OOD Case Study: Design Patterns

28

The objective of this chapter is to apply design patterns introduced in Chap. 27 to the case studies.

28.1 OOD: Preconditions

The following should be true prior to starting this chapter.

- You understand the ways in which a software design pattern is described and you understand the benefits of using software design patterns.
- You understand the six characteristics of a good software design: simplicity, coupling, cohesion, information hiding, performance, and security.
- 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 have created and/or modified models that described an object-oriented software design.
- 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.

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_28

28.2 OOD: ABA Design Patterns

We will use the same list of requirements as stated in Chap. 18. These requirements are listed below. The last requirement, *in italics*, was added in Chap. 18.

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 uppercase letters, lowercase 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. *Allow for display of contact data in the address book.*

28.2.1 OOD: ABA GoF Patterns

The design of the ABA described in this chapter includes the implementation of three GoF patterns. The package diagram in Fig. 28.1 shows a view component with an ABA_A_view class implementing the Facade pattern and an ABA_A_viewCommand class implementing the Command pattern [1].

The GoF Facade pattern [1] says a subsystem must provide a single interface to hide the complexities of the subsystem. The package diagram in Fig. 28.1 clearly shows the controller classes having an association only with the ABA_A_view class. The class diagram in Fig. 28.2 shows instance variables allowing this class to call public methods within other view classes and has public methods called by the controller.

The GoF Command pattern [1] in this design encapsulates a user request into an object used by both the view and controller components. The class diagram in Fig. 28.3 shows all the classes in the view component. The ABA_A_viewCommand

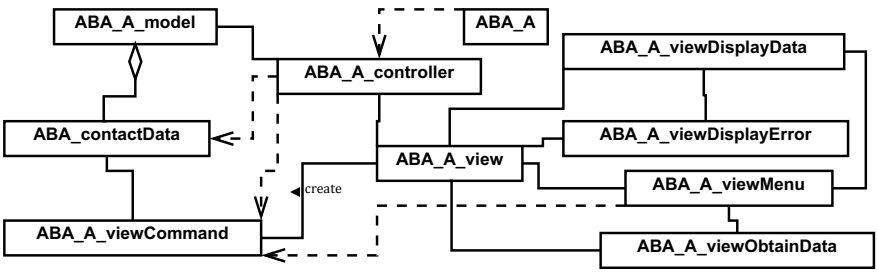

Fig. 28.1 Package diagram—ABA

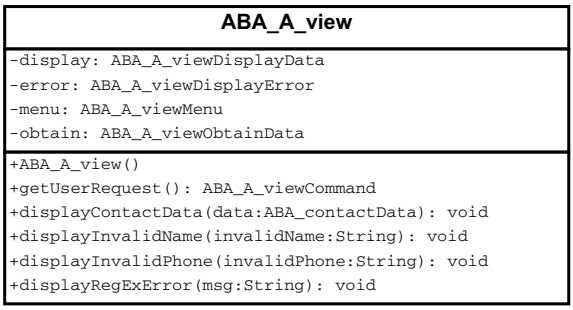

Fig. 28.2 Class diagram—ABA view class

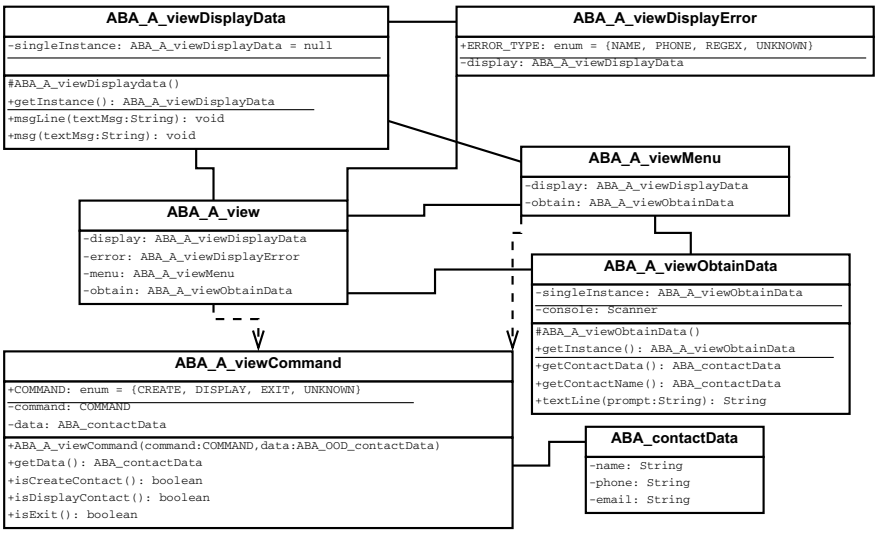

Fig. 28.3 Class diagram—ABA view component

class defines an enumerated data type to represent the three user requests: create, display, and exit. The data instance variable in this class is used to store the contact data associated with the user request. For a create command, the user enters all three values—name, phone, and email. For a display request, the user enters only a contact name. No additional user data is needed for the exit command.

