OOD Case Study: Model–View–Controller

15

The objective of this chapter is to apply the Model–View–Controller architectural pattern to the case study using object-oriented design techniques.

15.1 OOD: Preconditions

The following should be true prior to starting this chapter.

- You understand Model–View–Controller as a software architecture that separates the user interface (view) from the application data (model). This separation is achieved by putting the domain logic in the controller and enforcing constraints on how these three components communicate with each other.
- You understand the six characteristics of a good software design: simplicity, coupling, cohesion, information hiding, performance, and security. You have evaluated a software design using these criteria.
- You understand the notion of abstraction as a design process. Abstraction is the ability to generalize a concept by removing details that are not needed to properly convey the design perspective being emphasized.
- You understand five program design criteria: separation of concerns, design for reuse, design only what is needed, performance, and security. You have evaluated program code using these criteria.

15.2 OOD: Transition to MVC

The case study software design discussed in Chap. 12 exhibits low (or weak) cohesion, which characterizes a poor design. This chapter will implement Model–View–Controller to improve the cohesion of the ABA while maintaining the good design

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_15

characteristics of simple, low coupling, logarithmic time performance, and doing input/output data validation.

One way to think about the implementation of Model–View–Controller is from a requirements perspective. We identify the component or components that need to be involved in order to satisfy each requirement. The following requirements have been implemented in the address book application.

1. Allow for entry and (nonpersistent) storage of people's names.
2. Store for each person a single phone number and a single email address.
3. Use a simple text-based user interface to obtain the contact data.
4. Ensure that each name contains only upper case letters, lower case letters, and spaces. At least one letter must be entered for a name. There is no maximum limit on the size of the name entered.
5. Ensure that each phone number contains only the digits zero through nine. At least one digit must be entered for a phone number. There is no maximum limit on the size of the phone number entered.
6. Prevent a duplicate name from being stored in the address book.
7. Do not retrieve any information from the address book.

Implementing requirement 1 involves both the view (*allow for entry*) and model (*allow for nonpersistent storage*) components. Requirement 2 will be part of the model component while requirement 3 is associated with the view component. Requirements 4 and 5 describe the types of validation needed for the name and phone number values. Since this is a standalone application, i.e., it is not distributed across computing devices; this validation belongs to the controller component. (It is important to emphasize that any application whose components are distributed across computing devices should have validation requirements implemented in multiple components. While this duplicates much of the validation logic, it will greatly improve the defensive capabilities of the application in resisting malicious attacks via input channels.) Requirement 6 is another validation requirement, but this one can only be implemented in the model component. This is because the model is the only component that knows the type of data structure being used to non-persistently store the data. Finally, requirement 7 exists to simplify the design of the address book application by eliminating the need for a user interface to include user requests to display address book data. This last requirement affects the design of all three components by reducing the need for logic to display address book entries to the user.[1]

Given the above allocation of requirements to components, requirement 1 has been divided into two sub-requirements. Table 15.1 summarizes the allocation of requirements to components. The remaining sections in this chapter will refer to requirements 1a, 1b, and 2 through 6.

[1] While address book data is displayed once the user exits the application, this is done to verify the address book contained correct information based on the order in which data was entered by the user. As explained in prior chapters, this display of address book data is test code used to verify the correctness of the logic implementing requirements 1 through 6.

Table 15.1 Map requirements to MVC

Requirement	Model	View	Controller
1a. Allow for entry of a person's name		X	
1b. Allow for (nonpersistent) storage of people's names	X		
2. Store for each person a single phone number and a single email address	X		
3. Use a simple text-based user interface to obtain a name, phone number, and email address for each contact		X	
4. Ensure that each name contains only upper case letters, lower case letters, and spaces. At least one letter must be entered for a name. There is no maximum limit on the size of the name entered			X
5. Ensure that each phone number contains only the digits zero through nine. At least one digit must be entered for a phone number. There is no maximum limit on the size of the phone number entered			X
6. Prevent a duplicate name from being stored in the address book	X		

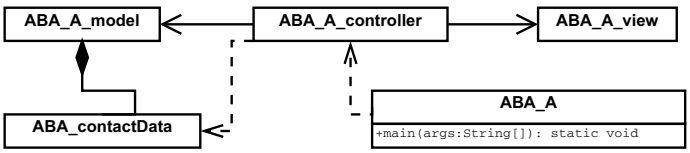

Fig. 15.1 Package diagram for MVC design ABA Version A

15.3 OOD: ABA MVC Design Version A

The Version A design of the first MVC implementation is described using design models and code snippets. The design is explained using a series of questions typically raised when first learning MVC.

15.3.1 OOD: In What Order Do MVC Objects Get Created?

The Version A design for the ABA using Model–View–Controller has five classes, as shown in the package diagram in Fig. 15.1. The model component consists of two classes: ABA_A_model and ABA_contactData. The view component has a single class named ABA_A_view. The controller component consists of the remaining two classes: ABA_A and ABA_A_controller.

The dependency relationship between ABA_A and ABA_A_controller exists because the main method creates an ABA_A_controller object and then uses this object to call the go() method. This is shown in Listing 15.1.

Listing 15.1 ABA MVC Design version A—main method

```
public class ABA_A {
    public static void main(String[] args)
    {
        ABA_A_controller aba = new ABA_A_controller();
        aba.go();
    }
}
```

The package diagram in Fig. 15.1 shows an association relationship between the ABA_A_controller class and two other classes: ABA_A_model and ABA_A_view. The default constructor in the ABA_A_controller class creates objects for these two classes, as shown in Listing 15.2.

Listing 15.2 ABA MVC Design version A—controller default constructor

```
public class ABA_A_controller
{
    private ABA_A_model model;
    private ABA_A_view view;

    public ABA_A_controller()
    {
        model = new  ABA_A_model();
        view = new ABA_A_view();
    }
```

The code in Listings 15.1 and 15.2 illustrates an important difference between dependency and association relationship types. An association relationship identifies a strong relationship between two classes. In the Version A design, the ABA_A_controller class has an instance variable for the two classes it is associated with (ABA_A_model and ABA_A_view). After the controller constructor has executed, any code in this class can communicate directly with the model or view objects. In contrast, a dependency relationship is a weaker relationship between two classes. In this design, the main method has a local variable that references the ABA_A_controller object. This object reference may only be used by code in the main method.

Another way to describe the difference between dependency and association relationship types is the impact a change would have on the related class. For the dependency relationship between ABA_A and ABA_A_controller, there are only two changes to the ABA_A_controller class that would force a change to the ABA_A code. These two changes are (1) the ABA_A_controller class name is changed or (2) the public go() method is changed by either changing its name or adding formal parameters. For the association relationships, a change to the ABA_A_model or ABA_A_view class name, or a change to any public method in these two classes, would result in making changes to the controller code.

The package diagram in Fig. 15.1 shows a composition relationship between the ABA_A_model and ABA_contactData classes. The default constructor method in

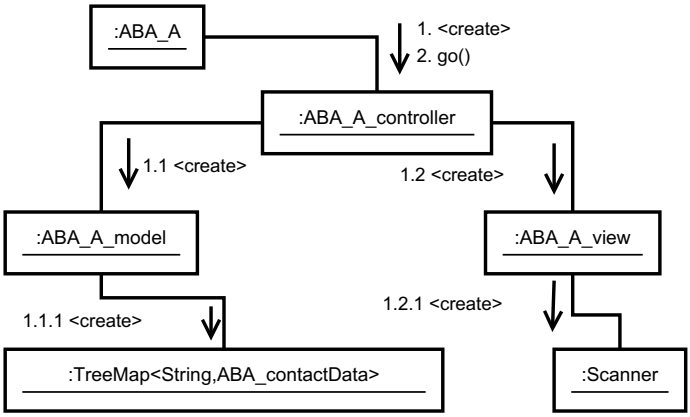

Fig. 15.2 Communication diagram—MVC design Version A—Start up

the ABA_A_model class creates a TreeMap data structure to store ABA_contactData objects. This is shown in Listing 15.3.

Listing 15.3 ABA MVC Design version A—model Startup Code

```
public class ABA_A_model {
    private TreeMap<String,ABA_contactData> book;
    private NavigableSet<String> keySet;
    private Iterator<String> iterKeySet;

    public ABA_A_model()
    {
        book = new TreeMap<String,ABA_contactData>();
        keySet = null;
    }
```

A composition relationship type identifies a whole-part relationship between a class that contains many instances of another class. In the Version A design, the ABA_A_model class has a book instance variable of type TreeMap representing a container. The book container is capable of storing many ABA_contactData objects.

The unified modeling language (UML) has two types of whole-part relationships: aggregation and composition. The notation for aggregation is a diamond symbol with no fill color. As shown in the package diagram in Fig. 15.1, composition uses a diamond symbol filled with black. This Version A design of the ABA uses composition because the ABA_contactData objects must be contained in the book data structure in order for these objects to exist. When the book container is destroyed, all of the ABA_contactData objects will also be destroyed.

The communication diagram in Fig. 15.2 shows the sequence of steps and the interaction of the MVC components in starting the ABA. The steps shown in this diagram are summarized below to reinforce the interactions between the MVC objects as the ABA Version A solution starts to execute.

Step 1: The main method creates an ABA_A_controller object.
Step 1.1: The ABA_A_controller constructor method creates an ABA_A_model object.
Step 1.1.1: The ABA_A_model constructor method creates a TreeMap data structure for nonpersistent storage of contact data, then initializes the keySet instance variable to null.
Step 1.2: The ABA_A_controller constructor method creates an ABA_A_view object.
Step 1.2.1: The ABA_A_view constructor method creates a Scanner object used to obtain user data.
Step 2: The main method calls the go method using the controller object.

The code for the ABA_A_view constructor method is shown in Listing 15.4.

