Appendix B

Constructing the Software Solution

Objectives

Upon completion of this appendix, you should be able to:

- Define a Java technology package hierarchy for the Solution model
- Recognize Java technology code that satisfies the elaborated Domain model

Process Map

This appendix describes a step in the Implementation workflow: implementing the solution code. Figure B-1 shows the activities and artifacts discussed in this appendix.

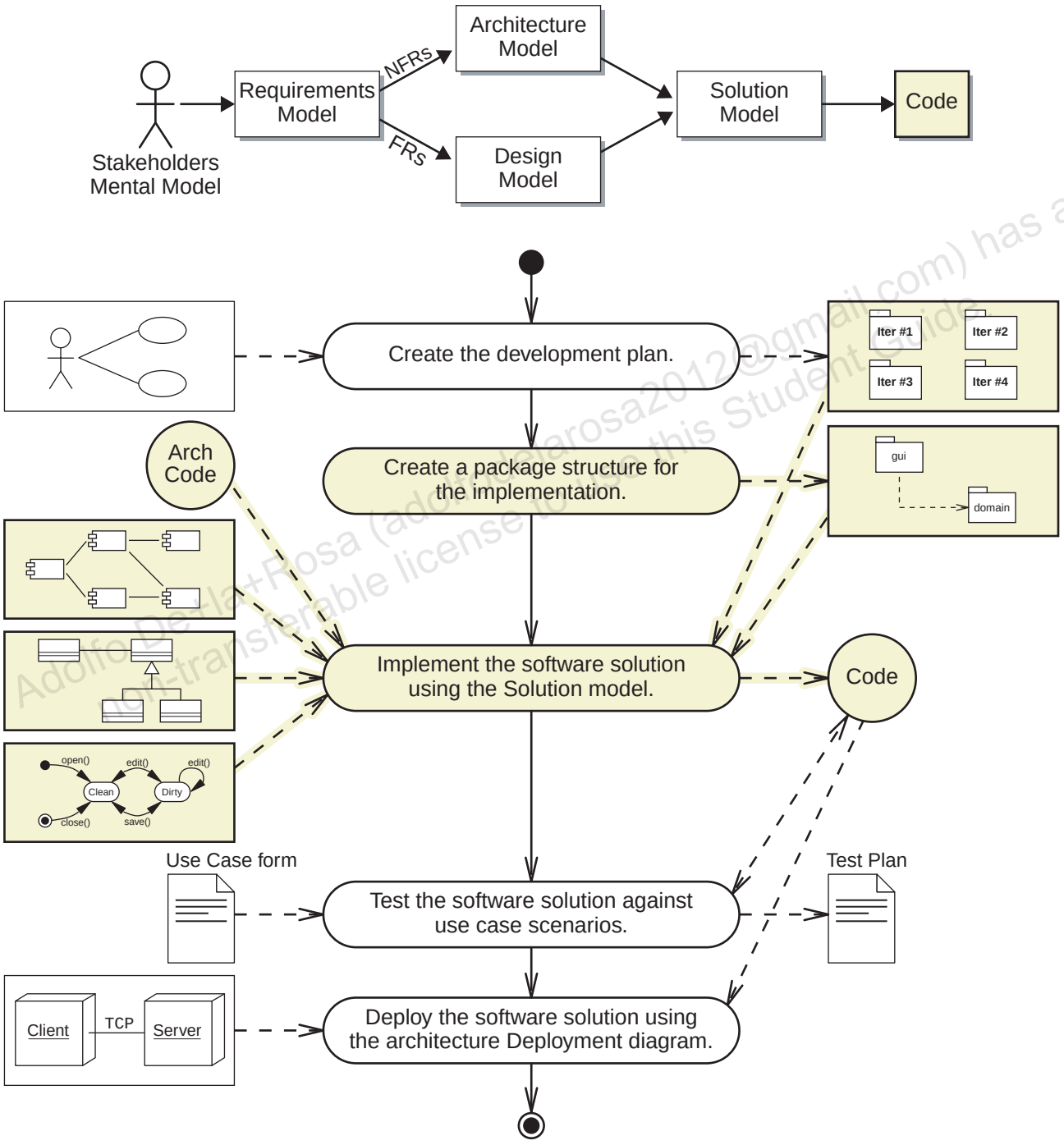


Figure B-1 Implementation Workflow Process Map

Defining a Package Structure for the Solution

This section describes the concepts and principles behind defining a useful package structure for a software solution. This section is grouped into the following subsections:

- Representing Java technology packages in UML
- Applying the principles of package dependencies
- Isolating subsystems and frameworks into packages
- Developing a package structure that satisfies the Solution model

Using UML Packages

A UML package is a modeling construct for grouping other UML diagrams or elements.

A UML package can also represent a Java technology package (a group of related classes). Figure B-2 illustrates an example of a UML package that models a Java technology package. The code in the annotation demonstrates the use of the `package` statement to declare that the `MyClass` class is part of the `com.bayview` package.

