control over it. In the general case, however, it is often necessary to write glue operations and classes. For example, if the `GIS` was also made of off-the-shelf components, we would not be able to change its API. We can use the `Adapter`pattern to address this mismatch (see Section 6.4.4 in Chapter 6, *System Design*).

7.4.6 Identifying and adjusting application frameworks

An **application framework** is a reusable partial application that can be specialized to produce custom applications [Johnson et al., 1988]. In contrast to class libraries, frameworks are targeted to particular technologies, such as data processing or cellular communications, or to application domains, such as user interfaces or real-time avionics. The key benefits of application frameworks are reusability and extensibility. Framework reusability leverages of the application domain knowledge and prior effort of experienced developers to avoid the recreation and revalidation of recurring solutions. An application framework enhances extensibility by providing **hook methods**, which are overwritten by the application to extend the application framework. Hook methods systematically decouple the interfaces and behaviors of an application domain from the variations required by an application in a particular context. Framework extensibility is essential to ensure timely customization of new application services and features.

Frameworks can be classified by their position in the software development process.

- **Infrastructure frameworks** aim to simplify the software development process. Examples include frameworks for operating systems [Campbell-Islam 1993], debuggers [Bruegge et al., 1993], communication tasks [Schmidt, 1997], and user interface design [Weinand et al., 1988]. System infrastructure frameworks are used internally within a software project and are usually not delivered to a client.
- **Middleware frameworks** are used to integrate existing distributed applications and components. Common examples include Microsoft's MFC and DCOM, Java RMI, WebObjects [Wilson & Ostrem, 1999], implementations of CORBA [OMG, 1995], and transactional databases.
- **Enterprise application frameworks** are application specific and focus on domains such as telecommunications, avionics, environmental modeling [Bruegge & Riedel, 1994] manufacturing, financial engineering [Birrer, 1993], and enterprise business activities.

Infrastructure and middleware frameworks are essential to rapidly create high-quality software systems, but they are usually not requested by external customers. Enterprise frameworks, however, support the development of end-user applications. As a result, buying infrastructure and middleware frameworks is more cost effective than building them [Fayad & Hamu, 1997].

Frameworks can also be classified by the techniques used to extend them.

- **White box frameworks** rely on inheritance and dynamic binding for extensibility. Existing functionality is extended by subclassing framework base classes and overriding predefined hook methods using patterns such as the template method pattern [Gamma et al., 1994].
- **Black box frameworks** support extensibility by defining interfaces for components that can be plugged into the framework. Existing functionality is reused by defining components that conform to a particular interface and integrating these components with the framework, using delegation.

Whitebox frameworks require intimate knowledge of the framework's internal structure. Whitebox frameworks produce systems that are tightly coupled to the specific details of the framework's inheritance hierarchies, and thus changes in the framework can require the recompilation of the application. Blackbox frameworks are easier to use than whitebox frameworks, because they rely on delegation instead of inheritance. However, blackbox frameworks are more difficult to develop, because they require the definition of interfaces and hooks that anticipate a wide range of potential use cases. Moreover, it is easier to extend and reconfigure blackbox frameworks dynamically, as they emphasize dynamic object relationships rather than static class relationships. [Johnson et al., 1988].

Frameworks are closely related to design patterns, class libraries, and components.

Design patterns versus frameworks. The main difference between frameworks and patterns is that frameworks focus on reuse of concrete designs, algorithms, and implementations in a particular programming language. In contrast, patterns focus on reuse of abstract designs and small collections of cooperating classes. Frameworks focus on a particular application domain, whereas design patterns can be viewed more as building blocks of frameworks.

Class libraries versus frameworks. Classes in a framework cooperate to provide a reusable architectural skeleton for a family of related applications. In contrast, class libraries are less domain specific and provide a smaller scope of reuse. For instance, class library components, such as classes for strings, complex numbers, arrays, and bitsets can be used across many application domains. Class libraries are typically passive; that is, they do not implement or constrain the control flow. Frameworks, however, are active; that is, they control the flow of control within an application. In practice, frameworks and class libraries are complementary technologies. For instance, frameworks use class libraries, such as foundation classes, internally to simplify the development of the framework. Similarly, application-specific code invoked by framework event handlers uses class libraries to perform basic tasks, such as string processing, file management, and numerical analysis.

Components versus frameworks. Components are self-contained instances of classes that are plugged together to form complete applications. In terms of reuse, a component is a blackbox that defines a cohesive set of operations, which can be used based solely with knowledge of the syntax and semantics of its interface. Compared with frameworks, components are less tightly coupled and can even be reused on the binary code level. That is, applications can reuse components without having to subclass from existing base classes. The advantage is that applications do not always have to be recompiled when components change. The relationship between frameworks and components is not predetermined. On the one hand, frameworks can be used to develop components, where the component interface provides a facade pattern for the internal class structure of the framework. On the other hand, components can be plugged into blackbox frameworks. In general, frameworks are used to simplify the development of infrastructure and middleware software, whereas components are used to simplify the development of end-user application software.

7.4.7 A framework example: WebObjects

WebObjects is a set of frameworks for developing interactive Web applications accessing existing data from relational databases. WebObjects consists of two infrastructure frameworks. The WebObjects framework[1] handles the interaction between Web browsers and Web servers. The Enterprise Object Framework (EOF) handles the interaction between Web servers and relational databases. The EOF supports database adapters that allow applications to connect to database management systems from particular vendors. For example, the EOF provides database adapters for Informix, Oracle, and Sybase servers and ODBC compliant adapters for databases running on the Windows platform. In the following, we concentrate on the WebObjects framework. More information on the EOF can be found in [Wilson & Ostrem, 1999].

Figure 7-19 shows an example of a dynamic publishing site built with WebObjects. The `WebBrowser` originates an HTTP request in the form of a URL, which is sent to the `WebServer`. If the `WebServer` detects that the request is to a static HTML page, it passes it on the `StaticHTML` object, which selects and sends the page back to the Web browser as a response. The Web browser then renders it for the user. If the `WebServer` detects that the request requires a dynamic HTML page, it passes the request to a WebObjects `WOAdapter`. The WebObjects `Adapter` packages the incoming HTML request and forwards it to the `WebObjectsApplication` object. Based on `Templates` defined by the developer and relevant data retrieved from the `RelationalDatabase`, the `WebObjectsApplication` then generates an HTML response page, which is passed back through the `WOAdaptor` to the `WebServer`. The `WebServer` then sends the page to the `WebBrowser`, which renders it for the user.

1. "WebObjects" is unfortunately the name of both the complete development environment and the Web framework. When referring to the framework, we always use the phrase, "WebObjects framework." When referring to the development environment, we simple use the term, "WebObjects."

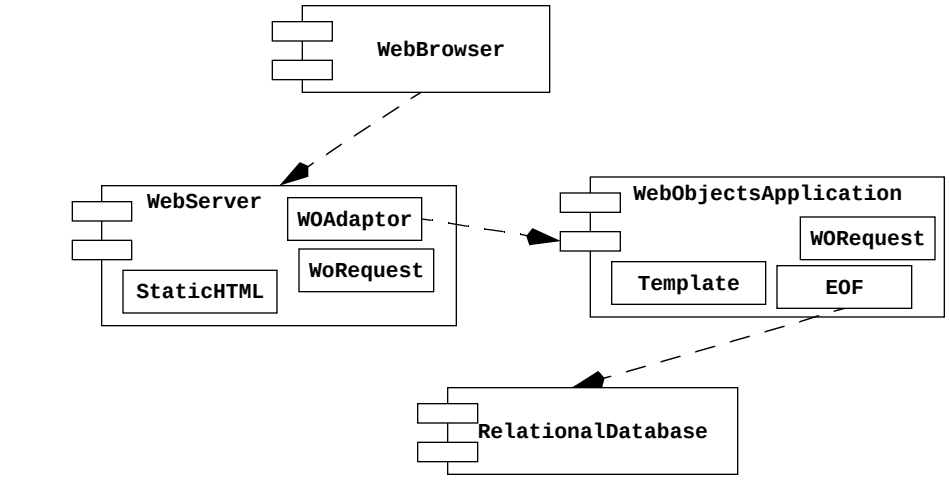


Figure 7-19 An example of dynamic site with WebObjects (UML component diagram).

A key abstraction provided by the WebObjects framework is an extension of the HTTP protocol to manage state. HTTP is a stateless request-response protocol, that is, a response is formulated for each request, but no state is maintained between successive requests. In many Web-based applications, however, state needs to be kept between requests. For example in JEWEL, emissions computations can take up to 30 days. The end user must be able to monitor and access the state of the emissions computation even if the Web browser is restarted. Several techniques have been proposed to keep track of state information in Web applications, including dynamically generated URLs, cookies, and hidden HTML fields. WebObjects provides the classes shown in Figure 7-20 to achieve the same purpose.