Listing 15.4 ABA MVC Design version A—view Startup Code

```
public class ABA_A_view {
    private ABA_A_controller controller;
    private Scanner console;

    public ABA_A_view(ABA_A_controller controller)
    {
        this.controller = controller;
        console = new Scanner(System.in);
    }
```

15.3.2 OOD: How Do MVC Objects Communicate with Each Other?

The simple, and obvious, answer is by using object references to call public methods. More specifically, the ABA_A_controller class has two instance variables named model and view, as shown in Listing 15.2. These two instance variables contain object references to the ABA_A_model and ABA_A_view objects, respectively. Thus, the controller uses the model variable to call public methods defined in the ABA_A_model class and the view variable to call public methods defined in the ABA_A_view class.

The communication diagram in Fig. 15.3 shows the sequence of steps and the interaction of the MVC components as the user is creating contact information. The steps shown in this diagram are summarized below to reinforce the interactions between the MVC objects as the ABA Version A solution obtains contact information from the user.

Step 2: The main method calls the go method using the controller object.
Step 2.1: The go method obtains a valid contact name. This involves calling the private getValidName method, which will call view.getName() as many times as necessary to ensure a valid contact name is entered. As shown in Listing 15.5,

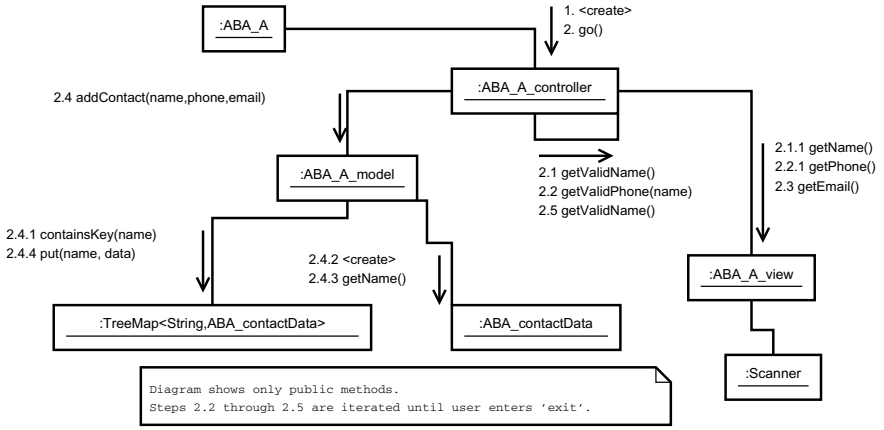

Fig. 15.3 Communication diagram—MVC design Version A

the special value "exit" is a valid contact name which allows the user to indicate they are done entering contact information. Steps 2.2 through 2.5 are performed as long as "exit" is not entered.

Step 2.2: The go method obtains a valid phone number. This involves calling the private getValidPhone method, which will call view.getPhone() as many times as necessary to ensure a valid phone number is entered.

Step 2.3: The go method obtains an email address. This involves calling view.getEmail() only once since there is no requirement to validate email addresses.[2]

Step 2.4: The go method tells the model component to store the contact information. This involves calling the model.addContact method to store the contact data in the book data structure.

Step 2.4.1: The addContact method first determines whether the contact name already exists in the book. When the contact name already exists in the data structure, the remaining 2.4.x steps are *not* performed.

Step 2.4.2: An ABA_contactData object is created.

Step 2.4.3: The contact name is retrieved from the ABA_contactData object.

Step 2.4.4: The name and ABA_contactData object are added to the book container.

[2]We'll discuss security more deeply in Chaps. 24 through 26. At this point in the book, our ABA case study is purposely small and simple so that we can focus on software design decisions and how they impact characteristics of a good software design. In general, software developers should ensure all data provided by an external entity is validated. Specific to the ABA, any serious contact management software should ensure that email addresses are valid and do not contain data that may lead to malicious behavior.

Step 2.5: The go method obtains a valid contact name. This involves calling view.getName() as many times as necessary to ensure a valid contact name is entered.

The code for the ABA_A_controller go method is shown in Listing 15.5.

Listing 15.5 ABA MVC Design version A—go() method

```
public void go() {
    String name, phone, email;
    name = getValidName();

    while (! name.equals("exit"))
    {
        phone = getValidPhone(name);
        email = view.getEmail(name);
        model.addContact(name, phone, email);
        name = getValidName();
    }
    displayBook();
}
```

The communication diagram in Fig. 15.4 shows what happens after the user enters "exit" to end the inputting of contact data. Specifically, the contents of the book are displayed. The steps shown in this diagram are summarized below to reinforce the interactions between the MVC objects as the ABA Version A solution displays contact data to the user.

Step 2.6: The go method calls the controller's displayBook method.
Steps 2.6.1 and 2.6.2: The displayBook method displays two messages by calling the view.displayMsg() method twice.

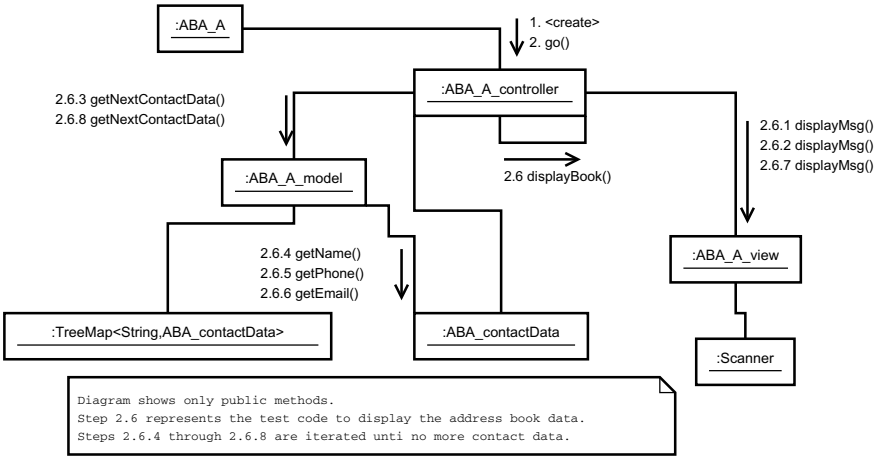

Fig. 15.4 Communication diagram—MVC design Version A—display book

Step 2.6.3: The displayBook method obtains the first contact data. This involves calling the model.getNextContactData method.

Step 2.6.4 through 2.6.6: The displayBook method obtains the contact's name, phone, and email. This involves calling the getName, getPhone, and getEmail methods in the ABA_contactData class.

Step 2.6.7: The displayBook method displays the contact data by calling the view.displayMsg method.

Step 2.6.8: The displayBook method obtains the next contact data. This involves calling the model.getNextContactData method.

15.3.3 OOD: ABA MVC Version A Software Design

The first ABA design using Model–View–Controller has five classes, as shown in Fig. 15.5. This diagram shows a class that contains the static main method, a class for each of the three MVC components, and a class (ABA_contactData) that contains

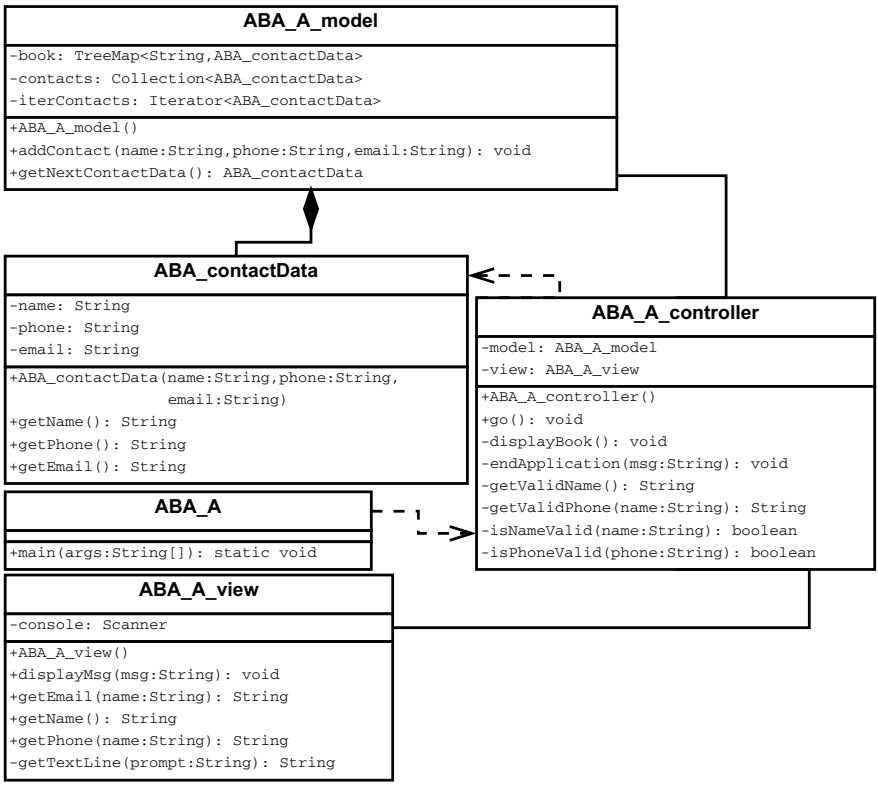

Fig. 15.5 Class diagram—MVC design Version A

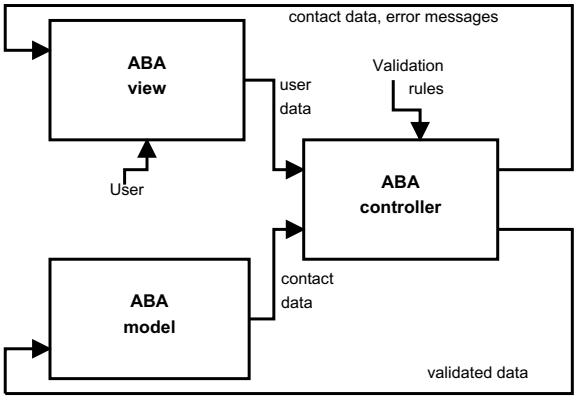

Fig. 15.6 IDEF0 function model—MVC design Version A

contact data for a person. While the controller class has a relationship with the model and view classes, the view and model classes have no relationship with each other. As described above, the model component consists of two classes (ABA_A_model and ABA_contactData), the view component is a single class (ABA_A_view), and the controller is two classes (ABA_A and ABA_A_controller).