The third GoF pattern used in this design is the Singleton pattern [1]. Two classes in the view component—ABA_A_viewDisplayData and ABA_A_viewObtainData—implement the Singleton pattern. The class diagram in Fig. 28.3 and Listing 28.1 shows the instance variables, constructor, and getInstance methods for the ABA_A_viewObtainData class. The private singleInstance variable is initialized to null and is used to store the single object instance. The constructor method in these classes is protected, while the public static getInstance method is used to return the single object instance, which is created in this method when the singleInstance vari-

able is null. Use of the Singleton pattern for these two classes makes sense. The ABA_A_viewDisplayData class is used by the view and viewDisplayError classes, while the ABA_A_viewObtainData class is used by the view and viewMenu classes.

Listing 28.1 ABA MVC Design—Singleton pattern

```
private static ABA_A_viewObtainData singleInstance = null;
private Scanner console;
private final String EXIT = "exit";

//pre: The single instance has not yet been constructed.
//post: singleInstance is now instantiated.
protected ABA_A_viewObtainData()
{
  console = new Scanner(System.in);
}

//pre: Someone needs the single instance for this class.
//post: Returns single object instance of this class.
public static ABA_A_viewObtainData getInstance()
{
  if (singleInstance == null)
    singleInstance = new ABA_A_viewObtainData();
  return singleInstance;
}
```

28.2.2 OOD: ABA Larman Patterns

The design of the ABA described in this chapter has about the same level of coupling between the controller and view as the solution described in Chap. 18, which was used as the staring point for this chapter's design. Larman's Low Coupling pattern [2] has been effectively implemented between the controller and view.

Cohesion within the view component has been improved (i.e., Larman's High Cohesion pattern [2] has improved the cohesion of classes in the view when compared to the Chap. 18 design). The specific changes to the view component in this chapter which increases cohesion are as follows.

ABA_A_viewMenu: This class is used to display and obtain a valid menu choice. This logic was part of the view class in the Chap. 18 design.

ABA_A_viewDisplayError: This class is used to display error messages. This logic was part of the view class in the Chap. 18 design.

ABA_A_view: The logic removed from this class, as described in the previous two items, has increased the cohesion of this class. As described above, the GoF Facade pattern has been implemented in this class. This resulted in creating two classes—viewMenu and viewDisplayError—just described.

28.2.3 OOD: ABA Fernandez Patterns

The design of the ABA described in this chapter does not include any security controls or mechanisms. We'll refer to the case study in Chap. 25 to discuss security patterns. First, Chap. 25 added three security requirements, listed below.

1. When the ABA starts, the user shall create a password. The password must contain at least eight characters with at least one uppercase letter, one lowercase letter, one digit, and one special character from the list ".!?@#$%&*_+-=".
2. Each request to create or display contact data shall first require the user to reenter their password. Three incorrect entries of the user's password shall result in the ABA exiting.
3. The data shall be non-persistently stored in encrypted form. A symmetric cryptographic algorithm shall be used.

These requirements resulted in a design which adheres to three security patterns described by Fernandez. The first pattern implemented in Chap. 25 is Authenticator [3]. The first two security requirements listed above describe the ABA password policy and when a user must authenticate. As described in Table 25.1, the security design principle complete mediation has been implemented since the user must reenter their password before being allowed to create or display a contact. The Authenticator pattern describes a solution which provides a single entry point into an application/system. In the case of the ABA design in Chap. 25, the user must create their password when the ABA starts. The user is not allowed to use the ABA until they've entered a valid password. In addition, the user must reenter their password each time they wish to create or display contact data.

The second pattern implemented in Chap. 25 is Symmetric Encryption [3]. The third security requirement listed above describes the ABA encryption policy when storing contact data. The encryption and decryption is done in the model component. Two classes shown in the model class diagram in Fig. 28.4 represent the implementation of the Symmetric Encryption pattern within a stand-alone application.