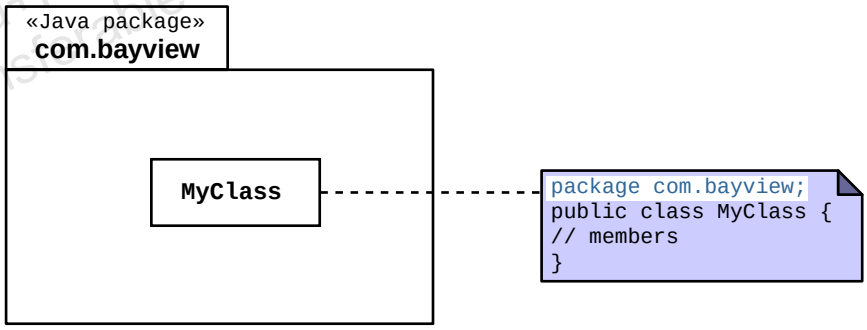


Figure B-2 Example Code for a Java Technology Package

Applying Package Principles

This subsection introduces the following package principles:

- Package dependencies
- Common Closure Principle (CCP)
- Acyclic Dependency Principle (ADP)

- Stable Dependencies Principle (SDP)
- Stable Abstractions Principle (SAP)

All of these principles are just guidelines for good design and that they require trade-offs. These principles should not be considered unbreakable rules. If your project has a requirement that leads you to break one or more of these guidelines, then do so.

Simple Package Dependencies

> "If changing the contents of a package P2 (might) impact the contents of another package P1, then we can say that P1 has a package dependency on P2." (Knoernschild page 24)
>
> All package principles are defined in terms of package dependencies. Figure B-3 illustrates a simple package dependency. Package `P1` depends on package `P2` because the `Client` class (in `P1`) uses the `Service` class (in `P2`). The code in the annotations demonstrate the use of the `import` statement to allow a class definition in one package to gain access to a class in another package.

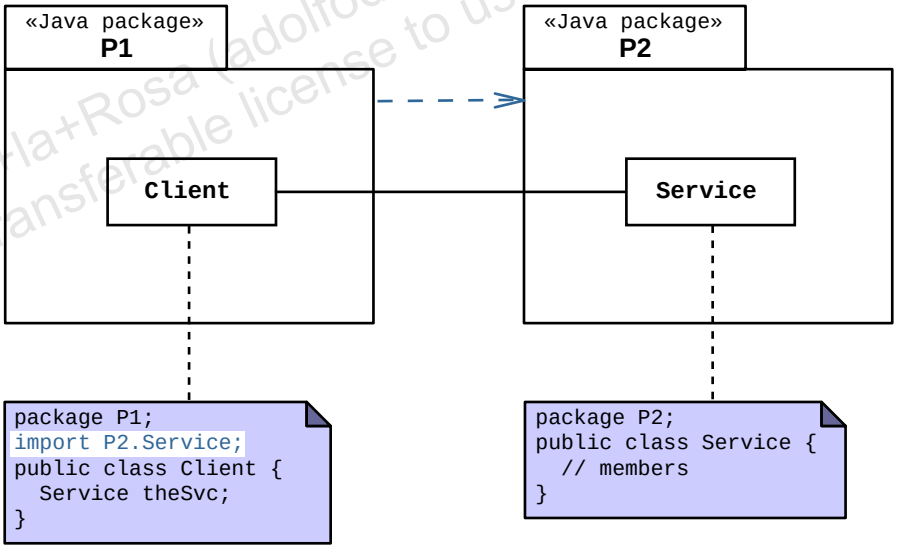


Figure B-3 An Example Package Dependency

Common Closure Principle

> "Classes that change together, belong together." (Knoernschild page 26)

CCP recognizes package cohesion, emphasizing the overall services offered by the entire package.

CCP tends to group packages by Separation of Concerns. For example:

- Grouping web components for a single use case together

  Web components make up the Boundary elements for use cases that require a Web application. For example, permitting Bay View customers to make reservations online. If there are changes to the "Manage a Reservation Online" use case, then it is likely that several web components for that use case will need to change. Therefore, CCP suggests that all of the web components for this use case be grouped together.

- Grouping related Domain classes together

  For small systems, all of the Domain entities could be grouped into a single package. Large systems might have groups of entities that are highly coupled. Using CCP, these groups would be separated into their own packages.

  For example, in the Hotel Reservation System, the `Reservation`, `Customer`, `Room`, and `Payment` classes might all be grouped together. Whereas the classes that support the catering side of the business might be grouped into a different package.

Acyclic Dependency Principle

> "The dependencies between packages must form no cycles." (Knoernschild page 27)

Suppose that a class, `AClass1` in package `A`, uses two other classes: `AClass2` in package `A` and `BClass1` in package `B`. And further suppose that `BClass1` uses `AClass2`. Figure B-4 illustrates this situation. `AClass1` is tightly coupled with `BClass1` and `BClass1` is tightly coupled with `AClass2`. This implies that package `A` is tightly coupled with package `B` and package `B` is tightly coupled with package `A`. This coupling forms a cycle in the package dependency diagram in Figure B-4 and is a violation of the ADP.