The IDEF0 function model shown in Fig. 15.6 describes the primary flows of data between the three ABA components. It also shows the User interacting only with the ABA view component while the ABA controller component performs validation of the data.

The way the user interacts with the ABA Version A design is shown in Fig. 15.7. The user first must enter a valid name, or "exit" to display the address book and end the ABA. Once a valid name is entered, a valid phone number must be entered. Once a valid phone number is entered, any value is accepted for an email address. Only when the name is *not* already in the address book is this contact data added to the address book.

15.3.4 OOD: Evaluate ABA MVC Version A Software Design

The object-oriented design just described will now be evaluated using the six design criteria introduced in Chap. 11.

15.3.4.1 Simplicity

The controller component has two classes. The ABA_A class contains the main method that allows the ABA to execute. This results in the ABA_A class being dependent on the ABA_A_controller class, as described in Sect. 15.3.1. The ABA MVC Version A design uses the controller for any communication between the model and view components. This results in the ABA_A_controller class having an association relationship with the ABA_A_model and ABA_A_view classes. Finally,

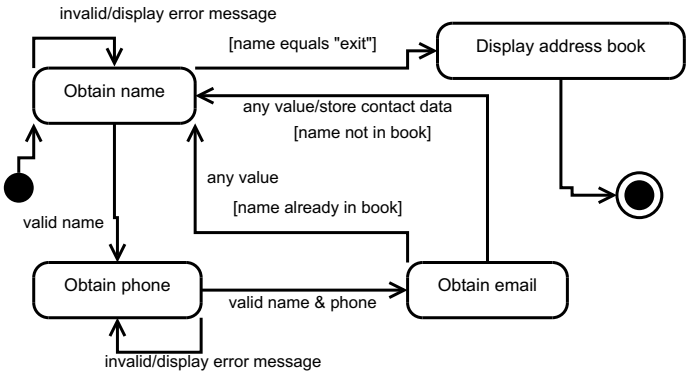

Fig. 15.7 State diagram—MVC design Version A

the controller component displays the contents of the address book via its private displaybook method. This method obtains ABA_contactData objects from the model component, resulting in the controller being dependent on the ABA_contactData class.

The model component has two classes. The ABA_A_model class represents the address book container via its TreeMap data structure. Since the TreeMap contains ABA_contactData objects, and these objects must be in the TreeMap in order to exist, a composition relationship exists between ABA_contactData and ABA_A_model classes.

The view component is a single class used to obtain data from the user and display information to the user.

There are a total of 5 classes, 9 instance variables, and 22 methods in the ABA MVC Version A solution. Of these, 7 instance variables and 20 methods are associated with implementing the seven requirements (1a, 1b, and 2 through 6) shown in Table 15.1. Only 2 instance variables and 2 methods are used as test code to verify the contents of the address book. This seems like a reasonable number of classes, instance variables, and methods for the number of requirements being implemented. The design of ABA_A appears to be simple to understand.

15.3.4.2 Coupling

The Model–View–Controller components described in the ABA MVC Version A design represent three distinct modules. The question is, to what degree does one of these modules rely on the other modules? We can address this question by considering four perspectives on how coupling between two modules can be expressed.

1. Look at the methods within a module that are called by other modules. Does the number of method calls seem excessive?

We can identify the public methods called by other components/modules by looking at the communication diagrams in Figs. 15.2, 15.3, and 15.4, and the class diagram in Fig. 15.5.

Controller public methods: The two public methods in the ABA_A_controller class are called by the main method. The other two components/modules do not call any controller public methods.

Model public methods: Two of the three public methods in the ABA_A_model class are called by the controller component/module to store contact data. The other public method in the ABA_A_model class and three of the four public methods in the ABA_A_contactData class are called by the controller component/module as part of the test code to verify the contents of the address book. The fourth public method in the ABA_A_contactData class is called by the ABA_A_model class to construct a contactData object. None of the model methods are called by the view component/module.

View public methods: All of the public methods are called by the controller. None of these methods are called by the model component.

Looking at coupling from the perspective of method calls shows no coupling between the model and view. The coupling between the controller and model is unidirectional; the controller calls methods within the model to either store or retrieve address book data. The coupling between the controller and view is also unidirectional, the controller calls methods within the view to obtain data from the user. The methods defined with the private access modifier can only be called from code within the same class. This ensures that no (method call) coupling exists between these private methods and another component.

2. Look at the data that is passed via each method call. Does the amount of data being passed between components via method calls seem too excessive or too complex? Looking at the class diagram in Fig. 15.5.

Controller public methods: The two methods in the ABA_A_controller class have no parameters. Since the main method is required to have a String[] argument, it is not considered relevant in terms of assessing coupling based on parameter passing.

Model public methods: Two of these methods have parameters. The addContact method has three parameters to pass the name, phone number, and email address for the address book entry to be stored. The constructor method ABA_contactData is passed the three data values to create an object that stores all of this data.

View public methods: Three of these methods have a parameter. The displayMsg function has a parameter to pass the message to be displayed to the user. The getEmail and getPhone methods have a name parameter so the view can ask for data for a specific contact person.

A parameter passing perspective shows very little coupling between the controller and model and between the controller and view. However, there is one method call from the controller to the view that does increase the coupling between these two components. The controller passes a specific error message when it calls the displayMsg method, which results in this message being displayed to the user. This error message is part of the user interface but is known by the controller component, increasing the coupling between these two components.

3. Look at the data that is returned via each method call. Does the amount of data being returned to another component seem too excessive or too complex?
   Looking at the class diagram in Fig. 15.5.

   Controller public methods: None of these methods return a data value.
   Model public methods: Four of these methods return a data value. The getName, getPhone, and getEmail methods return data values representing contact data while the getNextContactData method returns an ABA_contactData object.
   View public methods: Three of these methods return a String data value. The getEmail, getName, and getPhone methods return the email address, name, or phone number, respectively, entered by the user.

   A method return perspective shows no coupling between the model and view modules, coupling between the controller and model is unidirectional (model methods return data values to the controller but not vice versa), and coupling between the controller and view is also unidirectional (view methods return data values to the controller).
4. Look at the use of global data structures. Does one module define a global data structure that is referenced by another module?
   As seen in the class diagram in Fig. 15.5, all of the instance variables specified in each class are defined using the private access modifier (i.e., the dash in front of each attribute name indicates private). This ensures these data structures are accessible only by the methods specified within the class. The ABA_contactData class is constructed and stored within the model and given to the controller as the return value when the getNextContactData method is called. This class/object is used to give all of the contact data to the controller by encapsulating all of this data into a single class.

Given the above descriptions on coupling between the model, view, and controller components, the amount of coupling seems minimal and thus represents a good design.

15.3.4.3 Cohesion

The MVC components described in the design of ABA MVC Version A represent three distinct modules. The question is, to what degree does each of these modules contain methods that are strongly-related to each other and are single-minded in their purpose? We'll look at the purpose of each method and instance variable within the same module, and whether (or how much) these are related to each other.

Controller methods: With two exceptions, the controller methods either provide the processing flow of the application or implement data validation requirements. This matches the purpose of the controller component. The exception is the displayBook method, which is part of the test code to verify the contents of the address book.

The application processing flow implemented in the controller determines when the user has requested the application end by checking the name value entered to see if it matches "exit". This behavior requires the controller know how the user requests the application to end. This knowledge should reside in the view. One method within the controller is part of the test code that displays the contents of the address book. This method acts as an intermediary between the view—responsible for displaying address book data—and the model—which is responsible for storing this data.

The validation logic implemented in the controller includes the getValidName and getValidPhone methods passing a validation error message to the view for display to the user. This makes these two methods less cohesive with the rest of the controller since the controller is not responsible for communication with the user. The values passed to the validation methods and the method return values are all cohesive with the purpose of validating contact data.

Controller instance variables: The two private attributes allow the controller to call public methods in one of the other components.

Model methods: All of the model methods are designed to either store or retrieve address book data. This matches the purpose of the model component. The data used by these methods, whether this data comes from parameters or instance variables, all have to do with contact data. Likewise, the return values from these methods also have to do with contact data.

Model instance variables: The six private attributes in the model are all related to address book data. In particular, the *contacts: Collection* and *iterContacts: Iterator* instance variables are used by the test code to retrieve the address book data one entry at a time. Besides the constructor method, all the methods in the ABA_contactData class are *getter* methods. That is, once an ABA_contactData is constructed, the only thing the object allows is retrieval of its data.

View methods: All of the view methods are used to display information to and obtain data from the user. This matches the purpose of the view component. The data used by these methods, whether this data comes from parameters or instance variables, all have to do with the user interface. Likewise, the return values from these methods also have to do with data from the user.

View instance variables: The private attribute in the view allows this component to obtain data from the user.

The model component is highly cohesive, whereas the controller and view modules are less cohesive. As mentioned above, the controller validation logic gives the view an error message to be displayed, making the controller less cohesive. For the view, one could argue that the single class in the view is doing two distinctly different things—it is obtaining data from the user and it is displaying data to the user. Since the ABA is using a text-based user interface, the mechanisms used to obtain data (i.e., Scanner object) are different from the mechanisms to display data (i.e., System.out).

15.3.4.4 Information Hiding

Does each module in the MVC design Version A hide its implementation details from the other components? By implementation details, we mean the algorithms and data structures that are being used by the module.

Controller: The controller contains methods that either provide the processing flow of the application or implement the data validation requirements. None of the other components know how the controller performs validation.

Model: The model stores the address book data and determines if a name is already found in the address book. These data structures and algorithms are not known by the other modules.