The `WOApplication` class represents the application running on the `WebServer` waiting for requests from the associated `WebBrowser`. A cycle of the request-response loop begins whenever the `WOAdaptor` receives an incoming HTTP request. The `WOAdaptor` packages this request in a `WORequest` object and forwards it to the application object of class `WOApplication`. Requests are always triggered by a URL submitted by the `WebBrowser`. A top-level URL represents a special request and causes the creation of a new instance of type `WOSession`. The class `WOSession` encapsulates the state of an individual session, allowing it to track different users, even within a single application. A `WOSession` consists of one or more `WOComponents`, which represent a reusable Web page or portion of a Web page for display within an individual session. `WOComponents` may contain dynamic elements. When an application accesses the database, one or more of the dynamic elements of a component are filled with information retrieved from the database. The `WOSessionStore` provides persistency for `WOSession` objects: It stores sessions in the server and restores them by the application upon request.

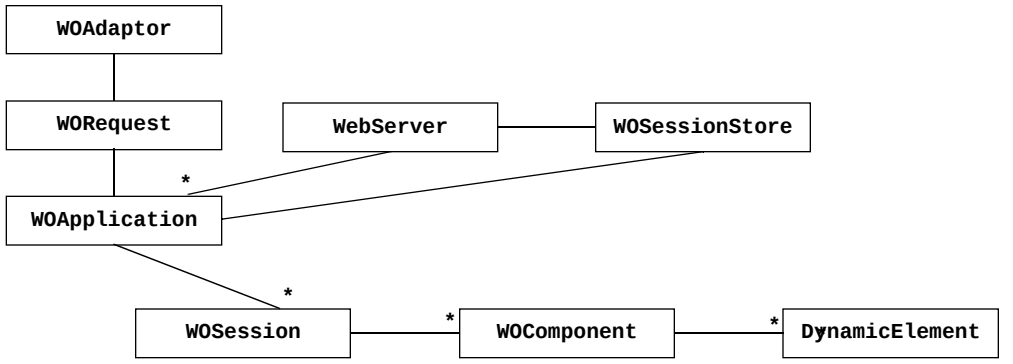


Figure 7-20 WebObject's State Management Classes. The HTTP protocol is inherently stateless. The State Management Classes allow to maintain information between individual requests.

The essence of building a WebObjects application is to refine the classes `WOApplication`, `WOSession`, and `WOComponent` and to intercept the flow of requests sent and received between them. Inherited methods from these classes are overridden when the developer needs to extend the default behavior. The earliest control point for refining objects of type `WOApplication` is when they are constructed. The last point of control is when the application object terminates. By adding code to the application object constructor or overriding the `WOApplication terminate()` method, the developer can customize the behavior of the WebObjects application as desired.

Once we have extended the object design model with off-the-shelf components and their related classes, we restructure the model to improve reusability and extensibility.

Restructuring activities

Once we have specified the subsystem interfaces, identified additional solution classes, selected components, and adapted them to fit our solution, we need to transform the object design model into a representation that is closer to the target machine. In this section, we describe three restructuring activities:

- realizing associations (Section 7.4.8)
- revisiting inheritance to increase reuse (Section 7.4.9)
- revisiting inheritance to remove implementation dependencies (Section 7.4.10)

7.4.8 Realizing associations

Associations are UML concepts that denote collections of bidirectional links between two or more objects. Object-oriented programming languages, however, do not provide the concept of association. Instead, they provide references, in which one object stores a handle to another object. References are unidirectional and take place between two objects. During object design,

we realize associations in terms of references, taking into account the multiplicity of the associations and their direction. Note that many UML modeling tools accomplish the transformation of associations into references automatically. Even if a tool accomplishes this transformation, it is nevertheless important that developers understand its rationale, as they have to deal with the generated code.

Unidirectional one-to-one associations. The simplest association is a one-to-one association. For example, `ZoomInAction`, the control object implementing the `ZoomIn` use case, has a one-to-one association with the `MapArea` whose `detail` level the `ZoomInAction` object modifies (Figure 7-21). Assume, moreover, that this association is unidirectional; that is, a `ZoomInAction` accesses the corresponding `MapArea`, but a `MapArea` does not need to access the corresponding `ZoomInAction` object. In this case, we realize this association using a reference from the `ZoomInAction`, that is, an attribute of `ZoomInAction` named `targetMap` of type `MapArea`

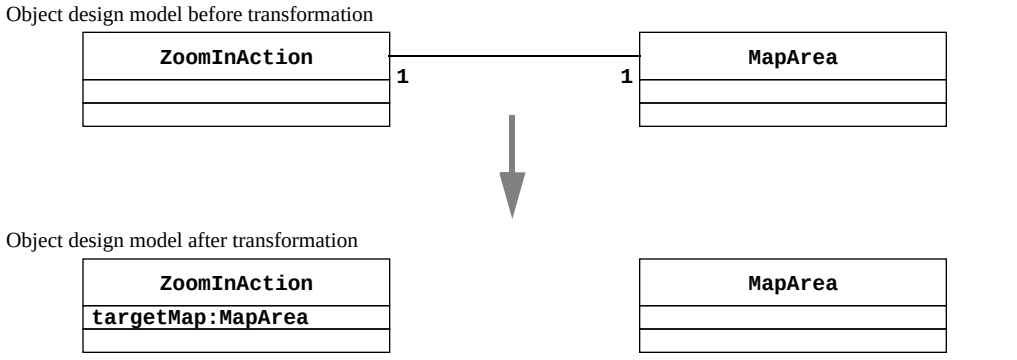
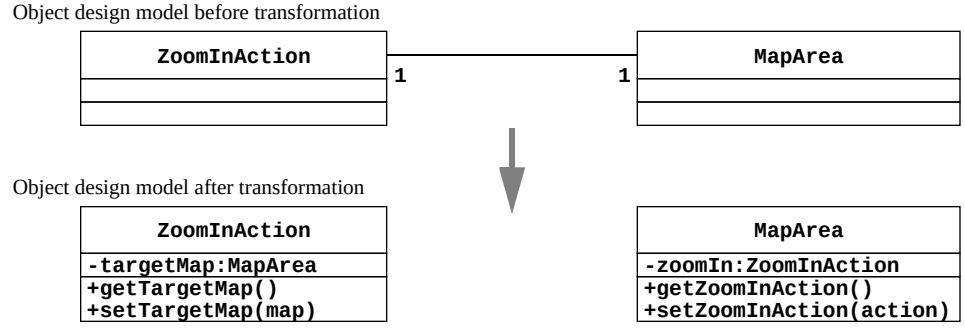


Figure 7-21 Realization of a unidirectional, one-to-one association (UML class diagram; arrow denotes the transformation of the object model).

Creating the association between `ZoomInAction` and `MapArea` translates into setting the `targetMap` attribute to refer to the correct `MapArea` object. Because each `ZoomInAction` object is associated with exactly one `MapArea`, a `null` value for the `targetMap` attribute can only occur when a `ZoomInAction` object is being created. A `null` `targetMap` is otherwise considered an error.

Bidirectional one-to-one associations. Assume that we modify the `MapArea` class so that the user can zoom by simply clicking on the map with the left and right button. In this case, a `MapArea` needs to access its corresponding `ZoomInAction` object. Consequently, the association between these two objects needs to be bidirectional. We add the `zoomIn` attribute to `MapArea` (Figure 7-22). This, however, is not sufficient: By adding a second attribute to realize the association, we introduce redundancy into the model. We need to ensure that if a given `MapArea` has a reference to a specific `ZoomInAction`, the `ZoomInAction` has a reference to that same

`MapArea` To ensure consistency, we change the visibility of the attributes to private and add two methods to each class to access and modify them. `setZoomInAction()` on the `MapArea` sets the `zoomIn` attribute to its parameter and then invokes `setTargetMap()` on `ZoomInAction` to change its `targetMap` attribute.[2] Finally, we need to address the initialization of the association and its destruction by calling `setTargetMap()` and `setZoomInAction()` when `MapArea` and `ZoomInAction` objects are created and destroyed. This ensures that both reference attributes are consistent at all times.




```
class MapArea extends JPanel {
    private ZoomInAction zoomIn;
    /* Other methods omitted */
    void setZoomInAction (action:ZoomInAction) {
        if (zoomIn != action) {
            zoomIn = action;
            zoomIn.setTargetMap(this);
        }
    }
}
class ZoomInAction extends AbstractAction {
    private MapArea targetMap;
    /* Other methods omitted */
    void setTargetMap(map:MapArea) {
        if (targetMap != map) {
            targetMap = map;
            targetMap.setZoomInAction(this);
        }
    }
}
```

Figure 7-22 Realization of a bidirectional one-to-one association (UML class diagram and Java excerpts; arrow denotes the transformation of the object design model).

2. Note that the `setZoomInAction()` and the `setTargetMap()` methods need to check first if the attribute needs to be modified before invoking the other method, such that they avoid an infinite recursion (see code in Figure 7-22).