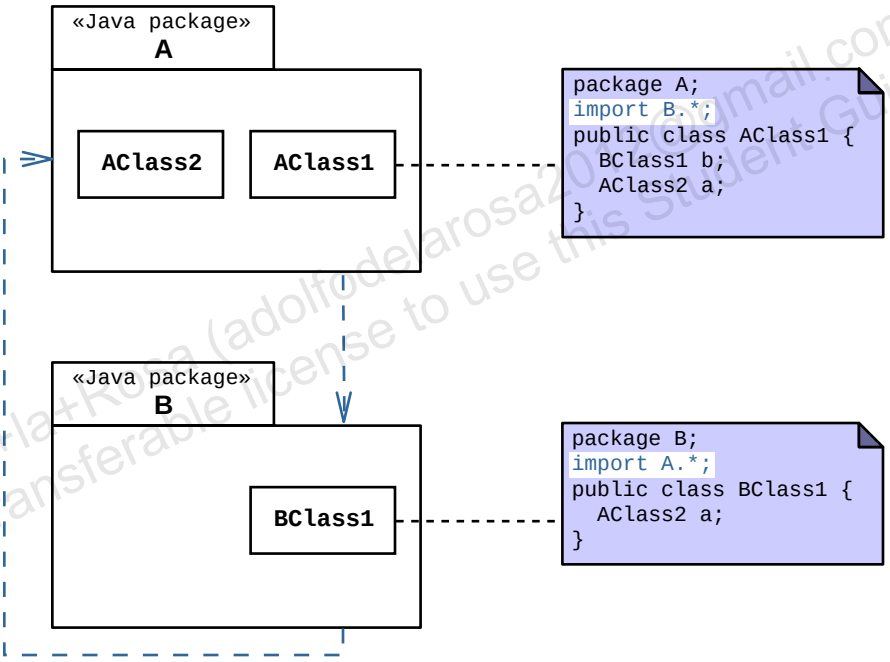


Figure B-4 An Example Violation of the ADP

Violations to ADP hinder the effectiveness of the Common Closure Principle because changes in one package might affect changes in all of the packages in the cycle with the original package. The goal is to attempt to isolate the effects of changes between packages. Therefore, you should eliminate package dependency cycles. One way to do that would be to lump all of the packages in a cycle into a single package, but this approach has large consequences.

Alternatively, you could refactor the set of classes in one (or more) packages in the cycle to eliminate the cycle. For example, you could move AClass2 into a different package, say A′ (pronounced A-prime). By doing this the cycle between package A and package B is broken. Now package A depends on packages B and A′; and package B depends only on package A′ (not package A). Figure B-5 illustrates this refactoring.

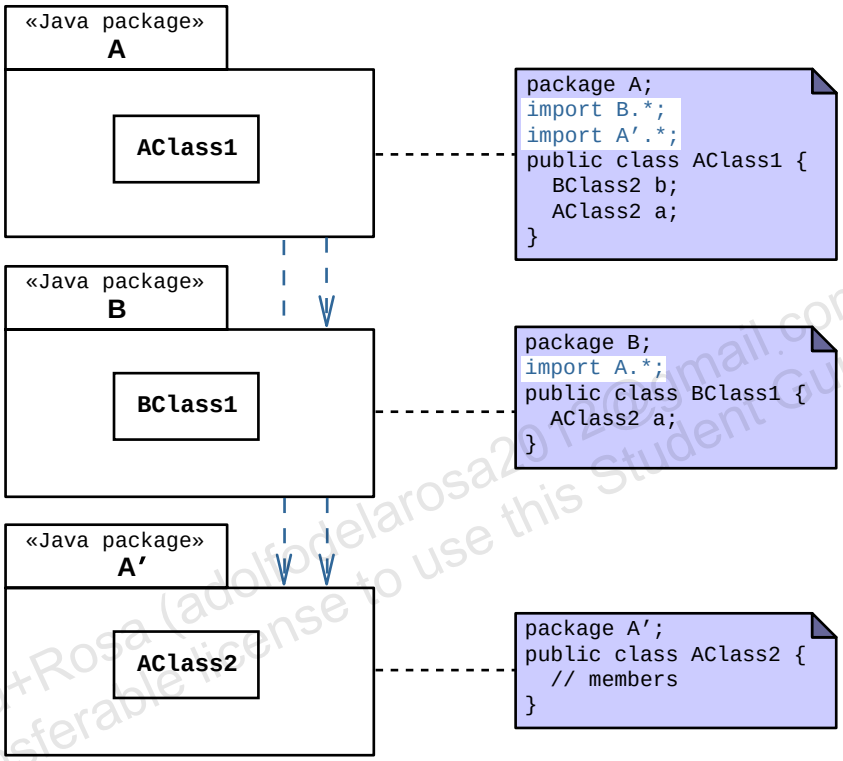


Figure B-5 An Example Refactoring to Achieve ACP

Stable Dependencies Principle

"Depend in the direction of stability." (Knoernschild page 29)

SDP is an extremely useful and straightforward principle. It states that a given package should depend on more stable packages. For example, the most stable parts of a system tend to be the Domain entities because these do not change very often. However, business services and especially user interfaces that implement use cases tend to be less stable because business processes and UI designs (especially for the Web) tend to change frequently. Figure B-6 shows an example of SDP.