The model includes a class—ABA_contactData—that is used to store one instance of contact data. Instances of this class are used by the model component and also passed to the controller component. Besides the constructor method, all the methods in the ABA_contactData class are *getter* methods. Since the MVC Version A implementation includes test code to display the contents of the address book, this sharing of contact data between model and controller is necessary since the controller must give the view the actual data to be displayed to the user.

View: The view displays information to and obtains data from the user. The way in which the view accomplishes this is not known by the other modules.

One other perspective on information hiding is to look to see how the interface to each component is defined. In this design, the list of public methods found in each class represents the interface to the component. The list of public methods is fairly typical given the list of requirements that we are addressing in our design. More about interfaces is discussed below.

15.3.4.5 Performance

The performance of the MVC Version A design is similar to the performance of the ABA solutions described in Chap. 6. Specifically, the use of a Java TreeMap gives us a worst case of $O(log_2 n)$ performance.

15.3.4.6 Security

The security of the MVC Version A design is similar to the security of the ABA solutions described in Chap. 9. Specifically, the design includes data input validation, data output validation, exception handling, and fail-safe defaults. In addition, Java is a type-safe language.

15.4 OOD: ABA MVC Design Version B

The design of MVC Version A of the ABA is okay, but can be improved. First, the coupling between the controller and view is too high. This is due to the (relatively) large number of view methods called by the controller, which results from the view methods being too specific regarding the input of user data. Fortunately, we already have the ABA_contactData class that contains all of the data for a single contact person. Why not have the view construct one of these objects each time it gets data from the user? The view can then give this object to the controller, which can then process this contact data.

Second, the cohesion of the controller can be improved. Specifically, the logic that compares the contact name to "exit" should really be in the user interface (view) component. To make the controller more cohesive, any user interface logic should be removed; the controller should only be responsible for the execution flow of the application and data validation.

The Version B design for the ABA using Model–View–Controller has seven classes, as shown in the package diagram in Fig. 15.8. The model and controller components have the same classes as the Version A design. The view component now has three classes: ABA_B_view, ABA_B_viewDisplayData, and ABA_B_viewObtainData.

The changes to the Version B package diagram, when compared to Version A, include the following: an association between the ABA_B_view and the two

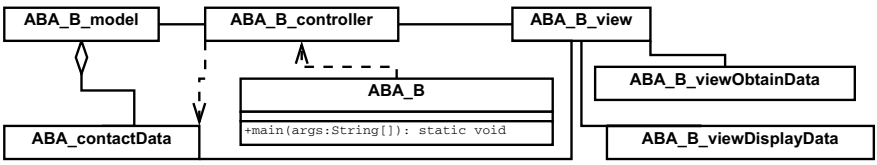

Fig. 15.8 Package diagram for MVC design ABA Version B

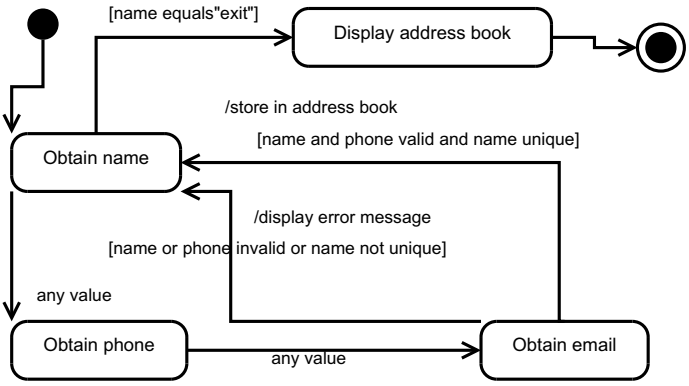

Fig. 15.9 State diagram—MVC design Version B

new classes—ABA_B_viewDisplayData and ABA_B_viewObtainData classes, an association between the ABA_B_view and ABA_contactData classes, and the composition relationship between ABA_B_model and ABA_B_contactData is now an aggregation relationship.

The association between the ABA_B_view and ABA_contactData classes is based on the view component creating an ABA_B_contactData object after obtaining the name, phone number, and email address from the user. The view returns this object to the controller, which will then validate the contact data. The fact that the view component creates an ABA_B_contactData object, which if valid, will be stored by the model in the TreeMap results in the relationship between the ABA_B_model and ABA_B_contactData being changed to an aggregation since the objects exist outside of the data structure.

To address the design flaws described above in Version A, the way the user interacts with the ABA has been changed in Version B. Figure 15.6 shows the IDEF0 function model for Version A. Since this is a high-level description of the Version A design, this function model also applies to Version B. However, the statechart in Fig. 15.9 illustrates significant changes to how the user interacts with the Version B ABA. The user enters a value for contact name, phone number, and email address. After entering these three values, a contact name value of "exit" causes the address book data to be displayed and the ABA ends. Otherwise, the contact name and phone number are validated. When these two values are valid, the contact data is added to the address book when the contact name is not already in the book. When either or both values are invalid, appropriate error messages are displayed. In either case, the user is prompted to enter the next contact person's name.

The second ABA design using Model–View–Controller has seven classes, as shown in Fig. 15.10. As described above for the Version B package diagram, the view component has two additional classes (ABA_B_viewDisplayData and ABA_B_viewObtainData) and creates ABA_contactData objects that are then stored in a TreeMap by the model component.

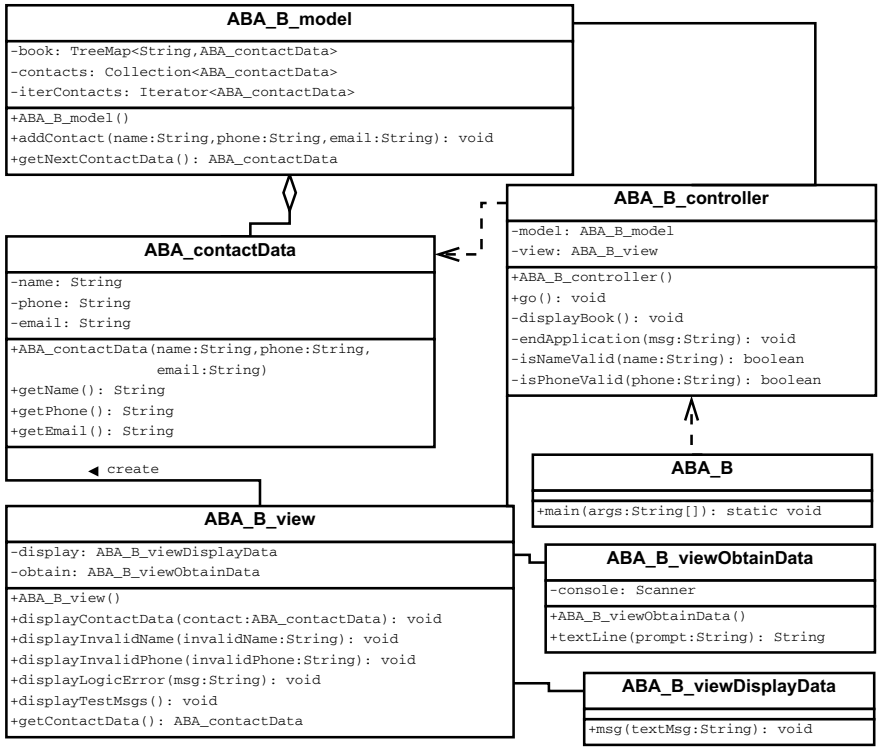

Fig. 15.10 Class diagram—MVC design Version B

15.4.1 OOD: In What Order Do MVC Objects Get Created?

The communication diagram in Fig. 15.11 shows the sequence of steps and the interaction of the MVC components in starting the ABA. The steps shown in this diagram are summarized below to reinforce the interactions between the MVC objects as the ABA Version B solution starts to execute.

Step 1: Same as MVC Version A design.
Step 1.1: Same as MVC Version A design.
Step 1.1.1: Same as MVC Version A design.
Step 1.2: Same as MVC Version A design.
Step 1.2.1: The ABA_A_view constructor method creates an ABA_A_viewDisplayData object used to display data to the user.

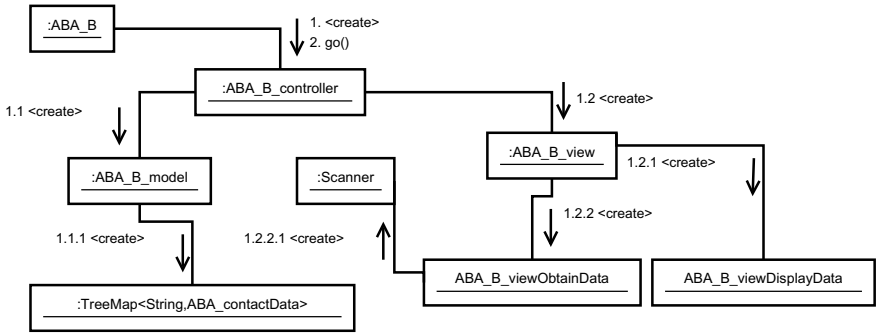

Fig. 15.11 Communication diagram—MVC design Version B—Start up

Step 1.2.2: The ABA_A_view constructor method creates an ABA_A_viewObtain Data object used to obtain user data.
Step 1.2.2.1: The ABA_A_viewObtainData constructor method creates a Scanner object used to obtain user data.
Step 2: Same as MVC Version A design.

The code for the ABA_A_view constructor method is shown in Listing 15.6.

Listing 15.6 ABA MVC Design version B—view Startup Code

```
public class ABA_B_view {
    private ABA_B_viewDisplayData display;
    private ABA_B_viewObtainData obtain;
    private final String EXIT = "exit";

    public ABA_B_view()
    {
        this.display = new ABA_B_viewDisplayData();
        this.obtain = new ABA_B_viewObtainData();
    }
```

15.4.2 OOD: How Do MVC Objects Communicate with Each Other?

The communication diagram in Fig. 15.12 shows the sequence of steps and the interaction of the MVC components as the user is creating contact information. The steps shown in this diagram are summarized below to reinforce the interactions between the MVC objects as the ABA Version B solution obtains contact information from the user.