The ABA_A_modelGenKey constructor method generates a secret key used by the AES symmetric cryptographic algorithm. The ABA_A_modelCrypto class has encrypt and decrypt methods that will obtain the secret key via the getKey method. The model component receives an ABA_contactData object from the controller and creates an ABA_modelCipherData object by encrypting the phone number and email address. The model component stores ABA_modelCipherData objects in its memory-based data structure. When a user requests display of contact data, the contact name is given to the model in plaintext form. This is used to find the contact data in the data structure. When the name is found, the model component will call the decrypt method twice, resulting in an ABA_contactData object being created and given to the controller for display by the view.

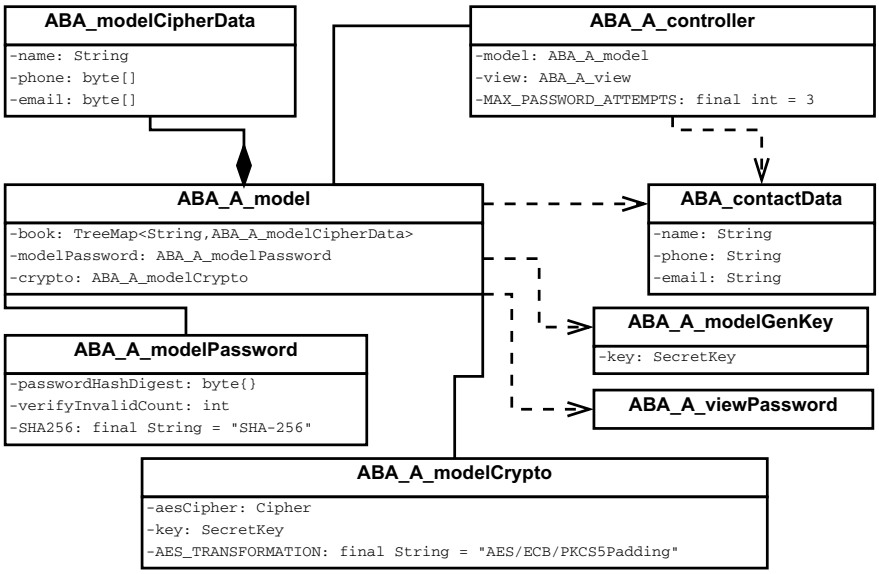

Fig. 28.4 ABA security design: Chap. 25 class diagram model component

The third pattern implemented in Chap. 25 is Secure Model–View–Controller [3]. Storing the user's password as a hash digest and storing the phone number and email address in encrypted form represents two significant security controls designed into the MVC framework.

28.3 OOD Top-Down Design Perspective

We'll use the personal finances case study to reinforce design choices when using design patterns in a top-down design approach.

28.3.1 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.

28.3.2 OOD Personal Finances: Design Patterns

Each subsection below briefly describes the use of design patterns (from [1–3]) for each object-oriented design discussed for the personal finances (PF) case study.

28.3.2.1 Design Patterns for Chap. 15 PF MVC

We'll describe possible design patterns based on the design described in Sect. 15.5. Based on the package diagram in Fig. 15.13, the following patterns could be used in the PF MVC design.

GoF Facade: The model and view classes should implement the GoF Facade pattern since classes in the controller component have an association relationship only with these two classes. That is, these two classes will hide the complexity of the other classes in the respective component.

GoF Singleton: The viewInput and viewOutput classes should implement the GoF Singleton pattern since we only need one instance of each of these to support the user interface.

Larman's Creator: The model component creates and owns many object instances representing different types of data stored in the PF application. This is expressed in the package diagram via the composition relationships between classes in this component. For example, the Finances class has a composition relationship with Account, Label, and Report classes, while the Account class has a composition relationship with the Transaction class. A composition relationship means the Finances class is responsible for creating and storing Account, Label, and Report objects while the Account class is responsible for creating and storing Transaction objects. Larman's description of his Creator pattern matches this description, resulting in this pattern being applicable to the PF MVC design.

GoF Factory: The number of different types of objects to be created and stored in the model component, as just described for Larman's Creator pattern, suggest we should use a GoF creational pattern to give our model component design more flexibility. The GoF Factory pattern could be used in this design. In this case, the

model component would have an abstract or concrete class with a factory method. This factory method would have a parameter indicating the type of object (e.g., Account, Label, Report, or Transaction) to create.

