Unauthorized reproduction or distribution prohibited. Copyright© 2020, Oracle and/or its affiliates.

Refining the Class Design Model

Objectives

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

- Refine the attributes of the Domain model
- Refine the relationships of the Domain model
- Refine the methods of the Domain model
- Declare the constructors of the Domain model
- Annotate method behavior
- Create components with interfaces

Adolfo De+la+Rosa (adolfo.delarosa2012@gmail.com) has a non-transferable license to use this Student Guide.

Copyright 2010 Sun Microsystems, Inc. All Rights Reserved. Sun Learning Services, Revision E

Additional Resources

Additional resources – The following reference provides additional information on the topics described in this module:

- The Object Management Group. "Unified Modeling Language (UML), Version 2.2" [`http://www.omg.org/technology/documents/formal/uml.htm`].
- Rumbaugh, James, Jacobson Ivor, Booch Grady. *The Unified Modeling Language Reference Manual (2nd ed).* Addison-Wesley, 2004.
- Larman, Craig. *Applying UML and Patterns (3rd ed).* Prentice Hall, 2005.

Unauthorized reproduction or distribution prohibited. Copyright© 2020, Oracle and/or its affiliates.

Adolfo De+la+Rosa (adolfodelarosa2012@gmail.com) has a non-transferable license to use this Student Guide.

Copyright 2010 Sun Microsystems, Inc. All Rights Reserved. Sun Learning Services, Revision E

Process Map

This module covers the next step in the Design workflow: refining the Design model. Figure 13-1 shows the activity and artifacts discussed in this module.

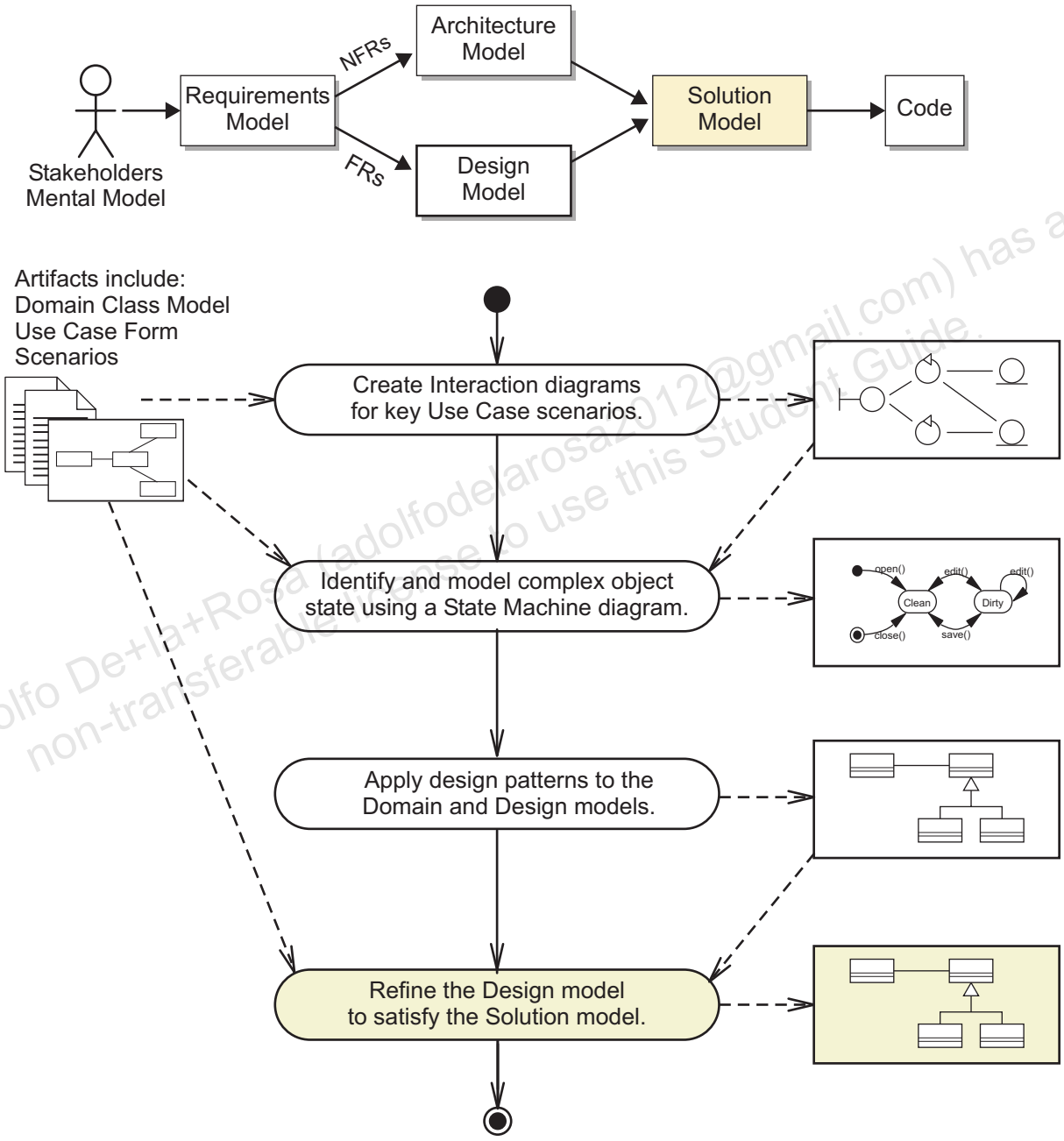

Figure 13-1 Class Design Workflow Process Map

Unauthorized reproduction or distribution prohibited. Copyright© 2020, Oracle and/or its affiliates.

Adolfo De+la+Rosa (adolfodelarosa2012@gmail.com) has a non-transferable license to use this Student Guide.

Copyright 2010 Sun Microsystems, Inc. All Rights Reserved. Sun Learning Services, Revision E

Refining Attributes of the Domain Model

This section discusses several issues around refining the attributes of the domain entities from the Analysis workflow.

Refining the Attribute Metadata

An attribute declaration in UML Class diagrams includes the following:

- Name

  This is the only mandatory feature of an attribute element. This declares the name of the attribute. During Analysis, you can specify attribute names with spaces in them; for example, “first name.” However, during Design you should convert the attribute names to a style that is appropriate to the implementation language; for example, “firstName” would be appropriate using the Java programming language.

- Visibility

  This optional feature defines how visible (or accessible) an attribute is outside of the class. Table 13-1 shows there are four predefined visibility values. The default value is public.

Table 13-1 UML Visibility Indicators

Visibility	Indicator
Public	+
Protected	#
Package private	~
Private	-

- Type

  This is the data type of the attribute. This is often language-specific, but most language support primitive values, such as integers, floating point numbers, and Boolean values, as well as object references.

- Multiplicity

  This specifies how many values might exist for the attribute. By default, the multiplicity is one.

Unauthorized reproduction or distribution prohibited. Copyright© 2020, Oracle and/or its affiliates.

Adolfo De Rosa (adolfo.delarosa2012@gmail.com) has a non-transferable license to use this Student Guide.