Step 2: Same as MVC Version A design.
Step 2.1: The go method obtains contact data. This involves calling the public view.getContactData method, which will call the textLine method three times to

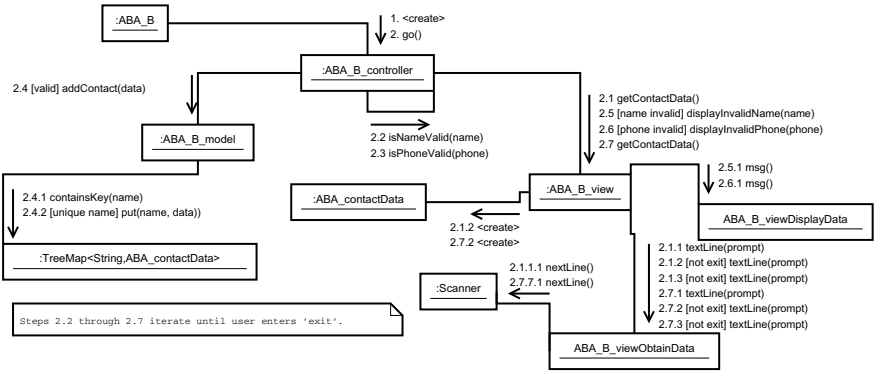

Fig. 15.12 Communication diagram—MVC design Version B

obtain a name, phone, and email address. As shown in Listing 15.7, the go method now iterates until the view component returns null instead of an ABA_contactData object. Listing 15.8 shows how the view component uses the special value "exit" to determine if the user is done entering contact data. Steps 2.2 through 2.7 are performed as long as the view does not return a null to the go method.

Step 2.2: The go method validates the contact name. This involves calling the private isNameValid method.

Step 2.3: The go method validates the phone number. This involves calling the private isPhoneValid method.

Step 2.4: When the contact data is valid, the go method tells the model component to store the contact information. This involves calling the model.addContact method to store the contact data in the book data structure.

Step 2.4.1: The addContact method first determines whether the contact name already exists in the book. When the contact name already exists in the data structure, Step 2.4.2 is *not* performed.

Step 2.4.2: The name and ABA_contactData object are added to the book container.

Step 2.5: When the contact name is invalid, the go method tells the view component to display an appropriate error message.

Step 2.6: When the phone number is invalid, the go method tells the view component to display an appropriate error message.

Step 2.7: The go method obtains contact data, as described by Step 2.1.

The code for the go method found in the ABA_A_controller class is shown in Listing 15.7. Of significance is the iteration to obtain contact data that stops when the view component returns null instead of an ABA_contactData object.

Listing 15.7 ABA MVC Design version B—go() method

```
public void go() {
    ABA_contactData data;
    boolean nameValid, phoneValid;
    data = view.getContactData();

    while (data != null)
    {
        nameValid = isNameValid(data.getName());
        phoneValid = isPhoneValid(data.getPhone());
        if (nameValid && phoneValid)
            model.addContact(data);
        else
        {
            if (! nameValid)
                view.displayInvalidName(data.getName());
            if (! phoneValid)
                view.displayInvalidPhone(data.getPhone());
        }
        data = view.getContactData();
    }
    displayBook();
}
```

The code for the getContactData method found in the ABA_A_view class is shown in Listing 15.8. This illustrates two important design choices: the view component is the only component that knows how the user enters contact data; and this method detects when the user is done entering data, as shown in the selection logic in the first if statement.

Listing 15.8 ABA MVC Design version B—getContactData() method

```
public ABA_contactData getContactData() {
    ABA_contactData userData;
    String name, phone, email;
    name = obtain.textLine("Enter contact name ('exit' to quit): ");
    if (name.equals(EXIT))
        userData = null;
    else
    {
        phone = obtain.textLine("Enter phone number for "+name+": ");
        email = obtain.textLine("Enter email address for "+name+": ");
        if (phone.equals(EXIT) || email.equals(EXIT))
            //One of these values may be EXIT if user enters CTRL-Z.
            userData = null;
        else
            userData = new ABA_contactData(name, phone, email);
    }
    return userData;
}
```

15.4.3 OOD: Evaluate ABA MVC Version B Software Design

The object-oriented design just described will now be evaluated using the six design criteria introduced in Chap. 11.

15.4.3.1 Simplicity

The design changes to Version B results in two new classes to the view component and an association relationship between the ABA_B_view and ABA_contactData classes.

There are a total of 7 classes, 11 instance variables, and 23 methods in the ABA MVC Version B solution. Of these, 7 instance variables and 20 methods are associated with implementing the seven requirements (1a, 1b, and 2 through 6) shown in Table 15.1. Only 2 instance variables and 3 methods are used as test code to verify the contents of the address book.

The changes made to improve the Version B design seem like a reasonable number of classes, instance variables, and methods for the number of requirements being implemented. The design of ABA_B appears to be simple to understand.

15.4.3.2 Coupling

The design changes to Version B adds an association relationship between the ABA_B_view and ABA_contactData classes, which reside in the view and model modules, respectively. We'll address this additional coupling between these two modules using the same four perspectives discussed in Sect. 15.3.4.

1. Look at the methods within a module that are called by other modules. Does the number of method calls seem excessive?
   The ABA_B_view class constructs an ABA_contactData object after obtaining the appropriate contact data from the user. This method call results in a coupling between the view and model components which did not exist in Version A.
2. Look at the data that is passed via each method call. Does the amount of data being passed between components via method calls seem too excessive or too complex?
   The construction of an ABA_contactData object results in the view module passing three data values representing the contact data entered by the user.
   The other significant change in Version B is the elimination of the generic displayMsg(String) public method and the addition of five public methods in the ABA_view class. In Version A, the controller would construct a String object representing the data or message to be displayed to the user. The controller would then call the displayMsg(String) method to display the data/message. This results in the Version A controller having significant knowledge regarding the format and content of what is being displayed to the user. In Version B, the five new public methods in the view component are designed for a single purpose. The displayContactdata(ABA_contactData), displayInvalidNameString, displayInvalid-

Phone(String), and displayLogicError(String) methods will format a message using the data contained in the formal parameter. The displayTestMsg() method has no formal parameters, resulting in this view method being completely responsible for formatting and displaying appropriate test messages. To conclude, the Version B view module now has *display* methods that are designed for specific purposes. They receive data from the controller and decide how to format this data for display to the user. Given this, we can say that the Version B controller has less information about how information is displayed to the user, reducing the coupling between these two modules.

3. Look at the data that is returned via each method call. Does the amount of data being returned to another component seem too excessive or too complex?
   The construction of an ABA_contactData object results in the view module having an object that was constructed by the model component.
4. Look at the use of global data structures. Does one module define a global data structure that is referenced by another module?
   No global data structures were added to the Version B design.

Even with the new association relationship between the view and model components, the amount of coupling in Version B is minimal and thus represents a good design.

15.4.3.3 Cohesion

Each of the MVC components is described below by identifying the changes in Version B and how these affect cohesion of the module.

Controller methods: Two methods in Version A are no longer in the ABA_B_controller class. Eliminating these two methods in Version B has no impact on the cohesiveness of the controller.
The application processing flow implemented in the Version B controller determines when the user has requested the application to end by testing the view.getContactData() method's return value. When this method returns null, the controller will end the application. Knowledge about how the user indicates an end to the ABA is now solely contained in the view component.
The validation logic implemented in the Version B controller now passes an invalid name or invalid phone number to the view module. The view is then responsible for displaying an appropriate error message to indicate the data value is invalid. In Version B, the controller is only responsible for performing validation logic; displaying an appropriate validation error message is now in the view (where it should be!).

Controller instance variables: Same as MVC Version A design.

Model methods: Same as MVC Version A design.

Model instance variables: Same as MVC Version A design.

View methods: The view component now has three classes. The public methods in the ABA_B_view class represent the interface used by the controller. While

the Version A view component had one public method responsible for displaying any data/message to the user, Version B has five public methods used to display data or messages. Each of these methods is responsible for a specific interaction with the user. The public methods in the two new classes are called only by the ABA_B_view class.

View instance variables: The private attributes in the three view classes are directly related to the purpose of this component.

The Version B controller component is more cohesive than Version A, largely due to the changes in how the controller interfaces with the view. The view component, even with its three classes, is more cohesive as it contains all of the knowledge about how the user interface is implemented.

15.4.3.4 Information Hiding

Each of the MVC components is described below by identifying the changes in Version B and how these affect information hiding within each module.

Controller: Same as MVC Version A design.

Model: The model continues to store ABA_contactData objects and determines if a name is already found in the address book. However, the Version B view component creates ABA_contactData objects and gives these to the controller for validation. This design change has a minor impact on information hiding since the view component now understands how to create an ABA_contactData object.

View: Same as MVC Version A design.

15.4.3.5 Performance

The performance of the MVC Version B design is similar to the performance of the ABA solutions described in Chap. 6. Specifically, the use of a Java TreeMap gives us a worst case of $O(log_2 n)$ performance.

15.4.3.6 Security

The security of the MVC Version B design is similar to the security of the ABA solutions described in Chap. 9. Specifically, the design includes data input validation, data output validation, exception handling, and fail-safe defaults. In addition, Java is a type-safe language.

15.5 OOD Top-Down Design Perspective

We'll continue to use the personal finances case study to reinforce Model–View–Controller via a top-down design approach.

15.5.1 OOD Personal Finances: A Second Case Study

The requirements for personal finances, as stated in Chap. 12, are listed below.