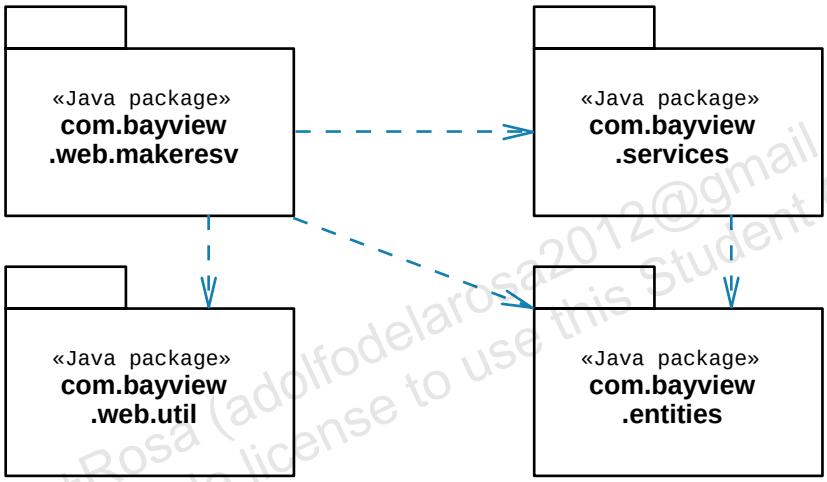


Figure B-6 An Example of the SDP for the Hotel Reservation System

Stability means a component is fixed, permanent, and unvarying.

- Business services and entities tend to be stable.
- Entities tend to be more stable than services.
- UI components tend to change.
- Utilities tend to be stable.

SDP (and CCP) compliment the Tiers partitioning of SunTone Architecture Methodology:

- Components closer to the resource tier tend to be more stable.
- Components closer to the client tier tend to be less stable.

Stable Abstractions Principle

> "Stable packages should be abstract packages." (Knoernschild page 31).

This principle is related to the Dependency Inversion Principle for classes and components. The best example of the SAP is that you can develop a package that holds all of the interfaces for the business services of your system. The UI classes depend on interfaces of the services (especially in a distributed system); therefore, you should create a package that holds the set of service interfaces. It is also a common practice to package the concrete service classes (that implement the service interfaces) into a separate package. Figure B-7 illustrates an example of this package dependency structure.

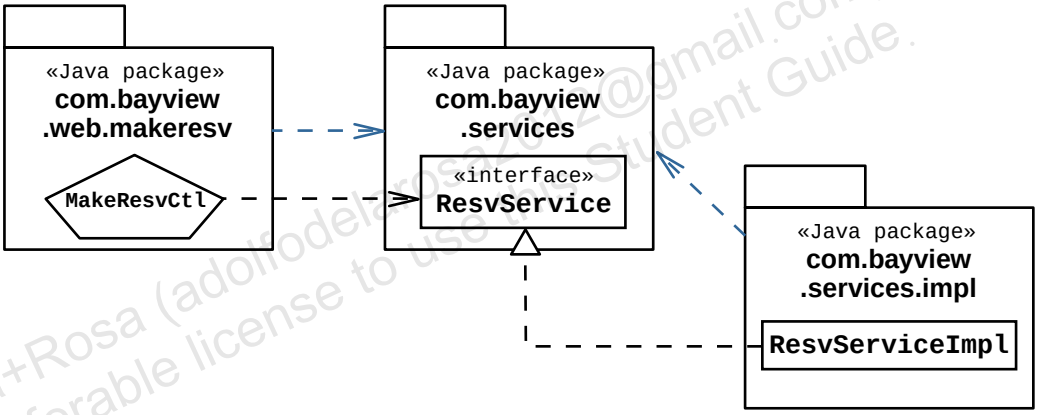


Figure B-7 An Example of the SAP for the Hotel Reservation System

The following are a few facts about the Stable Abstractions Principle:

- Less stable packages should depend on abstractions within another package, rather than concretions.
- Abstract classes and interfaces tend to change less frequently than their implementation classes.
- The formula for determining abstractness of a package is:

$$A = \frac{Na}{Nc}$$

Where, *Na* is the number of abstract classes and interfaces in the package and *Nc* is the total number of all classes and interfaces.

Isolating Subsystems and Frameworks

> A subsystem is "A collection of modules, some of which are visible to other subsystems and other of which are hidden." (Booch OOAD page 519)
>
> A framework is "A collection of classes that provide a set of services for a particular domain; a framework thus exports a number of individual classes and mechanisms that clients can use or adapt." (Booch OOAD page 514)

Subsystems and frameworks are usually placed in their own package or a set of packages. The I/O subsystem in the J2SE platform is grouped in the `java.io` (and now `java.nio`) package. The Abstract Window Toolkit (AWT) and Swing frameworks are in the `java.awt` and `javax.swing` packages, respectively.