Copyright 2010 Sun Microsystems, Inc. All Rights Reserved. Sun Learning Services, Revision E

- Initial value

  This specifies the initial value of this attribute when an object of this class is constructed. This is the default value at initialization, but it can be overridden by a constructor.

- Constraint (one of `changeable`, `addOnly`, or `frozen`)

  This optional feature specifies the characteristic of *changeability* for the attribute at runtime. The value `changeable` means that the attribute can be changed at any time. The value `frozen` means that the attribute must be set during object instantiation and cannot be reset. The value `addOnly` applies to attributes that have a multiplicity greater than one; it means that values can be added to this attribute but no values can be removed.

  These constraints are defined by the UML specification. However, you can use stereotypes to apply additional semantics to the attribute.

The UML syntax for an attribute declaration is as follows:

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

Figure 13-2 illustrates a refinement of the `Customer` class in which the attributes are fully specified. In this design, the first and last names of a customer cannot be changed after the object is created. Also the `phone` attribute supports one or two `PhoneNumber` objects: one for the customer's home and one for the work phone number.

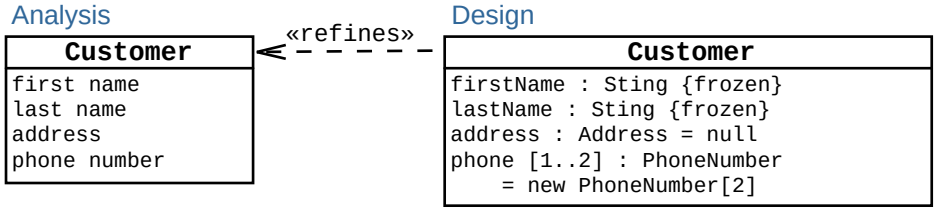

Figure 13-2 Example Refinement of Attributes

Unauthorized reproduction or distribution prohibited. Copyright© 2020, Oracle and/or its affiliates.

Adolfo De+la+Rosa (adolfodelarosa2012@gmail.com) has a non-transferable license to use this Student Guide.

Copyright 2010 Sun Microsystems, Inc. All Rights Reserved. Sun Learning Services, Revision E

Choosing an Appropriate Data Type

Choosing a data type is a trade-off of:

- Representational transparency

  How well does the data type correspond to the programmer's mental model of the attribute?

- Computational time

  How much time is required to convert the data into the representations needed for the user interface or data storage?

- Computational space

  How much space (in bytes) does the data require in memory? This issue is very important for small devices like cellphones and PDAs.

Table 13-2 illustrates a few of the issues in selecting a data type to represent a phone number attribute.

Table 13-2 Choosing a Data Type for a Phone Number

Data Type	Discussion
`String`	This data type might require mapping between the UI representation and the storage (DB) representation.
`long`	This data type conserves space, but might not be sufficient to represent large phone number (such as international numbers).
`PhoneNumber`	A value object is a class that represent the phone data. This data type is representationally transparent, but requires additional coding.
`char` array	This data type is similar to a `String` and adds no value.
`int` array	This data type conserves space, but is not represenationally transparent.

Unauthorized reproduction or distribution prohibited. Copyright© 2020, Oracle and/or its affiliates.

Adolfo De-la-Fresa (adolfodelafresa2012@gmail.com) has a non-transferable license to use this Student Guide.

Copyright 2010 Sun Microsystems, Inc. All Rights Reserved. Sun Learning Services, Revision E

Creating Derived Attributes

In Analysis, you might have an attribute that you know can be derived from another (more stable) source. The canonical example is the calculation of a person's age from their date of birth. A derived attribute is represented with a "/" character in front of the attribute name. Figure 13-3 illustrates this.

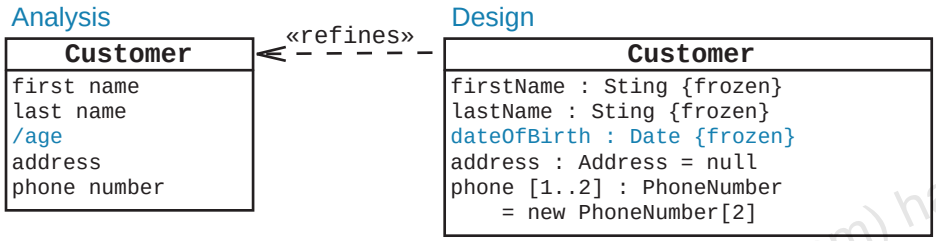

Figure 13-3 Example Derived Attribute

Applying Encapsulation

To use encapsulation, follow these steps:

1. Make all attributes private (visibility).
2. Add public accessor methods for all readable attributes.
3. Add public mutator methods for all writable (non-frozen) attributes.

Figure 13-4 shows an encapsulated class.

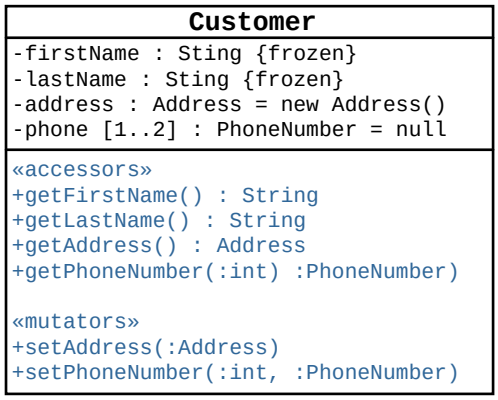

Figure 13-4 Example Use of Encapsulation

Unauthorized reproduction or distribution prohibited. Copyright© 2020, Oracle and/or its affiliates.

Adolfo De+la+Rosa (adolfodelarosa2012@gmail.com) has a non-transferable license to use this Student Guide.

Copyright 2010 Sun Microsystems, Inc. All Rights Reserved. Sun Learning Services, Revision E

Refining Class Relationships

There is no clear distinction between Analysis and Design especially in regards to modeling class relationships.

Design usually addresses these details:

- Type: association, aggregation, and composition
- Direction of traversal (also called navigation)
- Qualified associations
- Declaring association management methods
- Resolving many-to-many associations
- Resolving association classes

Relationship Types

There are three types of relationships:

- Association
- Aggregation
- Composition

These relationships imply that the related object is somehow tied to the original object (usually as an instance attribute).

Note – There is another type of relationship, Dependency, which states that one object uses another object to do some work, but that there is no instance attribute holding that object. For example, if `Class1` has a method that has a parameter of type `Class2`, then you can say that `Class1` depends on `Class2`.

Unauthorized reproduction or distribution prohibited. Copyright© 2020, Oracle and/or its affiliates.

Adolfo De+la+Rosa (adolfodelarosa2012@gmail.com) has a non-transferable license to use this Student Guide.

Copyright 2010 Sun Microsystems, Inc. All Rights Reserved. Sun Learning Services, Revision E

Association

> "The semantic relationship between two or more classifiers that specifies connections among their instances." (OMG page 537)

An association is a relationship in which one object holds a reference to another object as an instance variable. Figure 13-5 illustrates an example association.