28.3.2.2 Design Patterns for Chap. 18 PF TUI

We'll describe possible design patterns based on the design described in Sect. 18.5. Based on the class diagram in Fig. 18.8, the following patterns could be used in the PF TUI design.

GoF Facade: The view class should implement the GoF Facade pattern since this class represents the interface to the entire view component.

GoF Singleton: The viewInput and viewOutput classes should implement the GoF Singleton pattern since we only need one instance of each of these to support the text-based user interface. The class diagram in Fig. 28.5 shows the design changes to these two classes.

GoF Command: The viewUserRequest class encapsulates a user action/request within the personal finances application. A viewUserRequest object is returned to the controller component (see definition of the getRequest method in the view class) for further processing of the user request.

Larman's Creator or GoF Factory: As described above for the Chap. 15 PF MVC design, either of these patterns could be used in the view component. The class diagram in Fig. 28.5 shows the viewUserData class as a superclass of the viewAccount, viewLabel, viewReport, and viewTransaction classes. This suggests use of the GoF Factory pattern as described above for the PF MVC design patterns.

28.3.2.3 Design Patterns for Chap. 21 PF GUI

We'll describe possible design patterns based on the design described in Sect. 21.4.1. Based on the class diagram in Fig. 21.9, the following patterns could be used in the PF GUI design.

GoF Facade: The view class should implement the GoF Facade pattern since this class represents the interface to the entire view component.

GoF Singleton: The viewFinancesGUI class should implement the GoF Singleton pattern since we only need one instance of this to support the graphical-based user interface. The class diagram in Fig. 28.6 shows the design changes to this class. Note that the viewTransGUI class should not implement the Singleton pattern since an Account will need to display many Transactions.

GoF Command: The viewUserRequest class encapsulates a user action/request within the personal finances application. A viewUserRequest object is given to the controller component (as part of the processRequest method in the view class) for further processing of the user request.

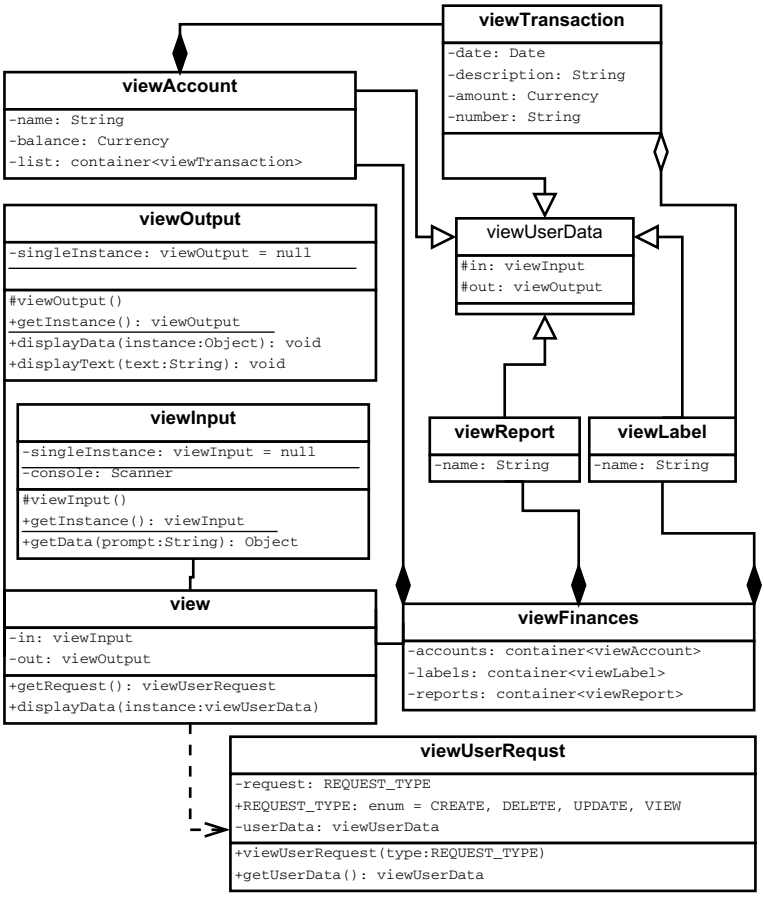

Fig. 28.5 PF Class Diagram: Chap. 18 updated with patterns

Larman's Creator or GoF Factory: As described above for the Chaps. 15 and 18 designs, either of these patterns could be used in the view component.