Only public classes or interfaces are visible outside of the package. Therefore, subsystems and frameworks can expose only the classes that are required by the client of these subsystems by making those classes public. All other classes should be kept package-private.

Developing a Package Structure for the Solution Model

From the package dependency principles, the following are a few guidelines for creating a package structure:

- Group components of each tier into their own package.

  For example, the business services and entities should be in their own package. Likewise, the UI and DAO components should be in their own package.

  This recommendation is supported by the CCP and the SDP.

- Within a tier, group components that change together into their own package.

  For example, the client-tier components for a web application might be grouped together.

  Another example is to group all of the DAO classes for a given data storage mechanism together. So if you need an XML and DBMS data storage mechanisms, then the DAO classes for each of these would be in their own package.

- Group interfaces of a set of services into a package and their implementation into another package.

  This structure is important for distributed systems because the Client and Presentation tier components are designed to work with the service interfaces rather than with the service implementation classes.

- Isolate subsystems into their own packages.

  As mentioned in the previous section, subsystems should have their own package because the classes in a subsystem tend to change together. This guideline is an application of the CCP.

The following figures illustrate these guidelines relative to the Hotel Reservation System. Figure B-8 shows the packages of the Application layer of the system with their dependencies. Notice that the dependencies are acyclic and tend toward stability.

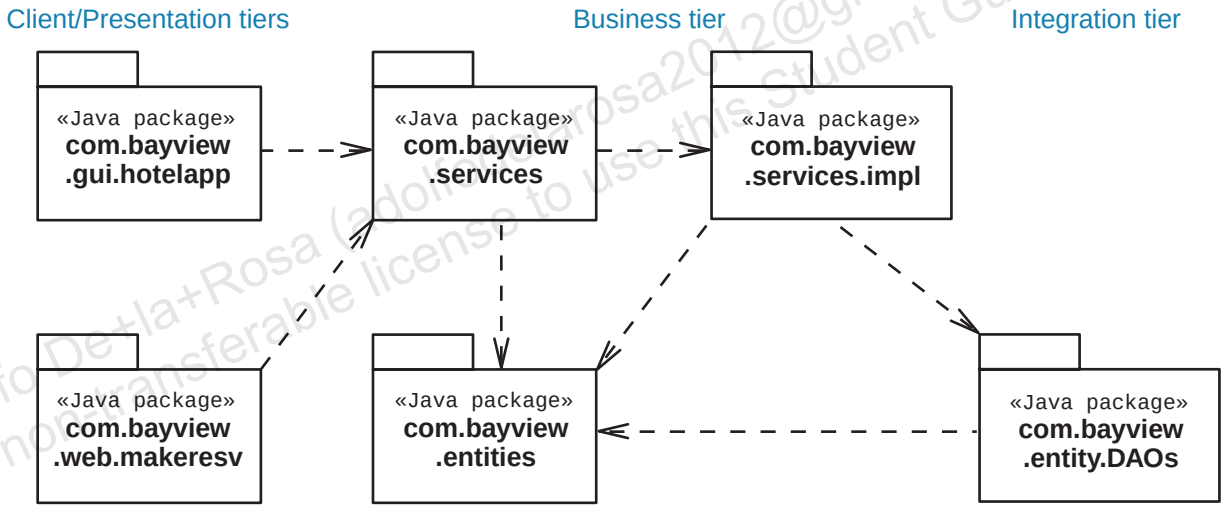


Figure B-8 HRS Package Dependencies Grouped by Tiers

Figure B-9 shows the dependencies between the packages of the Application layer and the packages of the Virtual Platform. For example, the web components in `com.bayview.web.createresv` depend on the servlet API package (`javax.servlet.http`).