- Allow a user to create as many accounts as they would like. Each account represents a single financial asset or liability that is provided as a financial service to an individual. Accounts are typically created to represent checking and savings accounts, credit card accounts, and loans for an automobile, school, or home. Each account has a name and a balance.
- Allow a user to enter as many transactions as they would like in each account. Each transaction represents a credit or debit within the account. Each transaction has a date, description, and amount (which can be a credit or debit). A transaction may optionally have a number (e.g., a check number for a checking account transaction) and zero or more labels.
- Allow a user to create labels, which are used to categorize a transaction. For example, labels may be created to indicate whether a transaction is associated with a charity, groceries, or home improvement.
- Allow a user to create as many reports as they would like. Each report includes one or more accounts, and zero or more labels.
- Allow a user to modify or delete any account, transaction, label, or report.

Below, we will compare our top-down MVC software design with Chap. 12 designs, which focused on the domain of personal finances and ignored implementation details related to user interface and persistent storage.

15.5.2 OOD Personal Finances: Structure

The package diagram in Fig. 15.13 shows three named packages reflecting the use of Model–View–Controller. Comparing this with the package diagram in Fig. 12.8, we see domain classes identified in Chap. 12 in both the model and view packages in Fig. 15.13. Given the responsibility of the model—to store domain data—the model needs to understand the structure and relationships of the domain data. Likewise, the responsibility of the view—to provide interaction with a user—also suggests this package understands the types of data to be obtained from or displayed to a user.

One other item to note is the presence of the controllerValidate class in the controller package. While no requirements have been stated for validation of personal finance data, designers and programmers should validate data that comes from any external source. In the absence of requirements, developers should identify validation requirements based on known threats and common sense. Put another way, if a vulnerability is discovered in the software that is traced back to not doing data validation, is someone going to point to the lack of requirements as the source of this vulnerability? Likely not; the source of the vulnerability will be the developers that did not design, program, and test to ensure validation is done appropriately.

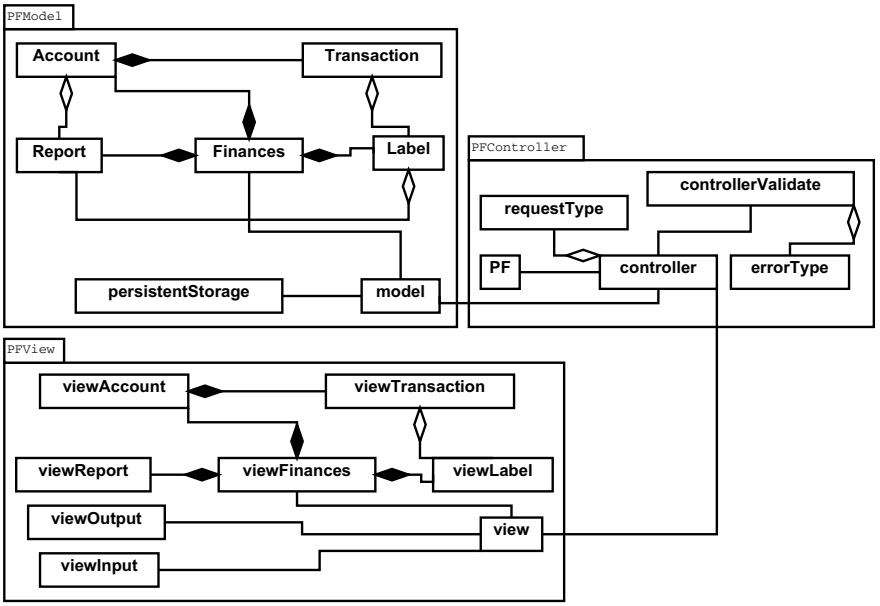

Fig. 15.13 Personal finance package diagram

Three class diagrams are shown below, one for each package. Figures 15.14, 15.15, and 15.16 show the attributes and public methods for each class in the model, view, and controller packages, respectively. These class diagrams are at a high level of abstraction; more work would be needed to translate these conceptual models into a detailed design or into an implementation. Examples of this abstraction include the following.

- Only one class identifies private methods (see the controllerValidate class). A more detailed design would likely show private methods in just about every class. As you know, private methods are used to implement common logic or detailed processing found within a single class.
- There are very few classes that have *getter* and *setter* methods. For example, the *Transaction* class will need to have a method that returns each attribute value (e.g., getDate(), getDescription(), getAmount() and getNumber()), and a method that sets each attribute value (e.g., setDate(date:Date), setDescription(desc:String), setAmount(amount:Currency), and setNumber(number:String)). The getter methods would be needed when generating a report and the setter methods would be needed when a transaction is modified.
- Some model classes use the generic term *container* to represent a data structure that will need to be used to store multiple data values.
- The model class persistentStorage is vague about the type of persistent storage technology used in the application. A generic PersistenceType represents the per-

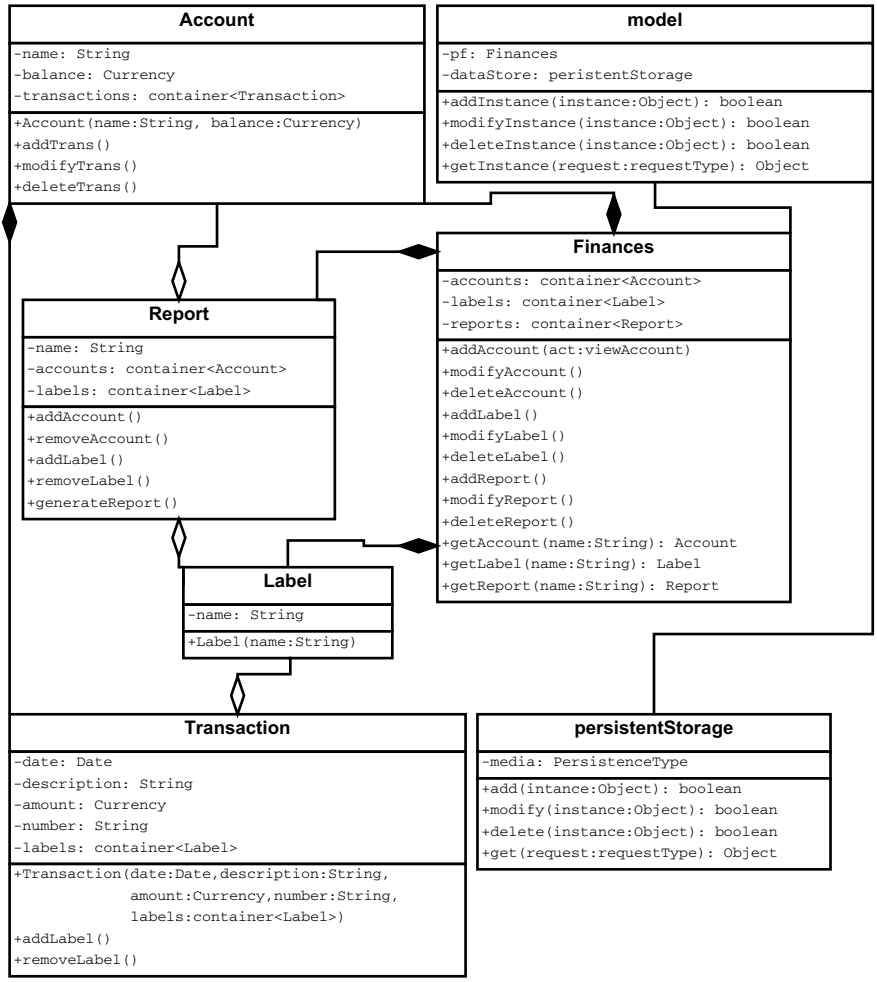

Fig. 15.14 Personal finance class diagram—model component

sistent data store, without having to specify, at this point in the design process, the type of data storage technology to be used. Once the persistent storage technology is selected, additional *helper classes* will likely be used to manage the details associated with the technology.

- The view classes representing the personal finance domain—viewAccount, viewFinances, viewLabel, viewReport, and viewTransaction—may be responsible for the user interface associated with each type of data. If this decision were made, then using these same classes as container objects passed to the controller would be a bad idea—a design should not allow the controller component to directly access user interface elements.