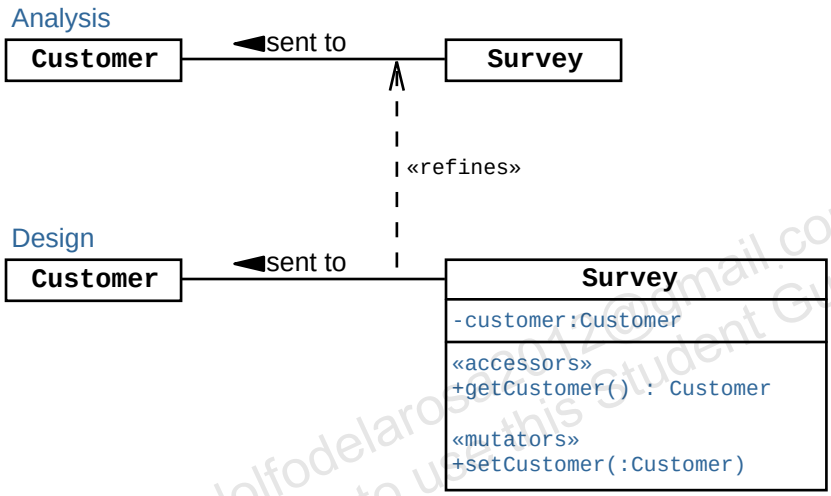

Figure 13-5 An Association Example

Unauthorized reproduction or distribution prohibited. Copyright© 2020, Oracle and/or its affiliates.

Adolfo De+la+Rosa (adolfodelarosa2012@gmail.com) has a non-transferable license to use this Student Guide.

Copyright 2010 Sun Microsystems, Inc. All Rights Reserved. Sun Learning Services, Revision E

Aggregation

> "A special form of association that specifies a whole-part relation between the aggregate (whole) and a component part." (OMG page 537)

The distinction between an association and an aggregation is largely semantic; there is no functional difference between the implementation of these two relationship types. Figure 13-6 illustrates an example aggregation.

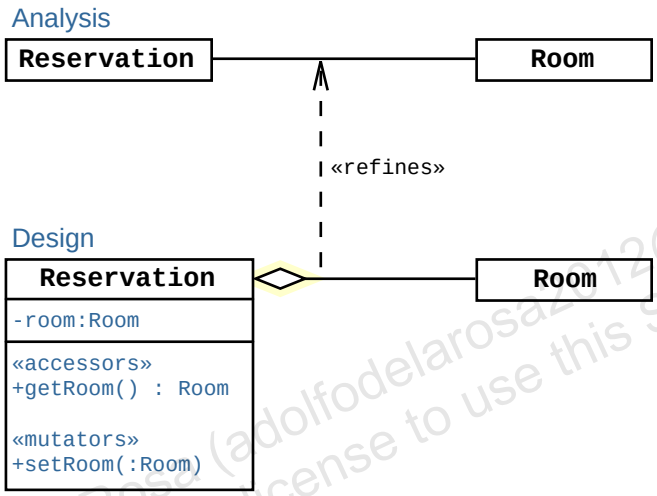

Figure 13-6 An Aggregation Example

Unauthorized reproduction or distribution prohibited. Copyright© 2020, Oracle and/or its affiliates.

Adolfo De la Rosa (adolfodelarosa2012@gmail.com) has a non-transferable license to use this Student Guide.

Copyright 2010 Sun Microsystems, Inc. All Rights Reserved. Sun Learning Services, Revision E

Composition

> "A form of aggregation that requires that a part instance be included in at most one composite at a time, and that the composite object is responsible for the creation and destruction of the parts." (OMG page 540)

The distinction between an aggregation and a composition is semantically similar, but functionally different. In general, the parts of a composition are not accessible outside of the whole object. For example, it might be important to model the payment for a reservation as a composition in which the `Reservation` object is in complete control over the creation (and destruction) of the `Payment` object. To modify the information about a payment, the `Reservation` class must have methods to modify the data within the `Payment` object. Figure 13-7 illustrates this example.

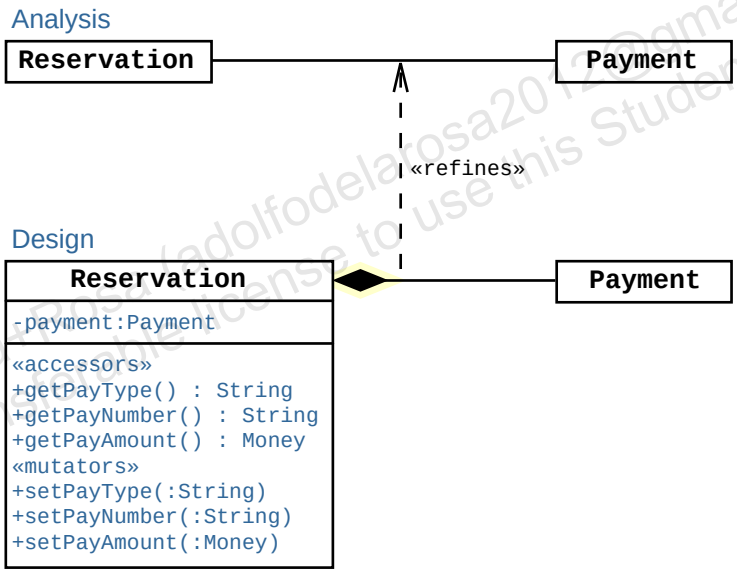

Figure 13-7 A Composition Example

This rule of not exposing the part object outside of the whole object might be impractical in some situations. For example, if the part object has many attributes which need to be modified by a client object outside of the whole object, then it would be impractical to create an accessor and mutator method for every part attribute. Also, if the composition is of many parts, then the accessor and mutator methods would have to use a qualifier to distinguish which part is to be accessed. Use of a qualifier can be difficult. Alternatively, you could expose the part objects to the client object, but that risks having the part object used inappropriately by the rest of the system. This decision is a design trade-off.

Unauthorized reproduction or distribution prohibited. Copyright© 2020, Oracle and/or its affiliates.

Adolfo De+la Rosa (adolfodelarosa2012@gmail.com) has a non-transferable license to use this Student Guide.

Copyright 2010 Sun Microsystems, Inc. All Rights Reserved. Sun Learning Services, Revision E

Navigation

A navigation arrow shows the direction of object traversal at runtime. Figure 13-8 illustrates there are three fundamental forms of navigation.