The direction of an association can often change during the development of the system. Unidirectional associations are much simpler to realize. Bidirectional associations are more complex and introduce mutual dependencies among classes. For example, in Figure 7-22, both the `MapArea` and the `ZoomInAction` classes need to be recompiled and tested when we change either class. In the case of a unidirectional association from the `ZoomInAction` class to the `MapArea` class, we do not need to worry about the `MapArea` class when we change the `ZoomInAction` class. Bidirectional associations, however, are sometimes necessary in the case of peer classes that need to work together closely. The choice between a unidirectional or a bidirectional association is a trade-off that we need to evaluate in the context of a specific pair of classes. To make the trade-off easier, however, we can systematically make all attributes private and provide corresponding `setAttribute()` and `getAttribute()` operations to modify the reference. This minimizes changes to class interfaces when making a unidirectional association bidirectional (and vice versa).

One-to-many associations. One-to-many associations, unlike one-to-one associations, cannot be realized using a single reference or a pair of references. Instead, we realize the "many" part using a collection of references. For example, the `Layer` class of the JEWEL GIS has a one-to-many association with the `LayerElement` class. Because `LayerElement`s have no specific order with respect to `Layer`s and because a `LayerElement` can be part of a `Layer` at most once, we use a set of references to model the "many" part of the association. Moreover, we decide to realize this association as a bidirectional association and so add the `addElement()`, `removeElement()`, `getLayer()` and `setLayer()` methods to the `Layer` and `LayerElement` classes to update the `layerElements` and `containedIn` attributes (see Figure 7-23). As in the one-to-one example, the association needs to be initialized and destroyed when `Layer` and `LayerElement` objects are created and destroyed.

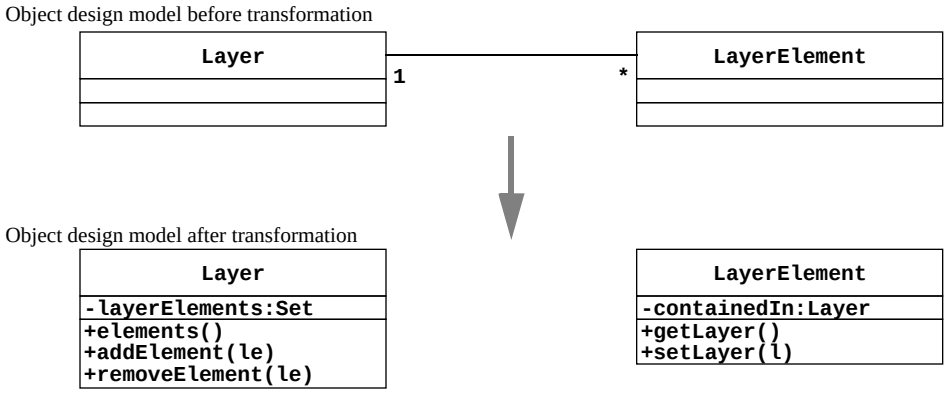


Figure 7-23 Realization of a bidirectional, one-to-many association (UML class diagram; arrow denotes the transformation of the object design model).

Note that the collection on the "many" side of the association depends on the constraints on the association. For example, if the `LayerElement`s of a `Layer` need to be ordered (e.g., indicating the order in which they should be drawn), we need to use an `Array` or a `Vector` instead of a `Set`. Similarly, if an association is qualified, we use a `Hashtable` to store the references.

Many-to-many associations. In this case, both end classes have attributes that are collections of references and operations to keep these collections consistent. For example, the `Polyline` class of the JEWEL GIS has an ordered many-to-many association with the `Point` class. This association is realized by using a `Vector` attribute in each class, which is modified by the operations `addPoint()`, `removePoint()`, `addPolyline()`, and `removePolyline()` (see Figure 7-24). As in the previous example, these operations ensure that both `Vector`s are consistent. Note, however, that the association between `Polyline` and `Point` should be unidirectional, given that none of the `Point` operations needs to access the `Polyline`s that include a given `Point`. We could then remove the `polylines` attribute and its related methods, in which case a unidirectional many-to-many association or a unidirectional one-to-many association becomes identical at the object design level.

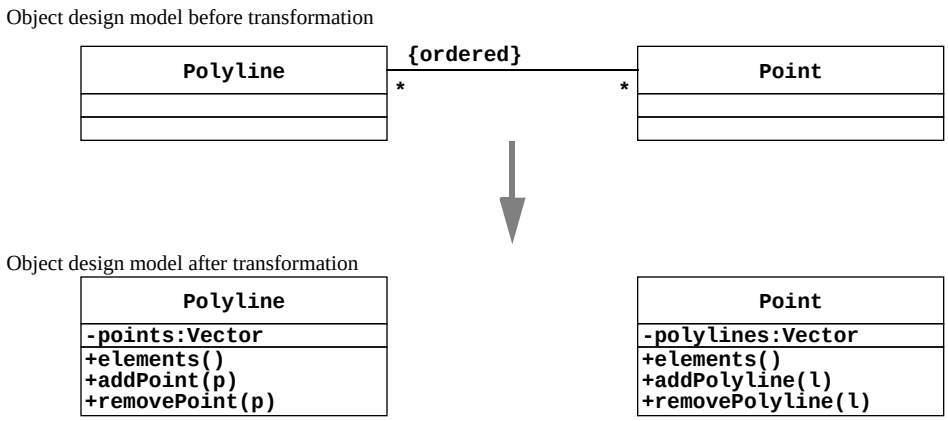


Figure 7-24 Realization of a bidirectional, many-to-many association (UML class diagram; arrow denotes the transformation of the object design model).

Associations as separate objects. In UML, associations can be associated with an association class that holds the attributes and operations of the association. We first transform the association class into a separate object and a number of binary associations. For example, consider the `SimulationRun` association in JEWEL (Figure 7-25). A `SimulationRun` relates an `EmissionSource` object and a `SimulationResult` object. The `SimulationRun` association class also holds attributes specific to the run, such as the date it was created, the user who ran the

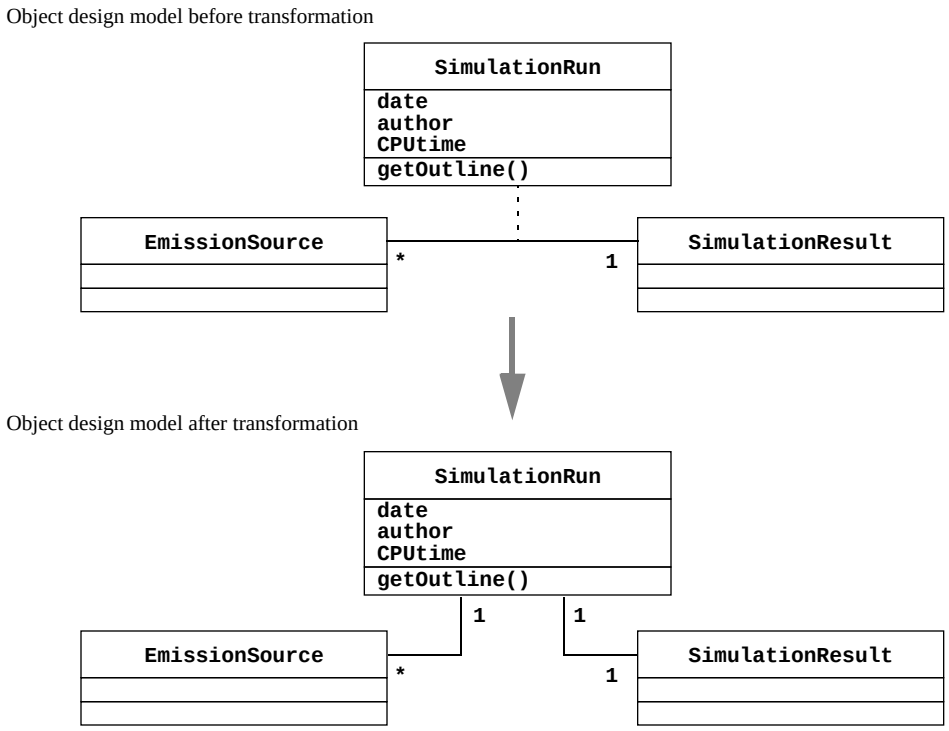


Figure 7-25 Transformation of an association class into an object and two binary associations (UML class diagram; arrow denotes the transformation of the object design model). Once the model contains only binary associations, each association is realized by using reference attributes and collections of references.

simulation, and the CPU time it took to complete the simulation. We first convert the association to an object called `SimulationRun` and two binary associations between the `SimulationRun` object and the other objects. We can then use the techniques discussed earlier to convert each binary association to a set of reference attributes.