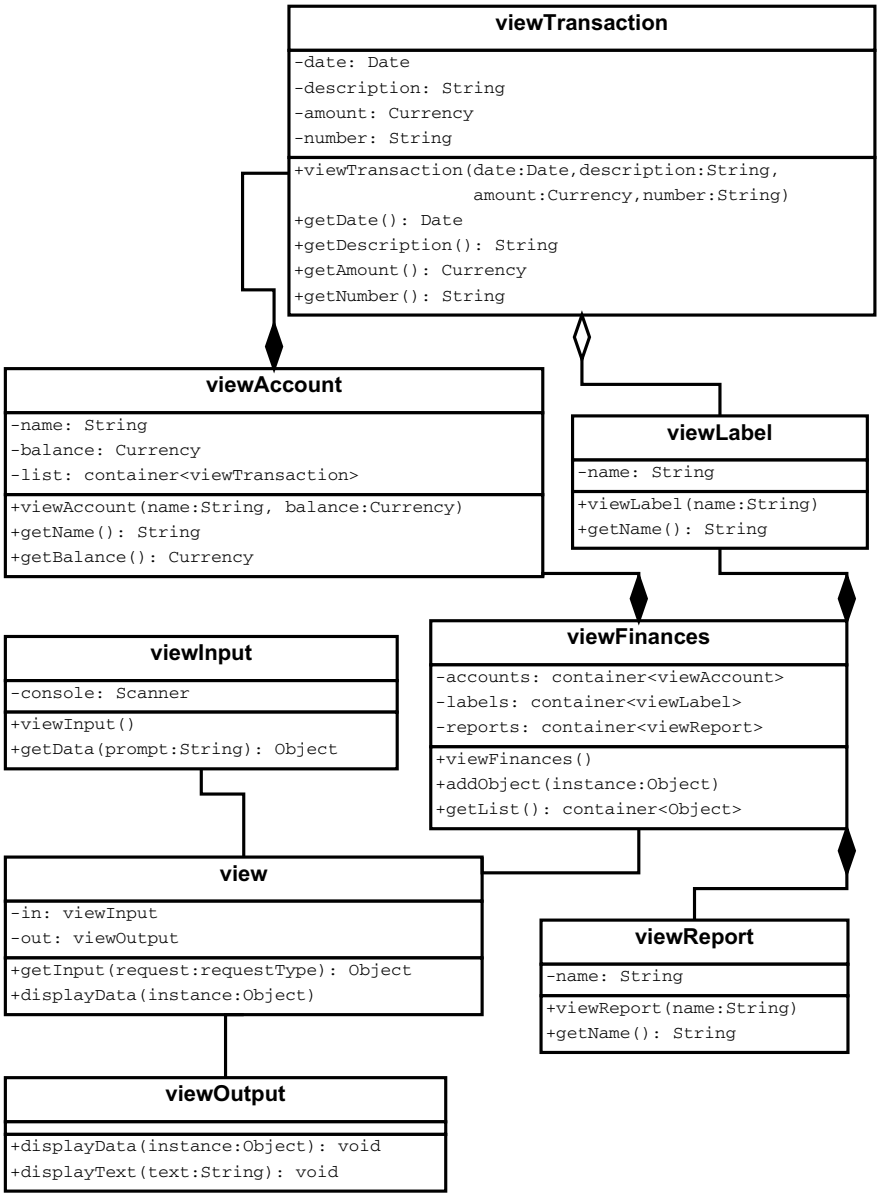

Fig. 15.15 Personal finance class diagram—view component

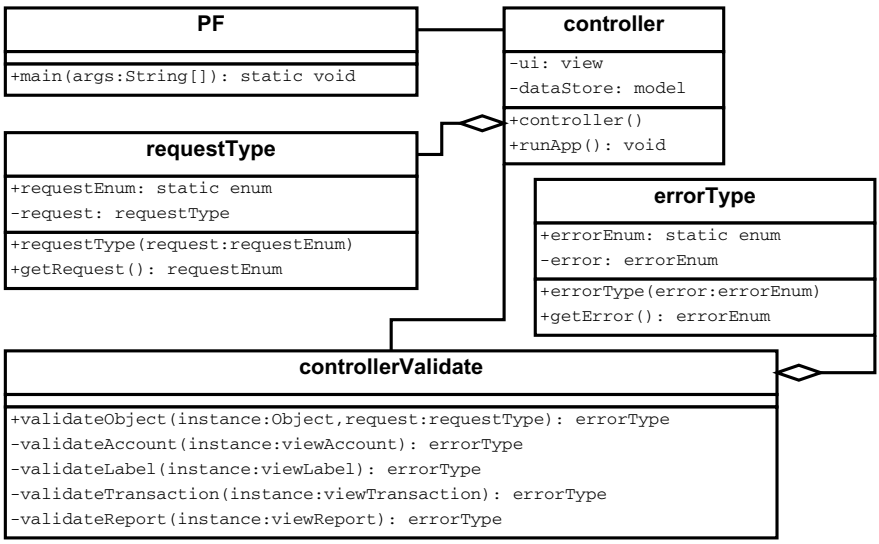

Fig. 15.16 Personal finance class diagram—controller component

- A more detailed design of the controller package would likely include additional *helper classes* to support the overall application flow of the personal finances software.
- The controller class would likely have many private methods that provide the detailed processing in support of the runApp method.

15.5.3 OOD Personal Finances: Structure and Behavior

The Data-Flow Diagram (DFD) used in Chap. 12 is shown in Fig. 15.17. This diagram is already at an appropriate level of abstraction. Modifying this diagram to show Model–View–Controller would not add any information that is not already being expressed in the package and class diagrams.

The UML communication diagram used in Chap. 12 is shown in Fig. 15.18. As you may recall, this diagram shows the basic interactions between the objects based on an understanding of the personal finance domain. In contrast, the UML communication diagram in Fig. 15.19 describes the object interactions associated with using Model–View–Controller. This communication diagram shows the interactions between objects to create a new account.

Figure 15.19 shows two messages labeled *1.1: getData(prompt)* and *1.2: getData(prompt)*. These result in the user entering the account name and balance. This user-supplied data is then used to create a viewAccount object (see Step 1.3), which is then given to the controller. Assuming the viewAccount object contains a valid account name and balance, the controller gives this object to the model via the *3: addInstance(viewAccount)* message. Within the model component, the viewAccount

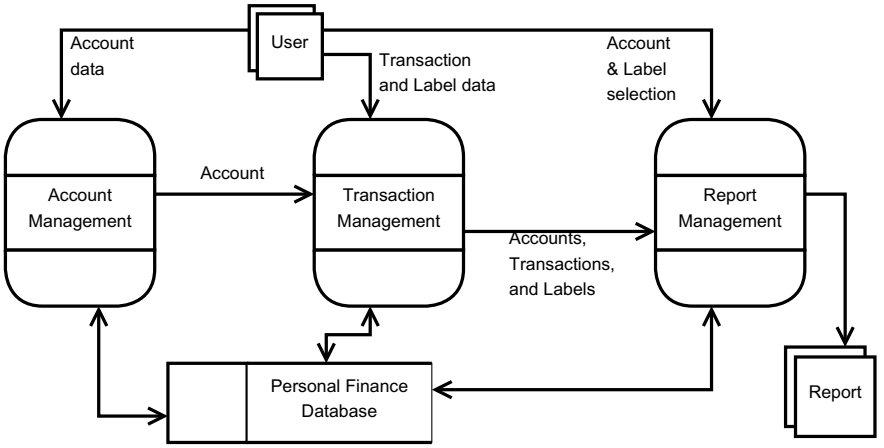

Fig. 15.17 Personal finance data-flow diagram (from Chap. 12)

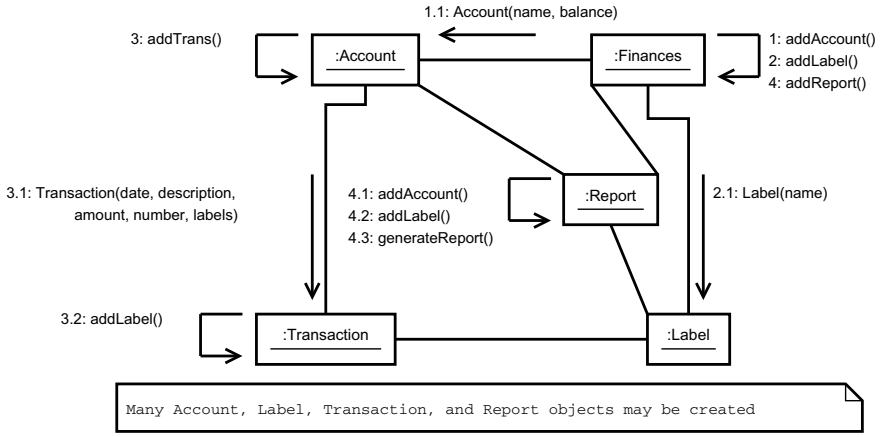

Fig. 15.18 Personal finance communication diagram (from Chap. 12)

object is used to add an account to the Finances container object via the *3.1: addAccount(viewAccount)* and *3.1.1: «create»* messages. After this, the account is added to persistentStorage via the message labeled 3.3.

15.5.4 OOD Personal Finances: Behavior

The UML statechart diagram described in Chap. 12 shows the user interactions with the personal finance application. Since changing the software design to use Model–View–Controller has no impact on the user interface design, there is no need to modify the statechart diagram. That is, the user interface behavior has not been changed even though the internal structure of the application code has gone through a dramatic change!

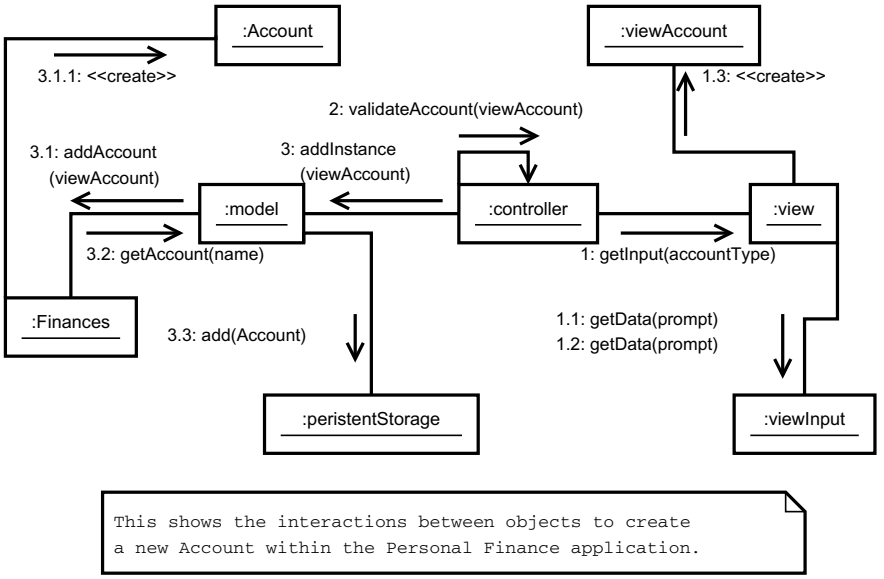

Fig. 15.19 Personal finance communication diagram

15.5.5 OOD Personal Finances: Summary

15.5.5.1 Design Models

The design models shown in Figs. 15.13, 15.14, 15.15, 15.16, and 15.19 show the impact Model–View–Controller has on the number and type of classes needed, and how these classes/objects interact with each other. As discussed in Chap. 12, a software architect may want/need to emphasize only certain aspects of the design. For example, if an understanding of object-orientation is critical to the success of the personal finance software, then the package, class, and communication diagrams are important while the DFD and statechart may be omitted. On the other hand, perhaps showing the interactions of the personal finance system with a user and persistent storage is important. In this case, the DFD, communication diagram, and statechart would be important to develop.

15.5.5.2 Evaluation

An evaluation of the personal finances design, as shown in Figs. 15.13, 15.14, 15.15, 15.16, and 15.19, is briefly described below.