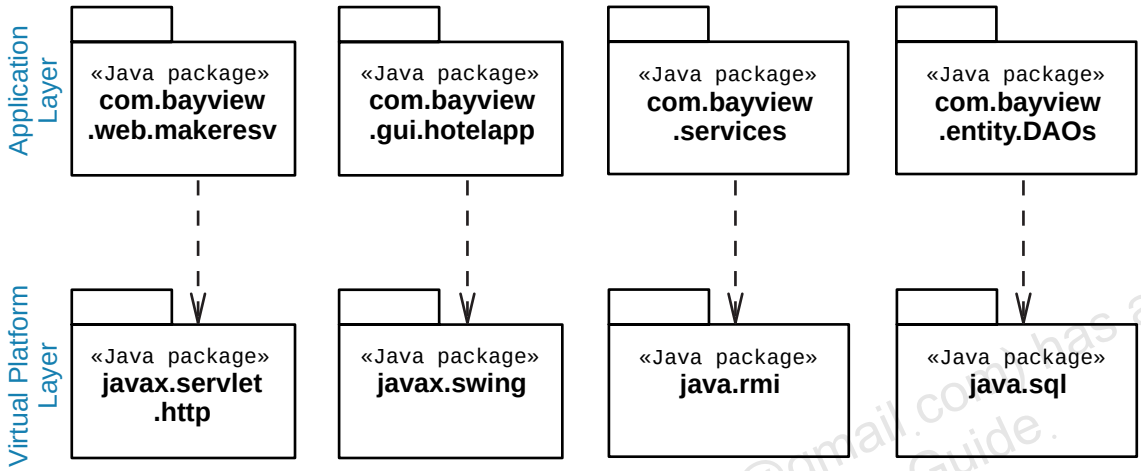


Figure B-9 HRS Package Dependencies Between the Application and Virtual Platform Layers

Mapping the Domain Model to Java Technology Class Code

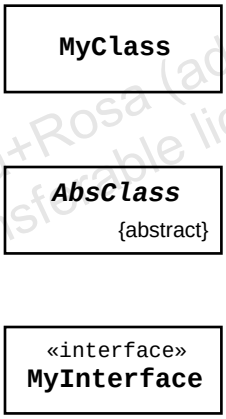
This section describes how to recognize Java technology code that implements the following modeling constructs in a UML Class diagram:

- Type information
- Attributes
- Associations

The following subsections illustrate these modeling constructs.

Type Information

There are three fundamental types of classes: a concrete class, an abstract class, and an interface. Figure B-10 illustrates Java technology code for each of these types.



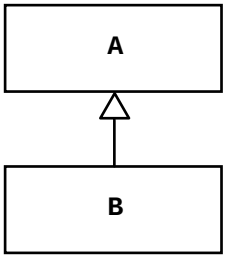

```
public class MyClass {
  // members
}
```

```
public abstract class AbsClass {
  // members
}
```

```
public interface MyInterface {
  // members
}
```

Figure B-10 Java Technology Code for Three Types of Classes

An additional element of type information is the superclass of a given class. The `extends` keyword is used in Java technology code to encode this type of class relationship. Figure B-11 shows an example of class inheritance.

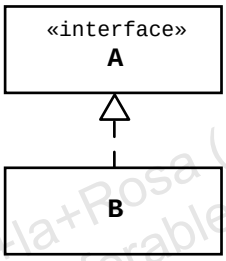



```
public class A {
  // members
}

public class B extends A {
  // members
}
```

Figure B-11 Java Technology Code for Class Inheritance

In Java technology, interfaces are implemented by classes. The `implements` keyword encodes this type of class relationship. Figure B-12 shows an example of interface implementation.




```
public interface A {
  // members
}

public class B implements A {
  // members
}
```

Figure B-12 Java Technology Code for Interface Implementation

Attributes

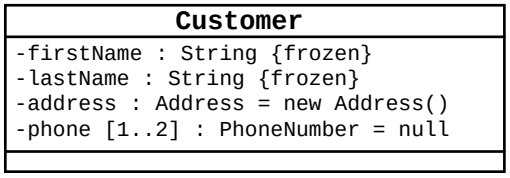
The UML syntax for an attribute declaration is:

```
[visibility] name [multiplicity] [: type] [= init-
value] [{property-string}]
```

The Java technology language syntax for an attribute declaration is:

```
[modifiers] type name [= init-value];
```

Figure B-13 illustrates an example class with attributes.




```
public class Customer {
  private final String firstName;
  private final String lastName;
  private Address address = new Address();
  private PhoneNumber[] phone = null;
}
```

Figure B-13 Java Technology Code for Attribute Declarations

The modifiers of an attribute declaration are determined by the following UML mapping rules:

- Visibility visibility maps to the attributes accessibility.

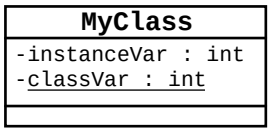
  In UML, visibility maps to the attributes accessibility. Table B-1 shows there are four predefined visibility values.

Table B-1 Accessibility Keywords

UML Symbol	Accessibility Keyword
+	`public`
#	`protected`
~	(no keyword)
-	`private`

- The `{frozen}` constraint maps to the `final` modifier.
- If the multiplicity of the attribute is greater than one, then the type of the attribute is either an array or a collection.
- Class scope can be declared.

An attribute can be declared to have class scope. In the UML, members (both attributes and methods) having class scope are represented by marking the member declaration with an underline. This maps to the `static` modifier. Figure B-14 illustrates an example.

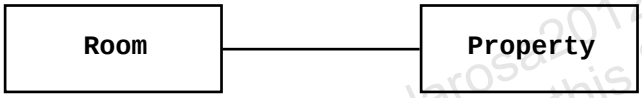



```
public class MyClass {
  private int instanceVar;
  private static int classVar;
}
```

Figure B-14 Java Technology Code for Class Scoped Attributes

Associations

Associations between classes are usually implemented as an attribute. Figure B-15 illustrates an example association.




```
public class Room {
  private Property property;
}
```

Figure B-15 Java Technology Code for an Association

There are several important features of associations:

- Navigation
- Multiplicity
- Qualified associations
- Aggregation and composition

Navigation

The navigation arrow specifies which class must implement the association and its access method.