Qualified associations. In this case, one or both association ends are associated with a key that is used to differentiate between associations. Qualified associations are realized the same way as one-to-many and many-to-many associations are, except for using a `Hashtable` object on the qualified end (as opposed to a `Vector` or a `Set`). For example, consider the association between `Scenario` and `SimulationRun` in JEWEL (Figure 7-26). A `Scenario` represents a situation that the users are investigating (e.g., a nuclear reactor leak). For each `Scenario`, users can create several `SimulationRuns`, each using a different set of `EmissionSources` or a different `EmissionModel`. Given that `SimulationRuns` are expensive, users also reuse runs across similar `Scenarios`. The `Scenario` end of the association is qualified with a name, enabling the user to distinguish between `SimulationRuns` within the same `Scenario`. We

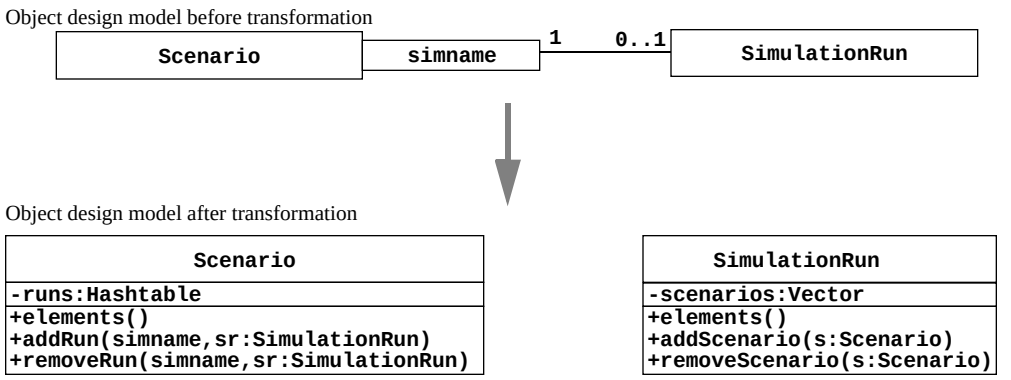


Figure 7-26 Realization of a bidirectional qualified association (UML class diagram; arrow denotes the transformation of the object design model).

realize this qualified association by creating a `runs` attribute on `Scenario`s and a `scenarios` attribute in `SimulationRun`s. The `runs` attribute is a `Hashtable` that is indexed by the name of a `SimulationRun`. Because the name is stored in the `Hashtable`, a specific `SimulationRun` can have different names across `Scenario`s. The `SimulationRun` end is realized, as before, as a `Vector` in the `SimulationRun` class.

7.4.9 Increasing reuse

Inheritance allows developers to reuse code across a number of similar classes. For example, JFC, as do most user interface toolkits, provides four types of buttons:

- a push button (`JButton`), which triggers an action when the end user clicks on the button
- a radio button (`JRadioButton`), which enables an end user to select one choice out of a set of options
- a checkbox (`JCheckBox`), which enables an end user to turn an option on or off
- a menu item (`JMenuItem`), which triggers an action when selected from a pulldown or a popup menu

These four buttons share a set of attributes (e.g., a text label, an icon) and behavior (e.g., something happens when the end user selects them). However, the behavior of each type of button is slightly different. To accommodate these differences while reusing as much code as possible, JFC introduces two abstract classes, `AbstractButton` and `JToggleButton`, and organizes these four types of buttons into the inheritance hierarchy depicted by Figure 7-27. The `AbstractButton` class defines the behavior shared by all JFC buttons. `JToggleButton` defines the behavior shared by the two state buttons (i.e., `JRadioButton` and `JCheckBox`es).

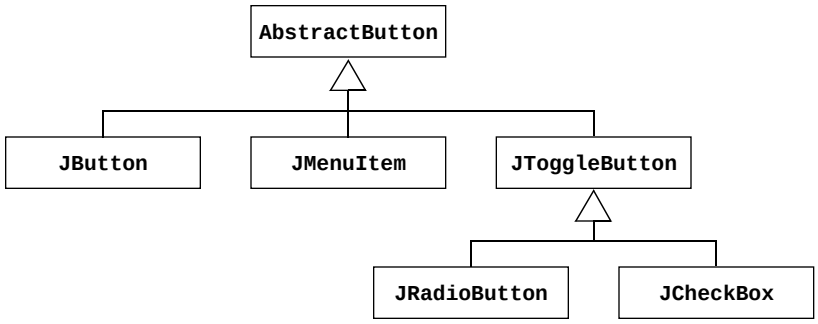


Figure 7-27 An example of code reuse with inheritance (UML class diagram).

There are two main advantages to using a well-designed inheritance hierarchy. First, more code is reused, leading to less redundancies and thus fewer opportunities for defects. Second, the resulting code is extensible, including a well-documented interface for creating future specializations (e.g., new types of buttons in the case of JFC). Reuse through inheritance comes at a cost, however. Developers must correctly anticipate which behavior should be shared and which behavior should be refined by the specialization, often without knowing all possible specializations. Moreover, once developers define an inheritance hierarchy and a paradigm for sharing code, the interfaces of the abstract classes become increasingly more rigid to change as many subclasses and client classes depend on them. Object design represents the last opportunity during development to revisit the inheritance hierarchies among application and solution objects. Any changes later in the process may introduce hard-to-detect defects and substantially increase the cost of the system.

There are two main approaches to designing an inheritance hierarchy for reuse. First, we can examine a number of similar classes and abstract out their common behavior. The `AbstractButton` example of Figure 7-27 is an example of this approach. Second, we can decouple a client class from an anticipated change by introducing a level of abstraction. Most design patterns [Gamma et al., 1994], including the `AbstractFactory` pattern below, use inheritance to protect against an anticipated change.

Consider the problem of writing a single application that works with several windowing styles (e.g., Windows, Macintosh, and Motif). Given a specific platform, the user works with a consistent set of windows, scrollbars, buttons, and menus. The application itself should not know or depend on a specific look and feel. The **Abstract Factory pattern** (Figure 7-28) solves this problem by providing an abstract class for each object that can be substituted (e.g., `AbstractWindow` and `AbstractButton`) and by providing an interface for creating groups of objects (i.e., the `AbstractFactory`). Concrete classes implement each abstract class for each factory. For example, the `AbstractButton` class is refined by the `MacButton` class and the `MotifButton` class. The `AbstractFactory` interface provides a `createButton()` operation to create a button. A concrete factory implements the `AbstractFactory` interface for each option.

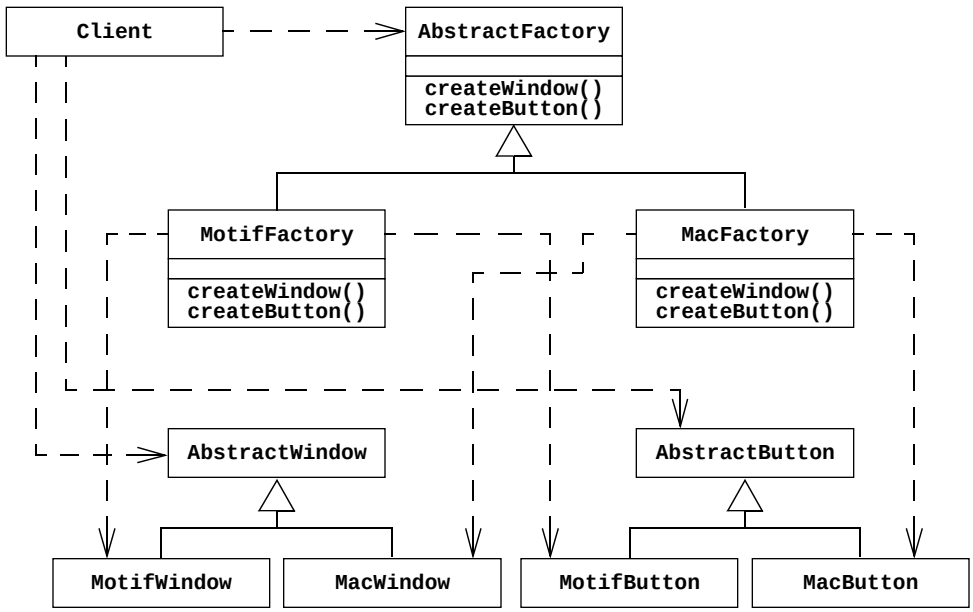


Figure 7-28 Abstract Factory design pattern (UML class diagram, dependencies represent <<call>> relationships). This design pattern uses inheritance to support different look and feels (e.g., Motif and Macintosh). If a new specialization is added, the client does not need to be changed.

The `MotifFactory.createButton()` method returns a `MotifButton`, whereas the `MacFactory.createButton()` method returns a `MacButton`. Note that both `createButton()` methods have the same interface for both specializations. Consequently, the caller only accesses the `AbstractFactory` interface and the abstract classes and is thus shielded from concrete implementations. Moreover, this allows new factories (e.g., `BeOSFactory` and `BeOSButton`) to be implemented in the future without changing the application.

7.4.10 Removing implementation dependencies

In system modeling, we use generalization relationships to classify objects into generalization/specification hierarchies. This allows us to differentiate the common behavior of the general case from the behavior that is specific to specialized objects. In an object-oriented programming language, generalization is realized with inheritance. This allows us to reuse attributes and operations from higher level classes. On the one hand, inheritance when used as a generalization mechanism results in fewer dependencies. For example, in the `AbstractFactory` design pattern (Figure 7-28), dependencies between the application and a specific look and feel is removed using the abstract classes `AbstractFactory`, `AbstractButton`, and

AbstractWindowOn the other hand, inheritance introduces dependencies along the hierarchy. For example, the classes MotifWindow and MacWindow are tightly coupled with AbstractWindowIn the case of generalization, this is acceptable, given that AbstractWindow MotifWindow, and MacWindoware strongly related concepts. These tight dependencies can become a problem when inheritance is used for other purposes than generalization. Consider the following example.