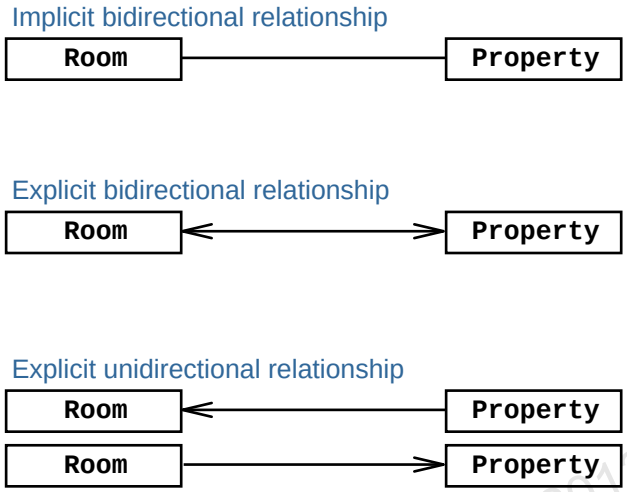

Figure 13-8 The Three Forms of Navigation Indicators

Navigation implies having accessor methods. In a bidirectional relationship both classes will have accessor methods. For example, in the first two cases in Figure 13-8 the `Room` class would have a `getProperty` method and the `Property` class would have a `getRoom` method. In a unidirectional relationship only the class at the end of the relationship would have an accessor method. For example, the third case in Figure 13-8, the `Property` class would have a `getRoom` method, but the `Room` class would *not* have a `getProperty` method.

Unauthorized reproduction or distribution prohibited. Copyright© 2020, Oracle and/or its affiliates.

Adolfo De-Hart Rosa (adolfodelarosa2012@gmail.com) has a non-transferable license to use this Student Guide.

Copyright 2010 Sun Microsystems, Inc. All Rights Reserved. Sun Learning Services, Revision E

Sometimes during analysis, you do not know what direction the software will need to navigate the association. This problem should be resolved during design. For example, the business analyst might not have determined if a `Payment` object could be navigated back to its corresponding `Reservation` object. Therefore, the analyst left the relationship as implicitly bidirectional. However, through Robustness analysis the design team decided that it is only necessary to navigate from the `Reservation` object to the `Payment` object. Figure 13-9 illustrates the appropriate navigation they assign.

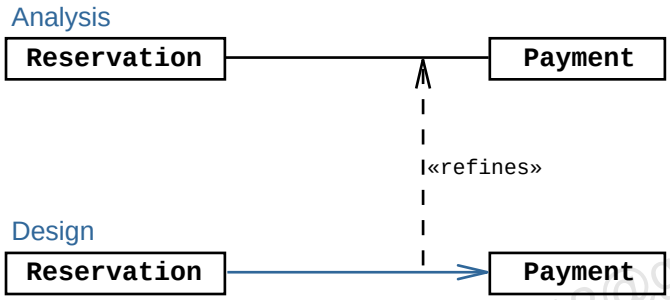

Figure 13-9 An Example Navigation Refinement

Qualified Associations

> "An association attribute or tuple of attributes whose values partition the set of objects related to an object across an association." (OMG page B-15)

In one-to-many or many-to-many associations, it is often useful to model how the system will access a single element in the association. To do this you must define a *qualifier* that provides a datum that uniquely identifies the particular object. For example, a hotel property consists of multiple rooms, and each room has a unique name. Figure 13-10 illustrates this name can be used to qualify the association.

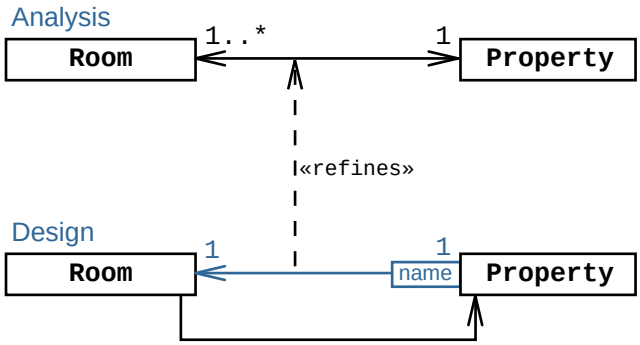

Figure 13-10 An Example Qualified Association

Unauthorized reproduction or distribution prohibited. Copyright© 2020, Oracle and/or its affiliates.

Adolfo De+la+Rosa (adolfodelarosa2012@gmail.com) has a non-transferable license to use this Student Guide.

Copyright 2010 Sun Microsystems, Inc. All Rights Reserved. Sun Learning Services, Revision E

Relationship Methods

Association methods enable the client to access and change associated objects. There are three cases: one-to-one, one-to-many, and many-to-many.

One-to-one relationships use a single instance variable reference. For example, the hotel management would like to send out surveys to customers. So the `Survey` class keeps an association to the `Customer` class using an instance variable and accessor and mutator methods. Figure 13-11 illustrates this example.

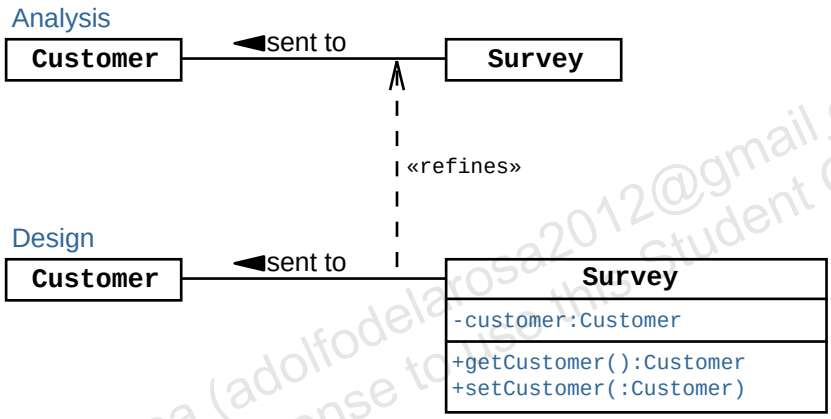

Figure 13-11 Association Methods for a One-to-One Relationship

Unauthorized reproduction or distribution prohibited. Copyright© 2020, Oracle and/or its affiliates.

Adolfo De la Rosa (adolfodelarosa2012@gmail.com) has a non-transferable license to use this Student Guide.

Copyright 2010 Sun Microsystems, Inc. All Rights Reserved. Sun Learning Services, Revision E

One-to-many relationships require the use of collections. For example, a hotel property contains multiple rooms. The `Property` class would store the `Room` objects in a `Collection` object. The accessor method `getRooms` might return an `Iterator` object which would enable the client object to iterate over each element in that collection. The `addRoom` and `removeRoom` mutator methods modify the collection of rooms. Figure 13-12 illustrates this example.

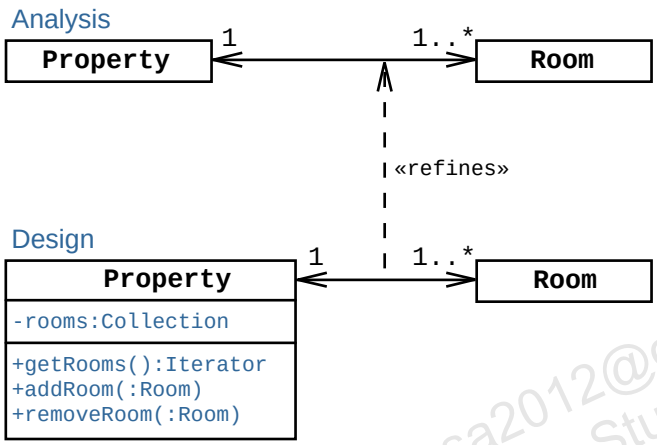

Figure 13-12 Association Methods for a One-to-Many Relationship

Note – The `Collection` and `Iterator` interfaces are built into the J2SE platform. Other languages have similar mechanisms for working with collections (also called containers).