- Unidirectional

  A unidirectional association indicates that the association navigates in only one direction. Figure B-16 shows an example where the `Room` class navigates to the `Property` class; therefore, the `Room` class must

include an attribute that maintains the association to an object of the `Property` class. Moreover, there is no attribute in the `Property` class to navigate to a `Room` object.

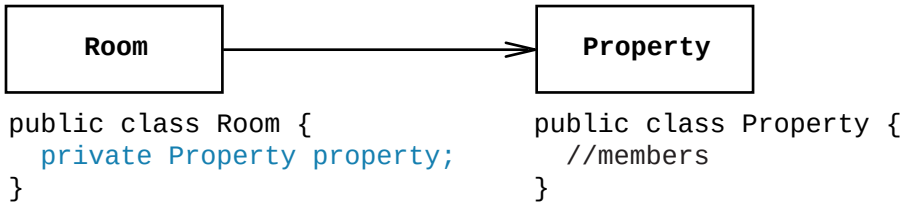


Figure B-16 Java Technology Code for a Unidirectional Association

- Bidirectional

  A bidirectional association indicates that the association can navigate in both directions. Figure B-17 shows an example where the `Room` class navigates to the `Property` class and vice versa. Therefore, the `Room` class must include an attribute that maintains the association to an object of the `Property` class. Moreover, the `Property` class must include an attribute that maintains the association to an object of the `Property` class must include an attribute that maintains the association to an object of the `Room` class.

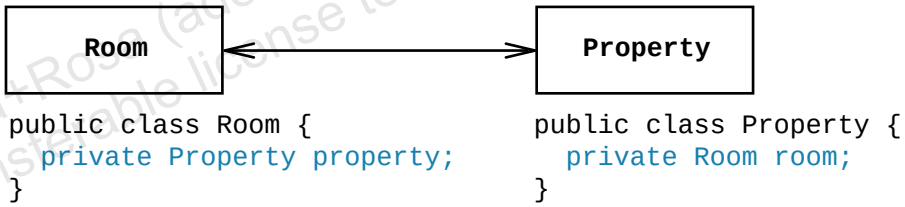
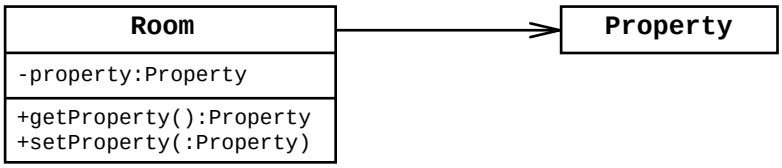


Figure B-17 Java Technology Code for a Bidirectional Association

Association Methods

Proper encapsulation also recommends keeping the association instance variable private and providing public access and mutator methods. In the case of a one-to-one association it is sufficient to supply simple accessor (`get`) and mutator (`set`) methods. Figure B-18 illustrates an example of this.

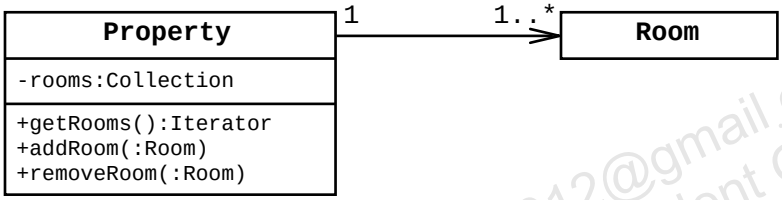



```
public class Room {
  private Property property;
  public Property getProperty() {
    return property;
  }
  public void setProperty(Property p) {
    property = p;
  }
}
```

Figure B-18 Java Technology Code for Association Methods

Association Multiplicity

When the multiplicity of an association is one-to-many or many-to-many, this adds significant complexity to the code that you must use to implement the association. There are many techniques for encoding these types of associations, such as using arrays. However, the most generic mechanism is to use the Java technology Collections API. For example, the `Property` class has a one-to-many association with the `Room` class. This can be represented with an attribute called `rooms` of type `Collection`. In this example, the implementation of that collection is a `HashSet`; you could also have used a list implementation class if the order of the rooms is important. Figure B-19 illustrates this example.




```
import java.util.*;
public class Property {
  private Collection rooms = new HashSet();
  public Iterator getRooms() {
    return rooms.iterator();
  }
  public void addRoom(Room r) {
    rooms.add(r);
  }
  public void removeRoom(Room r) {
    rooms.remove(r);
  }
}
```

Figure B-19 Java Technology Code for a One-to-Many Association

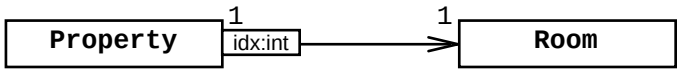
To access the collection of rooms in a property, you would need to encode a method to return an `Iterator` which enables you to iterate through each item in the collection. In Figure B-19, this is implemented by the `getRooms` method.