Simplicity: Since we have described a high-level design, or architecture, our efforts should have resulted in models that accurately reflect the needs/requirements while also abstracting away the details implied by the requirements. The package, DFD, and statechart models are simple to read while the class and communication

diagrams provide significantly more details. Having a class diagram for each of the three packages (i.e., for the three MVC components) gives the appearance that this design provides too many details, perhaps making the class diagrams too hard to read and understand for such a high-level design. On the other hand, it is necessary to show the personal finance domain classes in both the view and model to illustrate an understanding of the application domain.

Coupling: The package, class, DFD, and communication models provide information to help us assess the coupling between the design elements. In the case of the package, class, and communication models, we see the interactions between the three MVC components are fairly clean—one class within each component is responsible for interacting with another component. This reduces the coupling between the MVC components. Coupling between classes residing in the same package/component is necessarily going to be a bit higher since these classes are cooperating with each other to satisfy the purpose of the component.

Cohesion: By using the Model–View–Controller architecture, the design models now show those classes responsible for the user interface, data storage, program flow, and domain knowledge. While there is some redundancy in the classes in the view and model to represent the personal finance domain, these classes are now more cohesive since they focus on only one aspect—data storage or user interface. This has clearly been improved when we compare this chapter's design with Chap. 12.

Information hiding: Based on the use of MVC, it is clear that the view has no idea how the model performs its responsibilities and vice versa. Similarly, while the controller has an interface with both the model and view, neither of these components knows how the controller performs its responsibilities (i.e., controls program flow and implements domain-specific processing).

Performance: Without knowing more details about the data structures (i.e., containers) to be used, it is difficult to assess the performance of this high-level design. In addition, the *Personal Finance Database* shown on the DFD may have performance implications that cannot be assessed at this time.

Security: The case study requirements do not state any need for validation of data. However, the controller component clearly shows that it will be doing validation of accounts, labels, reports, and transactions.

15.6 OO Language Features

Most object-oriented programming languages, including Java, allow a developer to create both interfaces and abstract classes. These two concepts are introduced in this chapter and then used in the case studies in future chapters.

Interface: A blueprint used to create a class definition. A Java interface may contain static constants and method signatures. In a class diagram, an interface is realized by another interface or a class.

abstract class: A class that cannot be used to construct an object. An abstract class may contain abstract and concrete methods, class and instance variables, and class and instance methods. In a class diagram, an abstract class is inherited by another class, which may be abstract or concrete.

15.6.1 A Simple Example

This example will use an interface and an abstract class to represent a small part of a software design for an autonomous car.

Listing 15.9 shows an interface that has one static constant and two method signatures.

Listing 15.9 Java Interface

```
public interface demoInterfaceCar {
    public static final double TOO_FAST_KPH = 160.9;

    //Methods in an interface are abstract w/out using the keyword.
    public boolean slowDown(double amount);
    public boolean speedUp(double amount);
}
```

Listing 15.10 shows a class that implements this interface. This implementation must define a concrete method for each of the method signatures found in the interface. As shown in Listing 15.10, a class implementing an interface may have members (i.e., instance variables and methods) not associated with the interface.

Listing 15.10 Use a Java Interface

```
public class demoUseInterfaceCar implements demoInterfaceCar {
    private int numberPassengers;
    private double speed;

    public demoUseInterfaceCar(int numberPassengers, double speed)
    {
        this.numberPassengers = numberPassengers;
        this.speed = speed;
    }

    public int getNumberPassengers()
    {
        return numberPassengers;
    }

    //Implement the methods declared in demoInterfaceCar
    public boolean slowDown(double amount)
    {
        System.out.println("slowDown amount:" + amount);
        return true;
    }
```

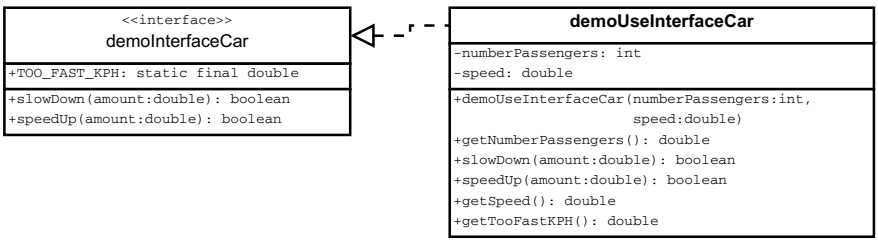

Fig. 15.20 Class diagram for demo interface

```
    public boolean speedUp(double amount)
    {
        System.out.println("speedUp amount:" + amount);
        return true;
    }

    public double getSpeed()
    {
        return speed;
    }
    public double getTooFastKPH()
    {
        return demoInterfaceCar.TOO_FAST_KPH;
    }
}
```

The class diagram in Fig. 15.20 shows the *realizes* relationship between the interface and the implementing class. A single class may implement many Java interfaces. This gives a designer flexibility in creating interfaces that provide for consistent implementation across many classes and packages.

Listing 15.11 shows an abstract class that has one instance variable, one static constant, two abstract methods, and one instance method.

Listing 15.11 Java Abstract Class

```
public abstract class demoAbstractCar
{
    private double speed;
    public static final double TOO_FAST_KPH = 160.9;

    //Abstract methods must be defined using the keyword.
    public abstract boolean slowDown(double amount);
    public abstract boolean speedUp(double amount);

    public double getSpeed()
    {
        return speed;
    }
}
```

Listing 15.12 shows a class that extends this abstract class. The subclass (or child class) must define a concrete method for each of the abstract methods found in the abstract class. As shown in Listing 15.12, an abstract class may have members not associated with the abstract class.

Listing 15.12 Use a Java Abstract Class

```
public class demoUseAbstractCar extends demoAbstractCar {
    private int numberPassengers;

    public demoUseAbstractCar(int numberPassengers,double speed)
    {
        this.numberPassengers = numberPassengers;
        super.speed = speed;
    }

    public int getNumberPassengers()
    {
        return numberPassengers;
    }

    //Implement the abstract methods declared in demoAbstractCar
    public boolean slowDown(double amount)
    {
        System.out.println("slowDown amount:" + amount);
        return true;
    }
    public boolean speedUp(double amount)
    {
        System.out.println("speedUp amount:" + amount);
        return true;
    }

    public double getTooFastKPH()
    {
        return demoAbstractCar.TOO_FAST_KPH;
    }
}
```

The class diagram in Fig. 15.21 shows the inheritance relationship between the abstract class and the subclass. A single class may extend a single (abstract or concrete) class.

15.6.1.1 Comparing Use of Interface and Abstract Class

The class diagrams in Figs. 15.20 and 15.21 result in the same list of publicly available methods. A programmer would instantiate a demoUseInterfaceCar or demoUseAbstractCar object to gain access to the methods (getNumberPassengers, slowDown, speedUp, getSpeed, and getTooFastKPH) representing the behavior of this automated car. The key differences in these two designs, which highlight the differences between a Java interface and a Java abstract class, are as follows.

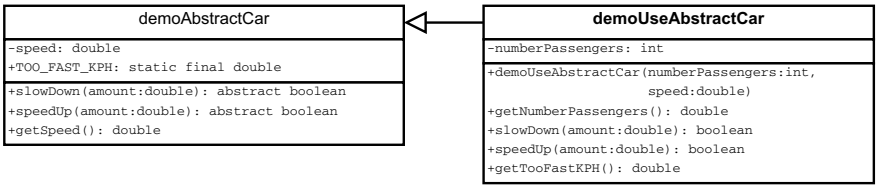

Fig. 15.21 Class diagram for demo abstract class

- The abstract class has a protected instance variable (speed) not found in the interface.
- The abstract class has a public method (getSpeed) not found in the interface.
- The class implementing the interface has a private instance variable (speed) not found in the class extending the abstract class.
- The class implementing the interface has a public method (getSpeed) not found in the class extending the abstract class.

Use an abstract class when you want to implement common behavior in a super class and also have no need to instantiate an object of the super class. Use an interface when you want to describe common behavior that should be included in many classes.

15.7 Post-conditions

The following should have been learned when completing this chapter.

- You understand how to apply the Model–View–Controller architectural pattern to modify an existing object-oriented design into these three components.
- You understand how to apply the Model–View–Controller architectural pattern to create a high-level design that accurately reflects the domain requirements while satisfying the design constraints inherent in MVC.
- You understand the six characteristics of a good software design: simplicity, coupling, cohesion, information hiding, performance, and security. You have evaluated a software design using these criteria.
- You understand that a UML class diagram, UML package diagram, IDEF0 function model, and data-flow diagram are design models used in object-oriented solutions to illustrate the structure of your software design.
- You understand that an IDEF0 function model, data-flow diagram, UML communication diagram, and UML statechart are design models used to illustrate the behavior of your software design.
- You have created and/or modified models that describe an object-oriented software design. This includes designing using a bottom-up and top-down approach.

Exercises

Discussion Questions

1. How is coupling between modules improved with a good implementation of MVC?
2. How is cohesion within a module improved with a good implementation of MVC?
3. How is information hiding improved with a good implementation of MVC?

Hands-On Exercises

1. Use an existing code solution that you've developed, develop design models that show how your solution could be redesigned to use MVC. Apply the software design criteria to your design models. How good or bad is your design?
2. Use your development of an application that you started in Chap. 3 for this exercise. Modify your design to use MVC and then evaluate your MVC design using the six software design criteria.
3. Continue Hands-on Exercise 3 from Chap. 12 by changing your software design to use MVC. Be sure to develop design models to illustrate the structure and behavior of your design. The list below serves as a reminder of the application domain you may have chosen for this exercise. Refer to the Hands-on Exercises in Chap. 12 for details on each domain.

   - Airline reservation and seat assignment,
   - Automated teller machine (ATM),
   - Bus transportation system,
   - Course-class enrollment,
   - Digital library,
   - Inventory and distribution control,
   - Online retail shopping cart,
   - Personal calendar, and
   - Travel itinerary.