Resolving Many-to-Many Relationships

Managing many-to-many relationships is challenging. This section describes two techniques for simplifying this type of relation.

Unauthorized reproduction or distribution prohibited. Copyright© 2020, Oracle and/or its affiliates.

Adolfo De la Rosa (adolfodelarosa2012@gmail.com) has a non-transferable license to use this Student Guide.

Copyright 2010 Sun Microsystems, Inc. All Rights Reserved. Sun Learning Services, Revision E

First, consider dropping the many-to-many requirement during design. For example, an automotive software system might need to model people and the cars they own. Clearly, in analysis you might decide that any person can own multiple cars and any car might be own by multiple people (at different times). However, if the software you are creating is for a car dealership, then it is not important to record the history of owners for a given car; the system just needs to record who bought the car. In this situation there is no requirement for a many-to-many relationship, so during design the team decides to drop this requirement. Figure 13-13 illustrates this situation.

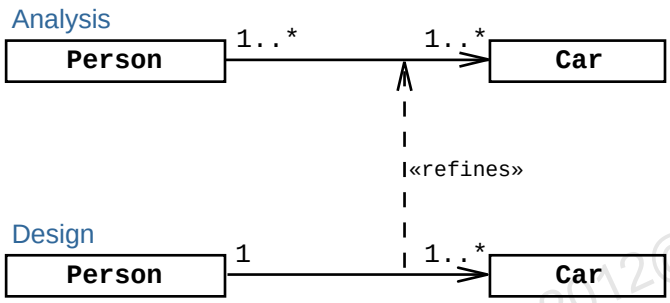

Figure 13-13 Drop the Many-to-Many Relationship

If the many-to-many association must be preserved, you can sometimes add a class in between the two classes that reduces the single many-to-many association to two one-to-many associations. In our person-to-car example, if the software needs to record the complete ownership history for a car, then an additional class, `Ownership`, could be introduced to resolve the many-to-many relationship. Figure 13-14 illustrates this situation.

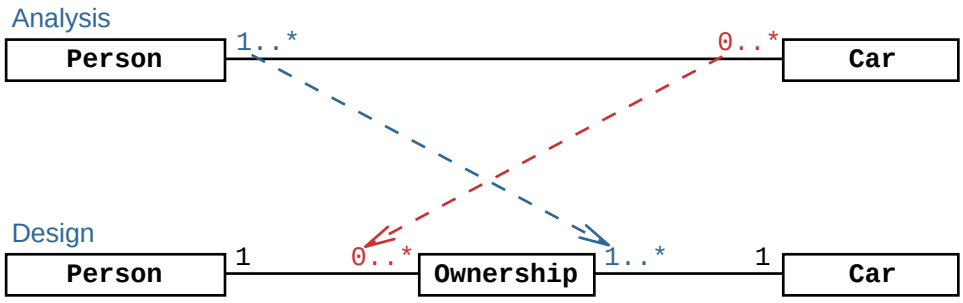

Figure 13-14 Introduce an Intermediate Relationship

Unauthorized reproduction or distribution prohibited. Copyright© 2020, Oracle and/or its affiliates.

Adolfo De-la-Rosa (adolfodelarosa2012@gmail.com) has a non-transferable license to use this Student Guide.

Copyright 2010 Sun Microsystems, Inc. All Rights Reserved. Sun Learning Services, Revision E

Resolving Association Classes

An association class can only exist in the Analysis version of the Domain model. An association class is purely conceptual, there is no programming technique to directly implement this type of concept. Therefore, association classes should be resolved by some construct that can be implemented during Design.

Imagine a soccer application in which the system must model a game played by two teams. Relative to a game, each team would have a score and a flag indicating whether or not that team had to forfeit (for example, not enough players showed up to play the game). During analysis, this situation is nicely modeled with an association class called `TeamScore`. However, this association class must be resolved during design as most OO languages do not support association classes. The developers may decide to move the data for the results into the `Game` class. This solution is feasible because there is a definite number of teams for the game: two. Figure 13-15 illustrates this example.

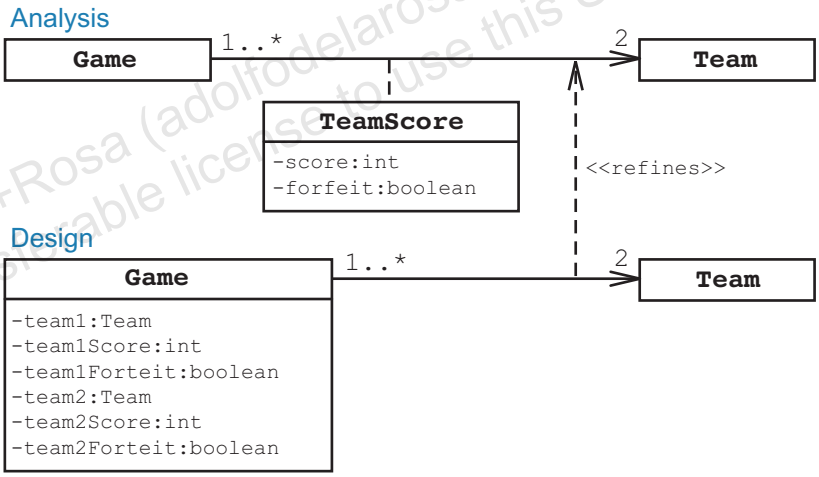

Figure 13-15 Resolving Association Classes: Case One

Unauthorized reproduction or distribution prohibited. Copyright© 2020, Oracle and/or its affiliates.

Adolfo De+la+Rosa (adolfodelarosa2012@gmail.com) has a non-transferable license to use this Student Guide

Copyright 2010 Sun Microsystems, Inc. All Rights Reserved. Sun Learning Services, Revision E

An alternative strategy is to place the association class in between the two primary classes. Figure 13-16 illustrates this.