To add or remove rooms from the collection, you would need to encode two methods: `addRoom` and `removeRoom`. These methods control access to the association collection.

Qualified Associations

Qualified associations partition a one-to-many (or many-to-many) association into smaller groups. Typically, qualified associations identify a unique element in the collection.

A many-to-many association can be qualified by an integer index. Figure B-20 shows an example. In this example, the rooms of a property object are ordered by an index. A typical implementation would use a `List` attribute.

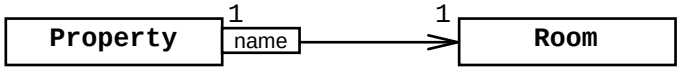



```
import java.util.*;
public class Property {
  private List rooms = new ArrayList();
  public Room getRoom(int idx) {
    return (Room) rooms.get(idx);
  }
  public void addRoom(int idx, Room r) {
    rooms.add(idx, r);
  }
  public void removeRoom(int idx) {
    rooms.remove(idx);
  }
}
```

Figure B-20 Java Technology Code for an Indexed Qualified Association

The accessor and mutator methods must permit the client of this class to access and manipulate the collection by the index. Therefore, the `getRoom` method takes an `idx` parameter to qualify which room is being requested. Similarly, the `addRoom` and `removeRoom` methods would use the index qualifier to specify which room is being added or removed.

A qualified association can also use a non-integer key to perform the lookup operation. This is sometimes called a "symbolic qualifier." Typically, this qualifier is a string value. Figure B-21 shows an example. In this example, the rooms of a property are qualified by the name of the room. A `Map` data structure is usually used to implement a symbolic qualified association.



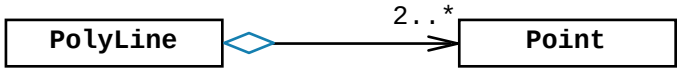

```
import java.util.*;
public class Property {
  private Map rooms = new HashMap();
  public Room getRoom(String name) {
    return (Room) rooms.get(name);
  }
  public void addRoom(Room r) {
    rooms.put(r.getName(), r);
  }
  public void removeRoom(Room r) {
    rooms.remove(r.getName());
  }
}
```

Figure B-21 Java Technology Code for a Symbolic Qualified Association

The accessor and mutator methods must permit the client of this class to access and manipulate the collection by the symbolic qualifier. Therefore, the `getRoom` method takes a name parameter to qualify which room is being requested. Similarly, the `addRoom` and `removeRoom` methods would use the name of the room object to specify which room is being added or removed.

Aggregation

An aggregation is an association that represents a whole-part relationship. The parts of an aggregation may be shared outside the aggregate class. Figure B-22 shows an example. In this example, a `PolyLine` object is constructed from two or more points.



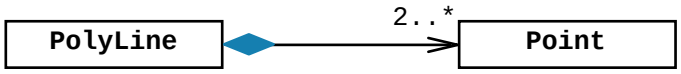

```
import java.util.*;
public class PolyLine {
  private List points = new LinkedList();
  public Iterator getPoints() {
    return points.iterator();
  }
  public void addPoint(Point p) {
    points.add(p);
  }
  public void removePoint(Point p) {
    points.remove(p);
  }
}
```

Figure B-22 Java Technology Code for an Aggregation

The `getPoints` accessor method enables the client class to iterate over each `Point` object. This exposes the `Point` objects to the client. This arrangement is permissible in an aggregation.

Composition

A composition is a specialization of an aggregation in which the custody of the part objects is completely owned by the whole object. To control this custody, the parts of a composition must not be exposed outside the composite class. Figure B-23 shows an example. In this example, the `Point` objects are not exposed to the client class.




```
import java.util.*;
public class PolyLine {
  private List points = new LinkedList();
  // no iterator method
  public void addPoint(int x, int y) {
    points.add(new Point(x,y));
  }
  public void removePoint(int x, int y) {
    DUMMY_POINT.setX(x);
    DUMMY_POINT.setY(y);
    points.remove(DUMMY_POINT);
  }
  private final static DUMMY_POINT = new Point(0,0);
}
```

Figure B-23 Java Technology Code for a Composition

To accomplish composition, there must not be any accessor method that returns the actual `Point` objects. In this example, no access method has been specified. The `addPoint` and `removePoint` methods are responsible for creating the internal `Point` objects. This convention requires that the parameters for these methods must include all of the necessary data to construct the `Point` objects (in this case the `x` and `y` coordinates of the point).

This type of control over the part objects in a composition can be extremely hard to accomplish. Typically, if the part objects are complex themselves, maintaining such tight control is too much of a burden. This is a design trade-off.

Summary

In this appendix, you were introduced to a few implementation considerations. Here are a few important concepts:

- It is important how components and classes are grouped into packages. You have seen a few principles that help guide that process.
- The UML can map closely with most OO languages. In this module, you have seen several common mappings between Class diagrams and Java technology code.