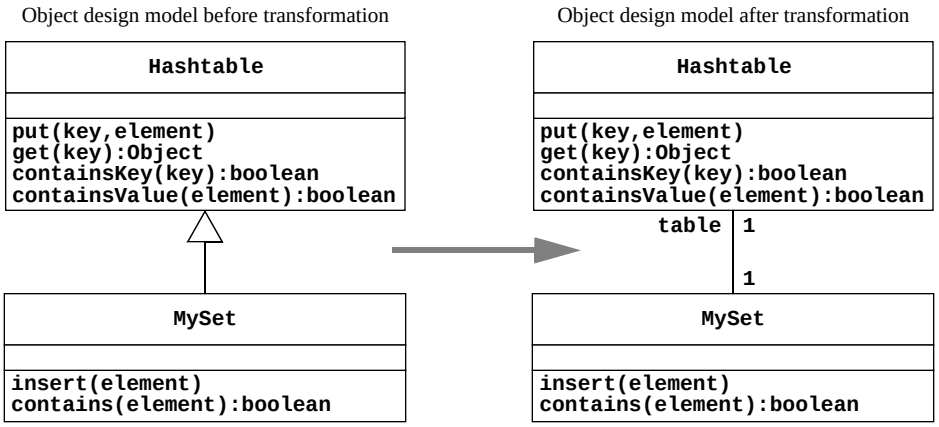
Assume for a moment that Java does not provide a Set abstraction and that we needed to write our own. We decide to reuse the java.util.Hashtableclass to implement a set abstraction that we call MySet. Inserting an element in MySet is equivalent to checking if the corresponding key exists in the table and creating an entry if necessary. Checking if an element is in MySet is equivalent to checking if an entry is associated with the corresponding key (see Figure 7-29, left column).

Such an implementation of a Set allows us to reuse code and provides us with the desired behavior. It also provides us, however, with unwanted behavior. For example, Hashtable implements the containsKey()operation to check if the specified object exists as a key in the Hashtableand the containsValue()operation to check if the specified object exists as an entry. Both of these operations are inherited by MySet. Given our implementation, the operation containsValue()invoked on a MySet object always returns null, which is counterintuitive. Worse, a developer using the MySetcan easily confuse the contains()and containsValue() operations and introduce a fault in the system that is difficult to detect. To address this issue, we could overwrite all operations inherited from Hashtablethat should not be used on MySet. This would lead to a MySetclass that is difficult to understand and reuse.

The fundamental problem in this example is that, although Hashtableprovides behavior that we would like to reuse in implementing Set, because it would save us time, the Set concept is not a refinement of the Hashtable concept. In contrast, the MacWindow class of the AbstractFactoryexample is a refinement of the AbstractWindowclass.

We call the use of inheritance for the sole purpose of reusing code **implementation inheritance**. Implementation inheritance enables developers to reuse code quickly by subclassing an existing class and refining its behavior. A Set implemented by inheriting from a Hashtable is an example of implementation inheritance. Conversely, the classification of concepts into specialization-generalization hierarchies is called **interface inheritance**. Interface inheritance is used for managing the complexity arising for a large number of related concepts. Interface inheritance is also called subtyping, in which case the superclass is called **supertype** and the subclass is called **subtype**. For example, Real and Integer are subtypes of Number. MapAreais a subtype of JPanel.

Implementation inheritance should be avoided. Although it provides a tempting mechanism for code reuse, it yields only short-term benefits and results into systems that are difficult to modify. **Delegation** is a better alternative to implementation inheritance if code can be reused. A class is said to delegate to another class if it implements an operation by merely resending a message to another class. Delegation makes explicit the dependencies between the




```
/* Implementation of MySet using
inheritance */
class MySet extends Hashtable {
    /* Constructor omitted */
    MySet() {
    }

    void insert(Object element) {
        if (this.get(element) == null){
            this.put(element, this);
        }
    }
    boolean contains(Object element){
        return
        (this.get(element)!=null);
    }
    /* Other methods omitted */
}
```

```
/* Implementation of MySet using
delegation */
class MySet {
    Hashtable table;
    MySet() {
        table = Hashtable();
    }
    void insert(Object element) {
        if (table.get(element)==null){
            table.put(element,this);
        }
    }
    boolean contains(Object element) {
        return
        (table.get(element) != null);
    }
    /* Other methods omitted */
}
```

Figure 7-29 An example of implementation inheritance. The left column depicts a questionable implementation of `MySet` using implementation inheritance. The right column depicts an improved implementation using delegation. (UML class diagram and Java).

reused class and the new class. The right column of Figure 7-29 shows an implementation of `MySet` using delegation instead of implementation inheritance. Note that the only significant addition is the attribute `table`and its initialization in the `MySet()`constructor.

For a thorough discussion of the trade-offs related to inheritance and delegation, the reader is referred to [Meyer, 1997].

Optimization activities

The direct translation of an analysis model results into a model that is often inefficient. During object design, we optimize the object model according to design goals, such as minimization of response time, execution time, or memory resources. In this section, we describe four simple optimizations:

- the addition of associations for optimizing access paths
- collapsing objects into attributes
- caching the result of expensive computations
- delaying expensive computations

When applying optimizations, developers must strike a balance between efficiency and clarity. Optimizations increase the efficiency of the system but also make it more complex and difficult to understand the system models.

7.4.11 Revisiting access paths

One common source of inefficient system performance is the repeated traversal of multiple associations when accessing needed information. To identify inefficient access paths, object designers should ask the following questions [Rumbaugh et al., 1991]:

- **For each operation:** How often is the operation called? What associations does the operation have to traverse to obtain the information it needs? Frequent operations should not require many traversals but should have a direct connection between the querying object and the queried object. If that direct connection is missing, an additional association should be added between these two objects.
- **For each association:** If it has a "many" association on one or both sides, is the multiplicity necessary? How often is the "many" side of an association involved in a search? If this is frequent, then the object designer should try to reduce "many" to "one." Otherwise, should the "many" side be ordered or indexed to improve access time?

In interface and reengineering projects, estimates for the frequency of access paths can be derived from the legacy system. In greenfield engineering projects (i.e., systems that are developed from scratch and that are not intended to replace a legacy system), the frequency of access paths are more difficult to estimate. In this case, redundant associations should not be added before a dynamic analysis of the full system—for example, during system testing—has determined which associations participate in the performance bottlenecks.

Another source of inefficient system performance is excessive modeling. During analysis many classes are identified that turn out to have no interesting behavior. In this case, object designers should ask:

- **For each attribute:** What operations use the attribute? Are `set()` and `get()` the only operations performed on the attribute? If yes, does the attribute really belong to this object or should it be moved to a calling object?

The systematic examination of the object model using the above questions should lead to a model with selected redundant associations, with fewer inefficient many-to-many associations, and fewer classes.

7.4.12 Collapsing objects: Turning objects into attributes

During analysis, developers identify many classes that are associated with domain concepts. During system design and object design, the object model is restructured and optimized, often leaving some of these classes with only a few attributes and little behavior. Such classes, when associated only with one other class, can be collapsed into an attribute, thus reducing the overall complexity of the model.

Consider, for example, an object model that includes `Person`s identified by a `SocialSecurity` object. During analysis, two classes may have been identified. Each `Person` is associated with a `SocialSecurity` class, which stores a unique ID string identifying the `Person` Further modeling did not reveal any additional behavior for the `SocialSecurity` object. Moreover, no other classes have associations with the `SocialSecurity` class. In this case, the `SocialSecurity` class should be collapsed into an attribute of the `Person` class.

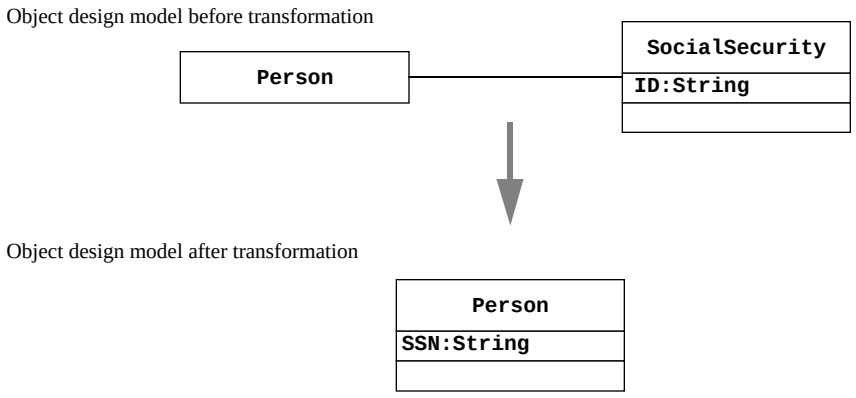


Figure 7-30 Alternative representations of a unique identifier for a `Person` (UML class diagrams).

The decision of collapsing classes is not always obvious. In the case of a social security system, the `SocialSecurity` class may have much more behavior, such as specialized routines for generating new numbers based on birth dates and the location of the original application. In general, developers should delay collapsing decisions until the beginning of the implementation, when responsibilities for each class are clear.