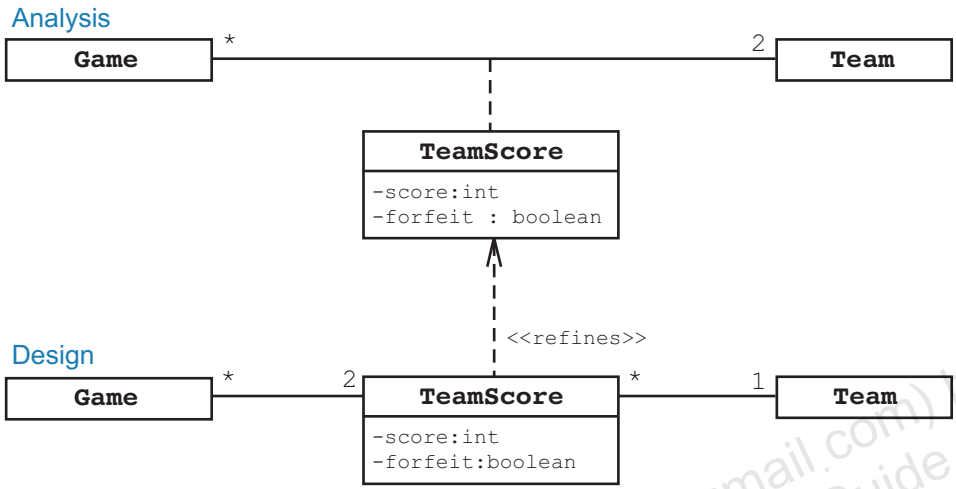

Figure 13-16 Resolving Association Classes: Case Two

Unauthorized reproduction or distribution prohibited. Copyright© 2020, Oracle and/or its affiliates.

Adolfo De+la+Rosa (adolfodelarosa2012@gmail.com) has a non-transferable license to use this Student Guide.

Copyright 2010 Sun Microsystems, Inc. All Rights Reserved. Sun Learning Services, Revision E

Refining Methods

Methods are identified during the following activities:

- CRC analysis, which determines responsibilities
- Robustness analysis, which identifies methods in Service classes
- Design, which identifies accessor and mutator methods for attributes and associations

Other types of methods:

- Object management

  This type of method performs tasks that relate to the management of the object within the programming language. For example, C++ requires explicit memory management through the use of destructor methods. In Java technology, the `equals` and `toString` methods are considered object management methods.

- Unit testing

  You can supply specific methods in your classes to perform unit testing. In Java technology, you could use a `main` method to perform unit tests. Alternatively, you could build separate test classes or use a unit testing framework, such as JUnit.

- Recovery, inverse, and complimentary operations

  Recovery methods provide a means to recover from a complex operation. Such methods are often used by UIs to provide a way to undo an operation. An example of an inverse method and a complimentary method is that if you have a `zoomIn` method, you will probably need a `zoomOut` method and a `zoomByPercentage` method respectively.

The UML syntax for a method declaration is as follows:

```
[visibility] name [({[param] [:type]}*)] [:return-
value] [{constraint}]
```

Unauthorized reproduction or distribution prohibited. Copyright© 2020, Oracle and/or its affiliates.

Adolfo De+la+Rosa (adelarosa2012@gmail.com) has a non-transferable license to use this Student Guide.

Copyright 2010 Sun Microsystems, Inc. All Rights Reserved. Sun Learning Services, Revision E

Suppose that during Robustness analysis, the `ResvService` class needs to support three operations: get a list of properties, create a reservation, and save the reservation. During design, these operations must be transformed into method declarations. Figure 13-17 illustrates this refinement.

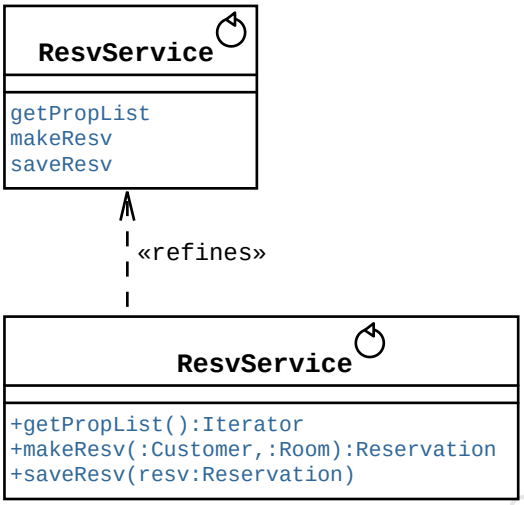

Figure 13-17 Example Refinement of Methods

Unauthorized reproduction or distribution prohibited. Copyright© 2020, Oracle and/or its affiliates.

Adolfo De+la+Rosa (adolfodelarosa2012@gmail.com) has a non-transferable license to use this Student Guide.

Copyright 2010 Sun Microsystems, Inc. All Rights Reserved. Sun Learning Services, Revision E

Annotating Method Behavior

UML annotations can be attached to each method to document the behavior of the method. The behavior may be documented as simple text description, pseudo code, or actual code. Figure 13-18 shows an example of attaching UML annotations to the methods of the `CoolingSate` class.

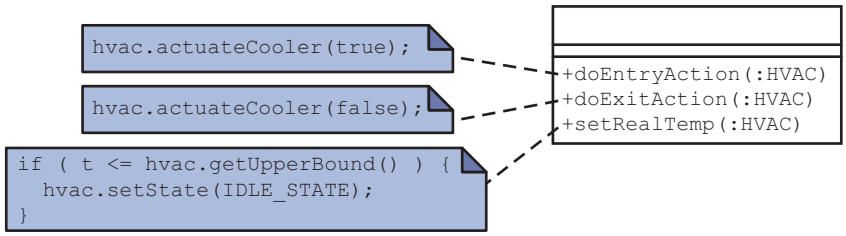

Figure 13-18 Example of UML Method Annotation

Unauthorized reproduction or distribution prohibited. Copyright© 2020, Oracle and/or its affiliates.

Adolfo De+la+Rosa (adolfodelarosa2012@gmail.com) has a non-transferable license to use this Student Guide.

Copyright 2010 Sun Microsystems, Inc. All Rights Reserved. Sun Learning Services, Revision E

Declaring Constructors

Constructors use similar syntax as methods and are similar to methods, except that there is no return type. Also, in Java technology the name of constructors is the same as the class name. Because constructors look very much like methods, it is good practice to separate the constructors from the methods using the «constructor» stereotype. In many languages, you are provided with a default constructor, which take no arguments. However, if you provide any other constructor, this default constructor is not provided. Some frameworks require you to provide a no-argument constructor even if you do not need the constructor in your code. Constructors can be annotated in the same way as methods. Figure 13-19 illustrates this.

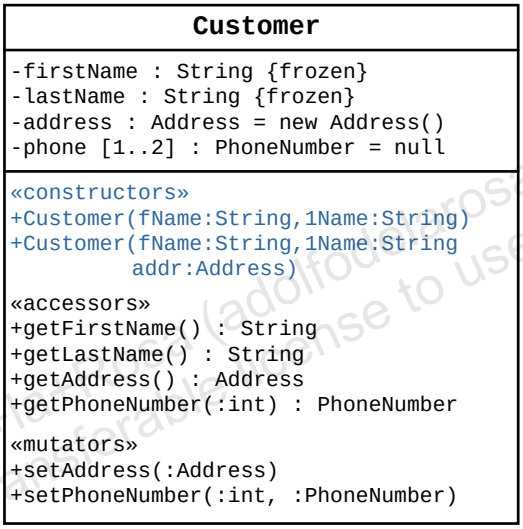

Figure 13-19 Example Constructors

Unauthorized reproduction or distribution prohibited. Copyright© 2020, Oracle and/or its affiliates.