28.3.2.4 Design Patterns for Chap. 25 PF Security

We'll describe possible design patterns based on the design described in Sect. 25.3. Based on the security design description found in Sect. 25.3.2, the following security patterns described by Fernandez should be used in the PF security design.

Authenticator: Use of a pin or password and two-factor authentication. The description of a distributed version of the personal finances application would result in using two Fernandez patterns: Remote Authenticator/Authorizer and

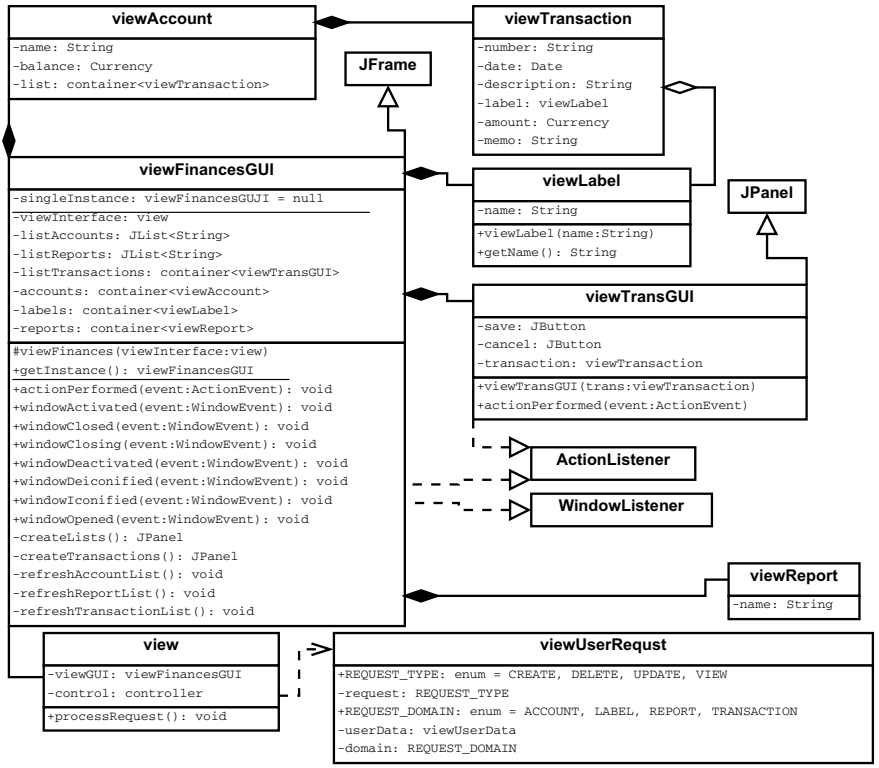

Fig. 28.6 PF Class Diagram: Chap. 21 updated with patterns

Credential. Both of these patterns address authentication in a distributed environment.

Symmetric Encryption: The encryption of data that is persistently stored.

Asymmetric Encryption: The description of a distributed version of the personal finances application suggests using asymmetric encryption between distributed PF components to securely agree to a secret key. This secret key is then used by a symmetric algorithm to encrypt and decrypt data in motion between distributed PF components.

Secure Model–View–Controller: The focus here is on ensuring that there are adequate security controls between the tiers (i.e., components) of the PF application.

Digital Signature with Hashing: The description of a distributed version of the PF application should include each PF component providing evidence proving that they are whom they say they are. We do not want a malicious client to gain access to server data, and we do not want a legitimate client interacting with a malicious server. Designing this pattern into the PF application should greatly reduce this risk.

28.4 OOD: Post-conditions

The following should have been learned when completing this chapter.

- You understand the ways in which a software design pattern is described and the benefits of using software design patterns.
- You've applied design patterns to a software design.
- 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 thinking about design from the bottom-up and from the top-down.

Exercises

Hands-on Exercises

1. Use an existing code solution you've developed, develop design models that utilize one or more design patterns. Apply the six characteristics of a good design to your design models. How good or bad is your design?
2. Use your development of an application you started in Chap. 3 for this exercise. Modify your design to use some design patterns, and then evaluate your design using the six characteristics of a good design.
3. Continue Hands-on Exercise 3 from Chap. 12 by developing a design that utilizes one or more design patterns. 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
- Travel itinerary

References

1. Gamma E, Helm R, Johnson R, Vlissides J (1995) Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley
2. Larman C (2002) Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, 2nd edn. Prentice Hall
3. Fernandez EB (2013) Security Patterns in Practice: Designing Secure Architectures Using Software Patterns, 1st edn. Wiley