7.4.13 Caching the result of expensive computations

Expensive computations often only need to be done once, because the base values from which the computation is done do not change or change slowly. In such cases, the result of the computation can be cached as a private attribute. Consider, for example, the `Layer.getOutline()` operation. Assume all `LayerElements` are defined once as part of the configuration of the system and do not change during the execution. Then, the vector of `Points` returned by the `Layer.getOutline()` operation is always the same for a given bbox and detail. Moreover, end users have the tendency to focus on a limited number of points around the map as they focus on a specific city or region. Taking into account these observations, a simple optimization is to add a private `cachedPoints` attribute to the `Layer` class, which remembers the result of the `getOutline()` operation for given bbox and detail pairs. The `getOutline()` operation then checks the `cachedPoints` attribute first, returns the corresponding `PointVector`, if found, it otherwise invokes the `getOutline()` operation on each contained `LayerElement`. Note that this approach includes a trade-off: On the one hand, we improve the average response time for the `getOutline()` operation; on the other hand, we consume memory space by storing redundant information.

7.4.14 Delaying expensive computations

An alternate approach to expensive computations is to delay them as long as possible. For example, consider an object representing an image stored as a file. Loading all the pixels that constitute the image from the file is expensive. However, the image data does not need to be loaded until the image is displayed. We can realize such an optimization using a **Proxy pattern** [Gamma et al., 1994]. An `ImageProxy` object takes the place of the `Image` and provides the same interface as the `Image` object (Figure 7-31). Simple operations (such as `width()` and `height()` are handled by `ImageProxy`. When `Image` needs to be drawn, however, `ImageProxy` loads the data from disk and creates an `RealImage` object. If the client does not invokes the `paint()` operation, the `RealImage` object is not created, thus saving substantial computation time. The calling classes only access the `ImageProxy` and the `RealImage` through the `Image` interface.

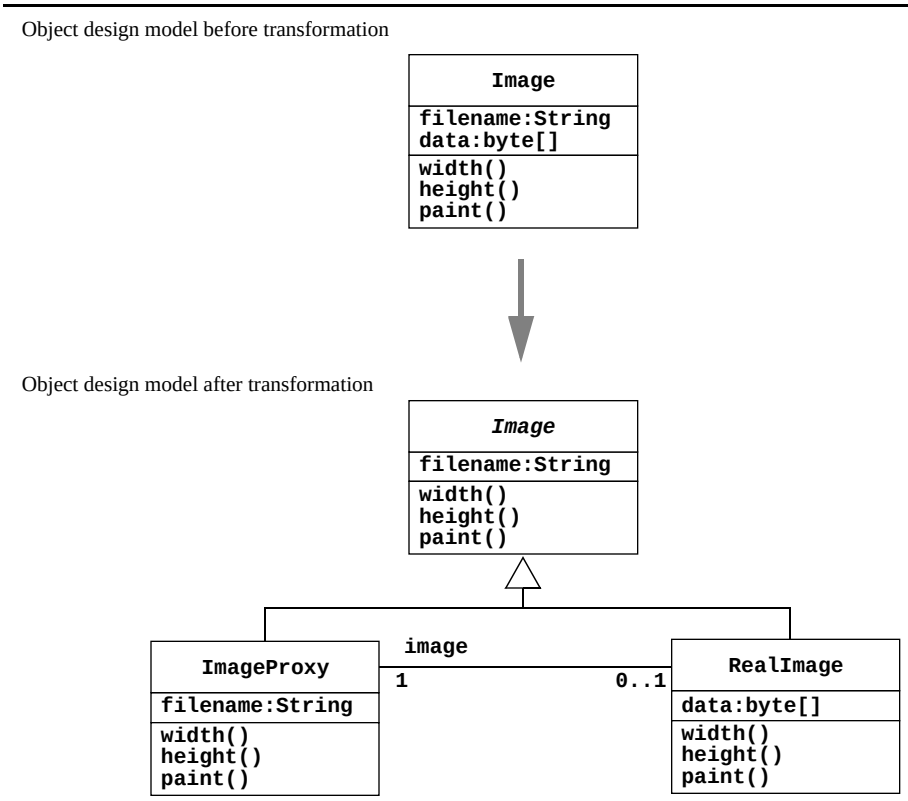


Figure 7-31 Delaying expensive computations using a `Proxy` pattern (UML class diagram).

7.5 Managing object design

In this section, we discuss management issues related to object design. There are two primary management challenges during object design:

- *Increased communication complexity.* The number of participants involved during this phase of development increases dramatically. The object design models and code are the result of the collaboration of many people. Management needs to ensure that decisions among these developers are made consistently with project goals.
- *Consistency with prior decisions and documents.* Developers often do not appreciate completely the consequences of analysis and system design decisions before object design. When detailing and refining the object design model, developers may question some of these decisions and reevaluate them in the light of lessons learned. The management challenge is to maintain a record of these revised decisions and to make sure all documents reflect the current state of development.

In Section 7.5.1, we discuss the Object Design Document, its development and maintenance, and its relationship with other documents. In Section 7.5.2, we describe the roles associated with object design.

7.5.1 Documenting object design

Object design is documented in the Object Design Document (ODD). It describes object design trade-offs made by developers, guidelines they followed for subsystem interfaces, the decomposition of subsystems into packages and classes, and the class interfaces. The ODD is used to exchange interface information among teams and as a reference during testing. The audience for the ODD includes system architects, (i.e., the developers who participate in the system design), developers who implement each subsystem, and testers.

The ODD enables developers to understand the subsystem sufficiently well that they can use it. Moreover, a good interface specification enables other developers to implement classes concurrently. In general, an interface specification should satisfy the following criteria [Liskov, 1986]:

- *Restrictiveness*. A specification should be precise enough that it excludes unwanted implementations. Preconditions and postconditions specifying border cases is one way to achieve restrictive specifications.
- *Generality.* A specification, however, should not restrict its implementation. This allows developers to develop and substitute increasingly efficient or elegant implementations that may not have been thought of when the subsystem was specified.
- *Clarity.* A specification should be easily and unambiguously understandable by developers. However restrictive and general a specification may be, it is useless if it is difficult to understand. Certain behaviors are more easily described in natural language, whereas boundary cases can be described with constraints and exceptions.

There are three main approaches to documenting object design.

- *Self-contained ODD generated from model*. The first approach is to document the object design model the same way we documented the analysis model or the system design model: We write and maintain a UML model using a tool and generate the document automatically. This document would duplicate any application objects identified during analysis. The disadvantages of this solution include redundancy with the Requirements Analysis Document (RAD) and a high level of effort for maintaining consistency with the RAD. Moreover, the ODD duplicates information in the source code and requires a high level of effort whenever the code changes. This often leads to an RAD and an ODD that are inaccurate or out of date.
- *ODD as extension of the RAD*. The second approach is to treat the object design model as an extension of the analysis model. In other terms, the object design is considered as the

set of application objects augmented with solution objects. The advantage of this solution is that maintaining consistency between the RAD and the ODD becomes much easier as a result of the reduction in redundancy. The disadvantages of this solution include polluting the RAD with information that is irrelevant to the client and the user. Moreover, object design is rarely as simple as identifying additional solution objects. Often, application objects are changed or transformed to accommodate design goals or efficiency concerns.

- *ODD embedded into source code*. The third approach is to embed the ODD into the source code. As in the first approach, we first represent the ODD using a modeling tool (see Figure 7-32). Once the ODD becomes stable, we use the modeling tool to generate class stubs. We describe each class interface using tagged comments that distinguish source code comments from object design descriptions. We can then generate the ODD using a tool that parses the source code and extracts the relevant information (e.g., Javadoc [Javadoc, 1999a]). Once the object design model is documented in the code, we abandon the initial object design model. The advantage of this approach is that the consistency between the object design model and the source code is much easier to maintain: when changes are made to the source code, the tagged comments need to be updated and the ODD regenerated. In this section, we only focus on this approach.

The fundamental issue is one of maintaining consistency among two models and the source code. Ideally, we want to maintain the analysis model, the object design model, and the source code using a single tool. Objects would then be described once and consistency among documentation, stubs, and code would be maintained automatically.

Presently, however, UML modeling tools provide facilities for generating a document from a model or class stubs from a model. The documentation generation facility can be used, for example, to generate the RAD from the analysis model (Figure 7-32). The class stub generation facility (called forward engineering) can be used in the self-contained ODD approach to generate the class interfaces and stubs for each method.

Some modeling tools provide facilities for reverse engineering, that is, recreating a UML model from source code. Such facilities are useful for creating object models from legacy code. They require, however, substantial hand processing, as the tool cannot recreate bidirectional associations based on reference attributes only.

Tool support currently falls short when maintaining two-way dependencies, in particular between the analysis model and the source code. Some tools, such as Rationale Rose [Rational, 1998], attempt to realize this functionality by embedding information about associations and other UML constructs in source code comments. Even though this allows the tool to recover syntactic changes from the source code, developers still need to update the model descriptions to reflect the changes. Because developers need different tools to change the source code and the model, the model usually falls behind.