Adolfo De-Rosa (adolfodelarosa2012@gmail.com) has a non-transferable license to use this Student Guide.

Copyright 2010 Sun Microsystems, Inc. All Rights Reserved. Sun Learning Services, Revision E

Reviewing the Coupling and Coherency of your Model

Reviewing Coupling

Ideally, your model should have the lowest coupling while maintaining the FRs and NFRs.

Coupling is a matter of degree. Figure 13-20 shows two versions of a lending library system—one with very high coupling and the other with low coupling. In the Low Coupling version, it would be quick and easy to access the `Book` class from the `Loan` class, but slower and more complex to access the `Loan` class from the `Book` class. In the Very High Coupling version, the access would be quick and easy in both directions. However, the code to add a loan would be more complex and slower as both `Book` and `Loan` will need to be updated. In addition, if adding a loan needs to be performed atomically in a multithreaded system, it would be more complex and might have a negative impact on performance.

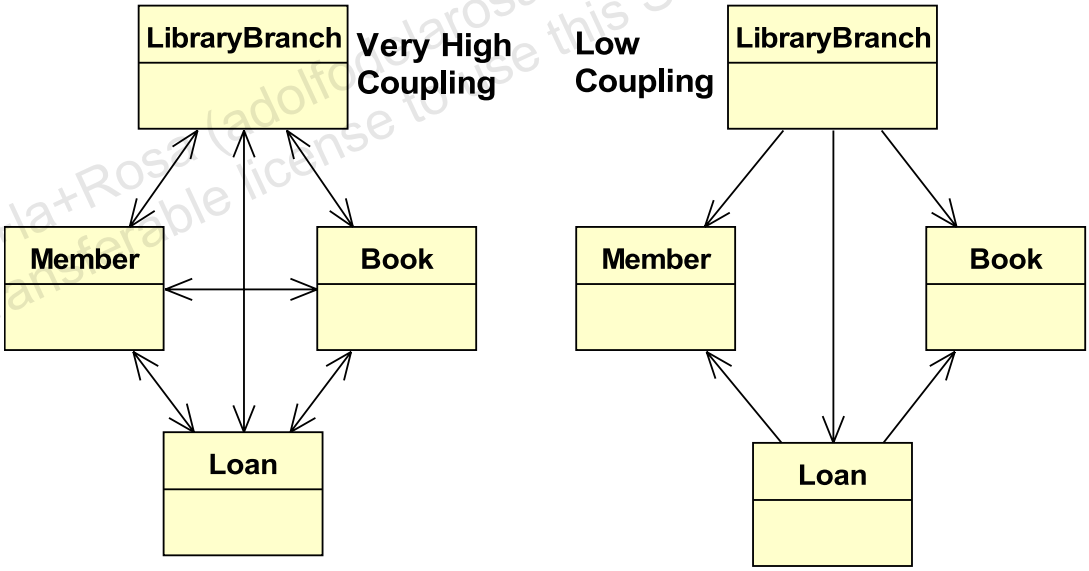

Figure 13-20 Example Comparing High and Low Coupling

Note – The low coupling shown in the figure might not be the lowest coupling possible.

Unauthorized reproduction or distribution prohibited. Copyright© 2020, Oracle and/or its affiliates.

Adolfo De-la+Rosa (adolfo.delarosa2012@mail.bcit.ca) has a non-transferable license to use this Student Guide

Copyright 2010 Sun Microsystems, Inc. All Rights Reserved. Sun Learning Services, Revision E

Reviewing Cohesion

Ideally, your model should have highly cohesive components.

Cohesion is a matter of degree; however, more cohesive components tend to be easier to understand, code, test, and adapt to future requirements. Figure 13-21 shows an example of a lending library system. In the Lower Cohesion version, the member and loan details are in the same class, whereas the member and loan details have been separated in the Higher Cohesion version.

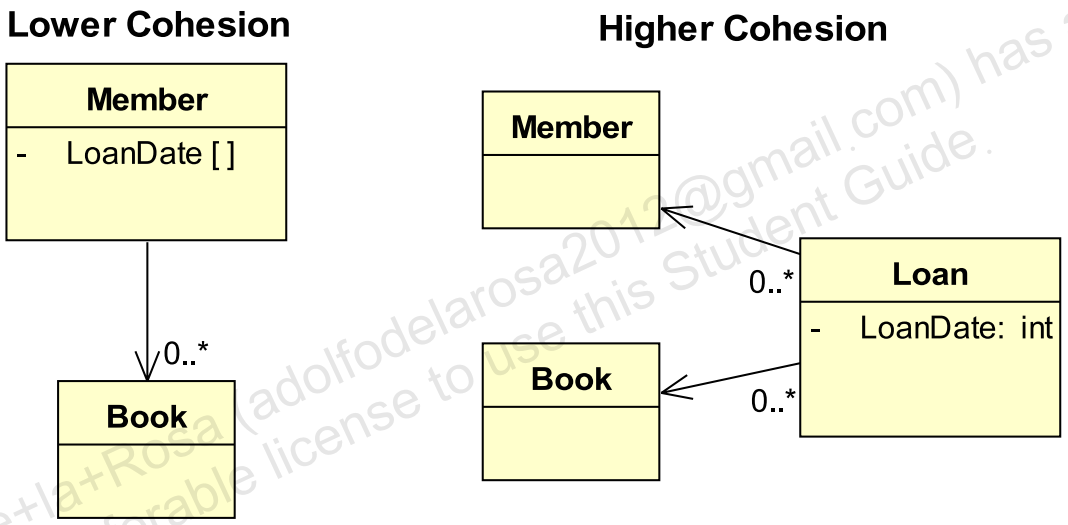

Figure 13-21 Example Comparing High and Low Cohesion

Unauthorized reproduction or distribution prohibited. Copyright© 2020, Oracle and/or its affiliates.

Adolfo De+la+Rosa (adolfodelarosa2012@gmail.com) has a non-transferable license to use this Student Guide.

Copyright 2010 Sun Microsystems, Inc. All Rights Reserved. Sun Learning Services, Revision E

Creating Components with Interfaces

Classes can be grouped into cohesive components with well-defined provided and required interfaces. Figure 13-22 shows an example of a lending library system where all the classes related to membership are grouped into the `Membership` component. The `MemberService` class is split into the `MemberService` interface, which is shown in the diagram as a provided interface, and the `MemberServiceImp` class, which contains the implementation and is hidden inside the component.