Until modeling tools provide better support for maintaining consistency between object models and source code, we find that generating the ODD from source code and focusing the

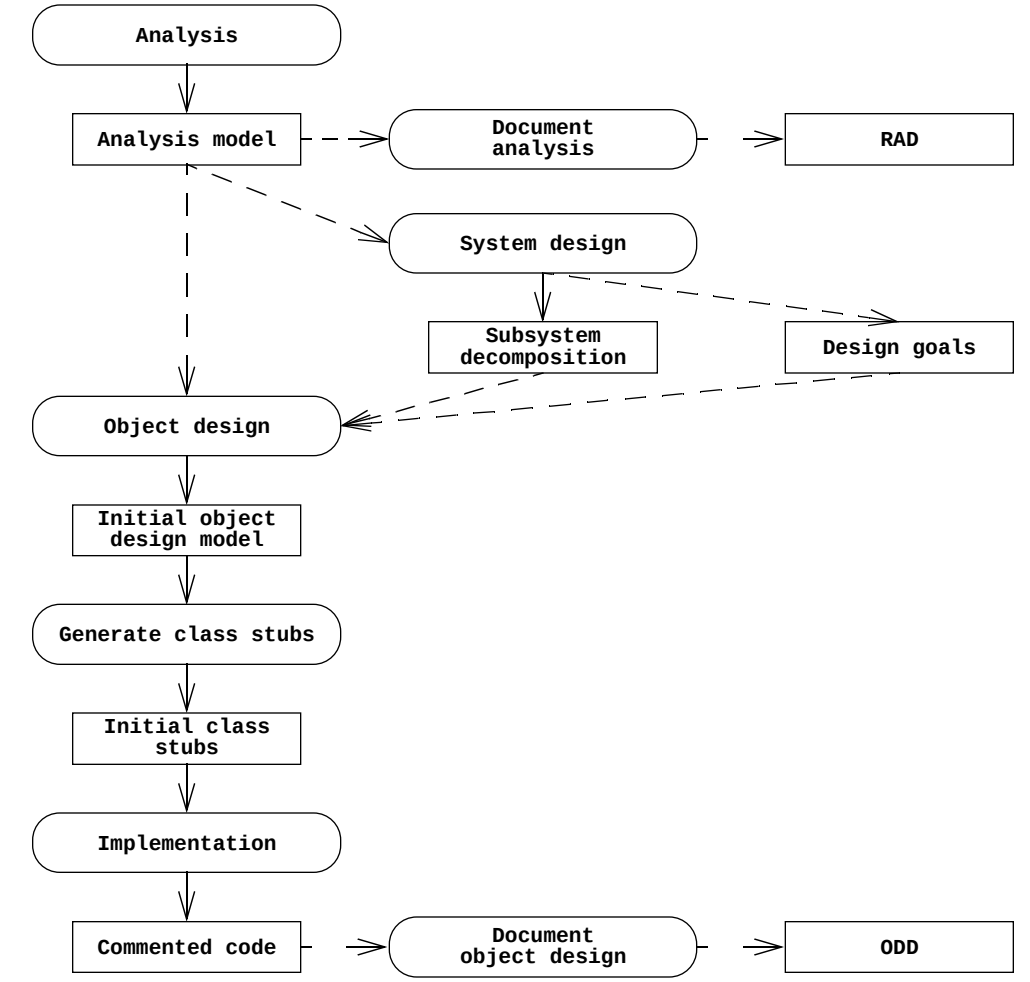


Figure 7-32 Embedded ODD approach. Class stubs are generated from the object design model. The object design model is then documented as tagged comments in the source code. The initial object design model is abandoned and the ODD is generated from the source code instead using a tool such as Javadoc (UML activity diagram).

RAD on the application domain is the most practical. It reduces the amount of redundant information that needs to be maintained, and it locates the object design information where it is the most accessible, that is, the source code. The consistency between the source code and the analysis model still needs to be maintained manually. This task is easier, however, because fewer code changes impact the analysis model than impact the object design model.

The following is an example template for a generated ODD:

Object Design Document

1. Introduction
   - 1.1 Object design trade-offs
   - 1.2 Interface documentation guidelines
   - 1.3 Definitions, acronyms, and abbreviations
   - 1.4 References
2. Packages
3. Class interfaces

Glossary

The first section of the ODD is an introduction to the document. It describes the general trade-offs made by developers (e.g., buy vs. build, memory space vs. response time), guidelines and conventions (e.g., naming conventions, boundary cases, exception handling mechanisms), and an overview of the document.

Interface documentation guidelines and coding conventions are the single most important factor that can improve communication between developers during object design. These include a list of rules that developers should use when designing and naming interfaces. Below are examples of such conventions.

- Classes are named with singular nouns.
- Methods are named with verb phrases, fields, and parameters with noun phrases.
- Error status is returned via an exception only, not a return value.
- Collections and containers have an `elements()` method returning an `Enumeration`.
- `Enumeration`s returned by `elements()` methods are robust to element removals.

Such conventions help developers design interfaces consistently, even if many developers contribute to the interface specification. Moreover, making these conventions explicit before object design makes it easier for developers to follow them. In general, these conventions should not evolve during the project.

The second section of the ODD, *Packages*, describes the decomposition of subsystems into packages and the file organization of the code. This includes an overview of each package, its dependencies with other packages, and its expected usage.

The third section, *Class interfaces*, describes the classes and their public interfaces. This includes an overview of each class, its dependencies with other classes and packages, its public attributes, operations, and the exceptions they can raise.

The initial version of the ODD can be written soon after the subsystem decomposition is stable. The ODD is updated every time new interfaces become available or existing ones are revised. Even if the subsystem is not yet functional, having a source code interface enables

developers to more easily code dependent subsystems and communicate unambiguously. Developers usually discover at this stage missing parameters and new boundary cases. The development of the ODD is different than other documents, as more participants are involved and as the document is revised more frequently. To accommodate a high rate of change and many developers, sections 2 and 3 can be generated by a tool from source code comments.

In Java, this can be done with Javadoc, a tool that generates Web pages from source code comments. Developers annotate interfaces and class declarations with tagged comments. For example, Figure 7-33 depicts the interface specification for the `Layer` class of the JEWEL example. The header comment in the file describes the purpose of the `Layer` class, its authors, its current version, and cross references to related classes. The `@see` tags are used by Javadoc to create cross references between classes. Following the header comment is the class and the method declarations. Each method comment contains a brief description of the purpose of the method, its parameters, and its return result. When using constraints, we also include preconditions and postconditions in the method header. The first sentence of the comment and the tagged comments are extracted and formatted by Javadoc. Keeping material for the ODD with the source code enables the developers to maintain consistency more easily and more rapidly. This is critical when multiple persons are involved.

For any system of useful size, the ODD represents a large amount of information that can translate to several hundreds or thousands of pages of documentation. Moreover, the ODD evolves rapidly during object design and integration, as developers understand better other subsystem's needs and find faults with their specifications. For these reasons, all versions of the ODD should be made available electronically, for example, as a set of Web pages. Moreover, different components of the ODD should be put under configuration management and synchronized with their corresponding source code files. We describe configuration management issues in more detail in Chapter 10, *Software Configuration Management.*

7.5.2 Assigning responsibilities

Object design is characterized by a large number of participants accessing and modifying a large amount of information. To ensure that changes to interfaces are documented and communicated in an orderly manner, several roles collaborate to control, communicate, and implement changes. These include the members of the architecture team who are responsible for system design and subsystem interfaces, liaisons who are responsible for interteam communication, and configuration managers who are responsible for tracking change.

Below are the main roles of object design.

- The **core architect** develops coding guidelines and conventions before object design starts. As for many conventions, the actual set of conventions is not as important as the commitment of all architects and developers to use the conventions. The core architects are also responsible for ensuring consistency with prior decisions documented in the SDD and RAD.

```
/* The class Layer is a container of LayerElements, each representing a
 * polygon or a polyline. For example, JEWEL typically has a road layer, a
 * water layer, a political layer, and an emissions layer.
 * @author John Smith
 * @version 0.1
 * @see LayerElement
 * @see Point
 */
class Layer {

    /* Member variables, constructors, and other methods omitted */
    Enumeration elements() {…};

    /* The getOutline operation returns an enumeration of points representing
     * the layer elements at a specified detail level. The operation only
     * returns points contained within the rectangle bbox.
     * @param      box        The clipping rectangle in universal coordinates
     * @param      detail     Detail level (big numbers mean more detail)
     * @return     A enumeration of points in universal coordinates.
     * @throws     ZeroDetail
     * @throws     ZeroBoundingBox
     * @pre        detail > 0.0 and bbox.width > 0.0 and bbox.height > 0.0
     * @post       forall LayerElement le in this.elements() |
     *                 forall Point p in le.points() |
     *                     result.contains(p)
     */
    Enumeration getOutline(Rectangle2D bbox, double detail) {…};

    /* Other methods omitted */
}
```

Figure 7-33 Interface description of the `Layer` class using Javadoc tagged comments (Java excerpts).