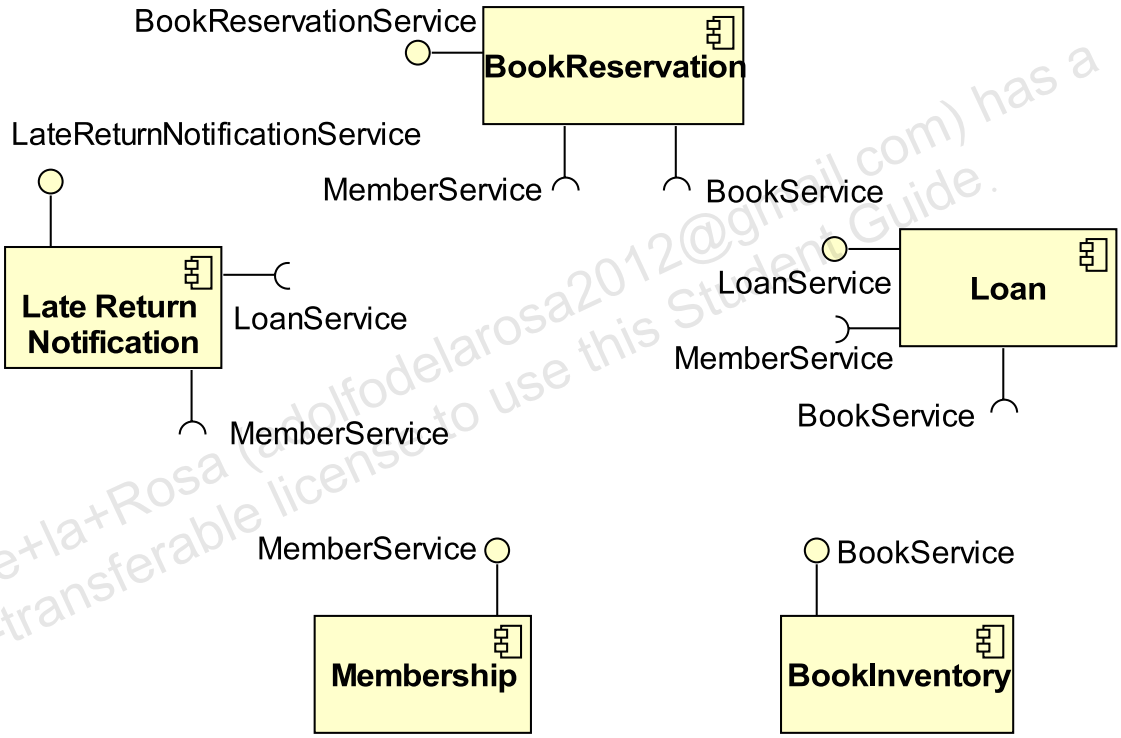

Figure 13-22 Example of Components with Required and Provided Interfaces

Unauthorized reproduction or distribution prohibited. Copyright© 2020, Oracle and/or its affiliates.

Adolfo De+la+Rosa (adolfodelarosa2012@gmail.com) has a non-transferable license to use this Student Guide.

Copyright 2010 Sun Microsystems, Inc. All Rights Reserved. Sun Learning Services, Revision E

You may show dependency arrows in the previous example. However with the required interface notation, these arrows are often redundant. Figure 13-23 shows dependency arrows between the required and provided interfaces.

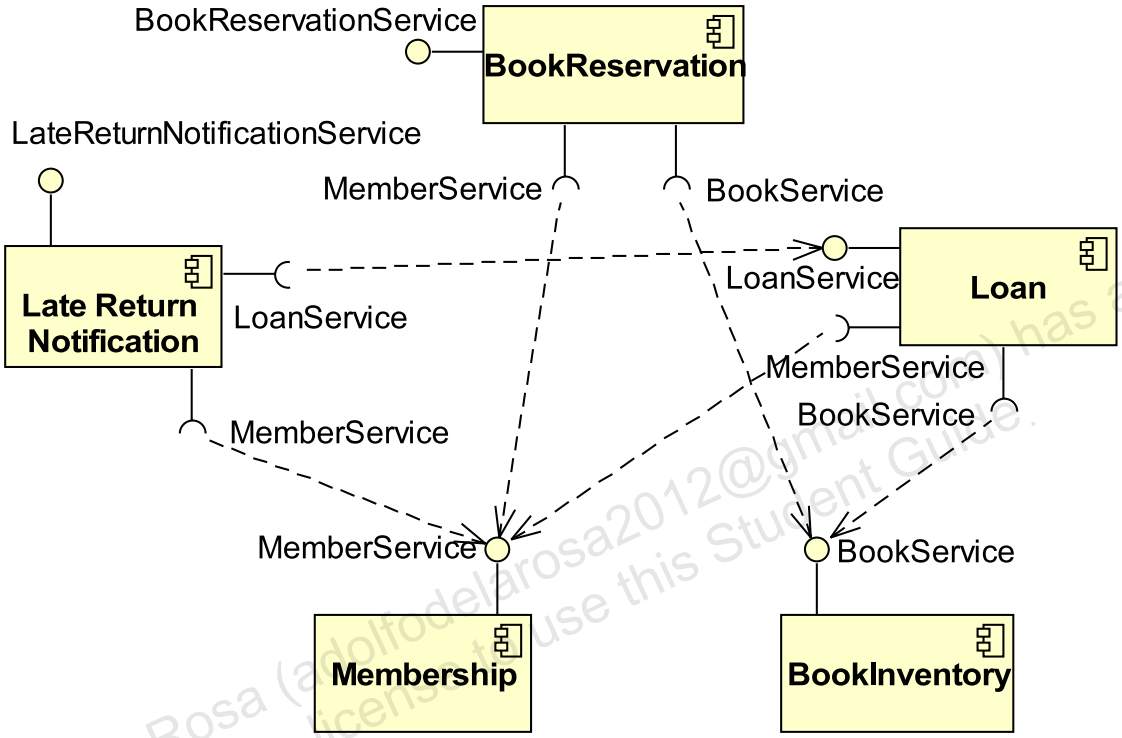

Figure 13-23 Example of Components with Required and Provided Interfaces and Dependency Arrows

Unauthorized reproduction or distribution prohibited. Copyright© 2020, Oracle and/or its affiliates.

Adolfo De Rosa (adolfodelarosa2012@gmail.com) has a non-transferable license to use this Student Guide.

Copyright 2010 Sun Microsystems, Inc. All Rights Reserved. Sun Learning Services, Revision E

Summary

In this module, you were introduced to the process of refining the Domain model. Here are a few important concepts:

- During the Design workflow, you must refine the Domain model to reflect the implementation paradigm.
- This module described how to refine the following Domain model features: attributes, relationships, methods, constructors, and method behavior (by using annotations).
- The classes should be reviewed to ensure that they maintain high cohesion and low coupling.
- Classes can grouped into cohesive components with well-defined interfaces.

Unauthorized reproduction or distribution prohibited. Copyright© 2020, Oracle and/or its affiliates.

Adolfo De+la+Rosa (adolfodelarosa2012@gmail.com) has a non-transferable license to use this Student Guide.

Copyright 2010 Sun Microsystems, Inc. All Rights Reserved. Sun Learning Services, Revision E

Unauthorized reproduction or distribution prohibited. Copyright© 2020, Oracle and/or its affiliates.

Adolfo De+la+Rosa (adolfodelarosa2012@gmail.com) has a non-transferable license to use this Student Guide.