- The **architecture liaisons** document the public subsystem interfaces for which they are responsible. This leads to a first draft of the ODD which is used by developers. Architecture liaisons also negotiate changes to public interfaces when they become necessary. Often, the issue is not of consensus, but rather, of communication: developers depending on the interface may welcome the change if they are notified first. The architecture liaisons and the core architects form the architecture team.
- The **object designers** refine and detail the interface specification of the class or subsystem they implement.
- The **configuration manager** of a subsystem releases changes to the interfaces and the ODD once they become available. The configuration manager also keeps track of the relationship between source code and ODD revisions.
- **Technical writers** from the documentation team clean up the final version of the ODD. They ensure that the document is consistent from a structural and content point of view. They also check for compliance with the guidelines.

As in system design, the architecture team is the integrating force of object design. The architecture team ensures that changes are consistent with project goals. The documentation team, including the technical writers, ensures that the changes are consistent with guidelines and conventions.

7.6 Exercises

1. Consider the `Polyline`, `Polygon`, and `Point` classes of Figure 7-14. Write the following constraints in OCL:
   - A `Polygon` is composed of a sequence of at least three `Points`.
   - A `Polygon` is composed of a sequence of `Points` starting and ending at the same `Point`.
   - The `Points` returned by the `getPoints(bbox)` method of a `Polygon` are within the bbox rectangle.
2. Consider the `Layer`, `LayerElement`, `Polyline`, and `Point` classes of Figure 7-14. Write constraints below in OCL. Note that the last two constraints require the use of the `forAll` OCL operator on collections.
   - A `detail` level cannot be part of both the `inDetailLevels` and the `notInDetailLevels` sets of a `Point`.
   - For a given `detail` level, `LayerElement.getOutline()` can only return `Points` that contain the `detail` level in their `inDetailLevels` set attribute.
   - The `inDetailLevels` and `notInDetailLevels` set can only grow as a consequence of `LayerElement.getOutline()`. In other words, once a detail level is in one of these sets, it cannot be removed.
3. Consider the `Point` class in Figures 7-14 and 7-15. Assume that we are evaluating an alternative design in which a global object called `DetailTable` tracks which `Points` have been included or excluded from a given detail level (instead of each `Point` having a `inDetailLevels` and `notInDetailLevels` attribute). This is realized by two associations between `DetailTable` and `Point`, which are indexed by `detailLevel` (see Figure 7-34). Write OCL constraints specifying that, given a detailLevel, a `DetailTable` can only have one link to a given `Point` (i.e., a `DetailTable` cannot have both an `includesPoint` and an `excludesPoint` association given a `Point` and `detailLevel`).
4. Using the transformations described in Sections 7.4.8–7.4.10, restructure the object design model of Figure 7-34.
5. Incrementally computing the `inDetailLevels` and `notInDetailLevels` attributes of a Point class as depicted in Figure 7-14 is an optimization. Using the terms we introduced in Section , name the kind of optimization that is performed.
6. Discuss the relative advantages of the `Point` class of Exercise 3 versus the `Point` class of Figure 7-14 from a response time and a memory space point of view.

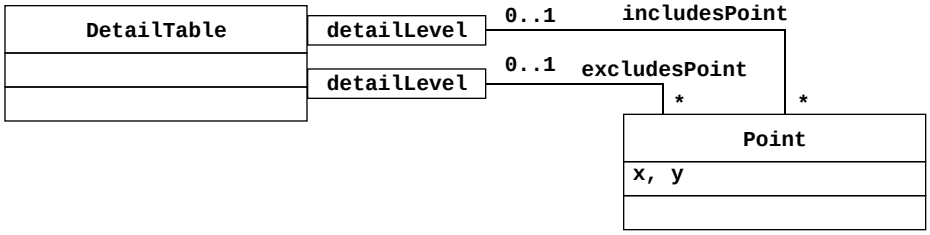


Figure 7-34 `DetailTable` is a global object tracking which `Points` have been included or excluded from a specified `detailLevel`. This is an alternative to the `inDetailLevels` and `notInDetailLevels` sets depicted in Figure 7-14.

7. Assume that you are using an application framework instead of the `GIS` we described in Section 7.4. The GIS application framework provides a `getOutline()` method that returns an `Enumeration` of `GISPoints` to represent a `Polyline` The JFC `drawPolyline()` method takes as parameters two arrays of coordinates and a integer denoting the number of points in the `Polyline` (Figure 7-18). Which design pattern should you use to address this interface mismatch? Draw a UML class diagram to illustrate your solution.
8. You are the developer responsible for the `getOutline()` method of the `Layer` class in Figure 7-15. You find that the current version of `getOutline()` does not properly exclude `Polylines` consisting of a single `Point` (as a result of the clipping). You repair the bug. Who should you notify?
9. You are the developer responsible for the `getOutline()` method of the `Layer` class in Figure 7-15. You change the method to represent `detailLevels` (that are stored in `inDetailLevels` or `notInDetailLevels`) using a positive integer instead of a floating point number. Who should you notify?
10. Why is maintaining consistency between the analysis model and the object design model difficult? Illustrate your point with a change to the object design model.

References

[Birrer, 1993] E. T. Birrer, “Frameworks in the financial engineering domain: An experience report,” *ECOOP'93 Proceedings*, Lecture Notes in Computer Science no. 707, 1993.

[Bruegge et al., 1993] B. Bruegge, T. Gottschalk, & B. Luo, “A framework for dynamic program analyzers,” *OOPSLA' 93, (Object-Oriented Programming Systems, Languages, and Applications)*, Washington, DC, pp. 65–82, Sept. 1993.

[Bruegge & Riedel, 1994] B. Bruegge & E. Riedel. “A geographic environmental modeling system: Towards an object-oriented framework,” *Proceedings of the European Conference on Object-Oriented Programming (ECOOP-94)*, Bologna, Italy, Lecture Notes in Computer Science, Springer Verlag, Berlin, July 1994.

[Bruegge et al., 1995] B. Bruegge, E. Riedel, G. McRae, & T. Russel, "GEMS: An environmental modeling system," *IEEE Journal for Computational Science and Engineering*, pp.55–68, Sept. 1995.

[Fayad & Hamu, 1997] M. E. Fayad & D. S. Hamu, "Object-oriented enterprise frameworks: Make vs. buy decisions and guidelines for selection," *The Communications of ACM*, 1997.

[Gamma et al., 1994] E. Gamma, R. Helm, R. Johnson, & J. Vlissides, *Design Patters: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1994.

[Horn, 1992] B. Horn. "Constraint patterns as a basis for object-oriented programming," in *Proceedings of the OOPSLA'92*, Vancouver, Canada, 1992.

[Hueni et al., 1995] H. Hueni, R. Johnson, & R. Engel, "A framework for network protocol software," *Proceedings of OOPSLA*, Austin, TX, Oct. 1995.

[iContract] Java Design by Contract Tool, http://www.reliable-systems.com/tools/iContract/iContract.htm.

[Javadoc, 1999a] Sun Microsystems, Javadoc homepage, http://java.sun.com/products/jdk/javadoc/.

[Javadoc, 1999b] Sun Microsystems, "How to write doc comments for Javadoc," http://java.sun.com/products/jdk/javadoc/writingdoccomments.html.

[JFC, 1999] *Java Foundation Classes*, JDK Documentation. Javasoft, 1999.

[Johnson et al., 1988] R. Johnson & B. Foote, "Designing reusable classes," *Journal of Object-Oriented Programming*. vol. 1, no. 5, pp. 22–35, 1988.

[Kompanek et al., 1996] A. Kompanek, A. Houghton, H. Karatassos, A. Wetmore, & B. Bruegge, "JEWEL: A distributed system for emissions modeling," *Conference for Air and Waste Management*, Nashville, TN, June 1996.

[Liskov, 1986] B. Liskov & J. Guttag, *Abstraction and Specification in Program Development*. McGraw-Hill, New York, 1986.

[Meyer, 1997] Bertrand Meyer, *Object-Oriented Software Construction*, 2nd ed. Prentice Hall, Upper Saddle River, 1997.

[OMG, 1995] Object Management Group, *The Common Object Request Broker: Architecture and Specification*. Wiley, New York, 1995.

[OMG, 1998] Object Management Group, *OMG Unified Modeling Language Specification*. Framingham, MA, 1998, http://www.omg.org.

[Rational, 1998] Rational Software Corp., *Rationale Rose 98: Using Rose*. Cupertino, CA, 1998.

[Rumbaugh et al., 1991] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, & W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[Schmidt, 1997] D. C. Schmidt, "Applying design patterns and frameworks to develop object-oriented communication software," *Handbook of Programming Languages*, vol. 1, Peter Salus (ed.), MacMillan Computer, 1997.

[Weinand et al., 1988] A. Weinand, E. Gamma, & R. Marty. ET++ – An object-oriented application framework in C++. In *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, San Diego, CA, September 1988.

[Wilson & Ostrem, 1999] G. Wilson & J. Ostrem, *WebObjects Developer's Guide*, Apple, Cupertino, CA, 1998.