OOD Case Study: Transition to Software Design

12

The objective of this chapter is to apply the characteristics of a good software design to the case study and to introduce additional design models that may be used to represent different levels of design abstraction.

12.1 OOD Preconditions

The following should be true prior to starting this chapter.

- You have been introduced to six software design characteristics—simplicity, coupling, cohesion, information hiding, performance, and security.
- 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 five criteria.
- You know that a class diagram is used in object-oriented solutions to illustrate the structure of your program design.
- You know that a Nassi–Shneiderman diagram and a statechart are models used to illustrate the behavior of your program design.

12.2 OOD Transition to Software Design

Our strategy in learning more about how to express a software design is to introduce selected design models that will help us evaluate the ABA based on the six software design characteristics. We will evaluate the Chap. 9 ABA Version B solution described in Sect. 9.2.2. Our choice of design models will consider ways to produce abstractions of the ABA that accurately represent its structure and/or behavior.

© Springer Nature Switzerland AG 2020
D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_12

12.2.1 OOD ABA Structure Design Models

The class diagram for the Chap. 9 ABA Version B solution is shown in Fig. 12.1a. The class diagram design model was introduced in Chap. 3 and used in Chaps. 6 and 9. The only change shown in Fig. 12.1a is the use of a plus sign (+), shown immediately to the left of each attribute and method name, to indicate that the class member has public access.

Another modeling technique that shows the structure of an object-oriented design is a UML (Unified Modeling Language) package diagram. The package diagram for the Chap. 9 ABA Version B solution is shown in Fig. 12.1b. The three classes shown in Fig. 12.1a are now contained within a package notation. This package is unnamed since no *package* statement was used in the Java source code file (i.e., ABA_B_solution.java).

12.2.2 OOD ABA Structure and Behavior Design Models

IDEF0 is a function modeling technique that shows both the structure and behavior of a design. Figure 12.2 shows two activities performed by the Chap. 9 ABA Version B solution. The IDEF0 modeling technique uses four types of arrows:

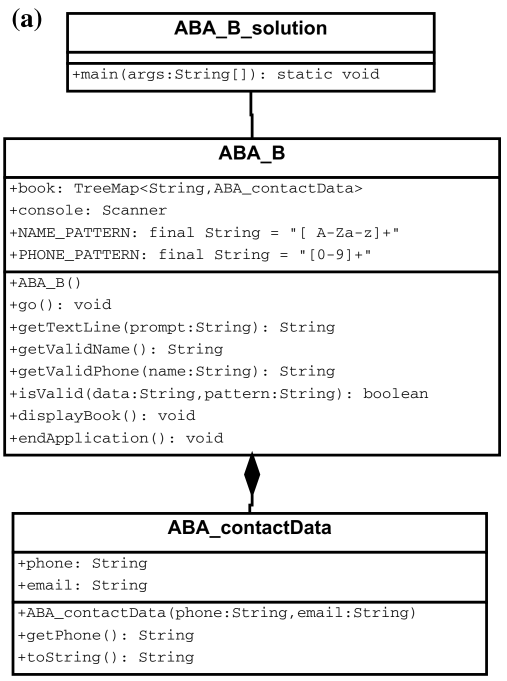
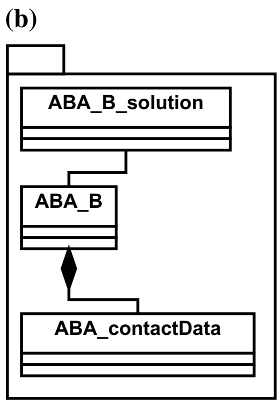

(a)
ABA_B_solution
+main(args:String[]): static void
ABA_B
+book: TreeMap<String,ABA_contactData>
+console: Scanner
+NAME_PATTERN: final String = "[A-Za-z]+"
+PHONE_PATTERN: final String = "[0-9]+"
+ABA_B()
+go(): void
+getTextLine(prompt:String): String
+getValidName(): String
+getValidPhone(name:String): String
+isValid(data:String,pattern:String): boolean
+displayBook(): void
+endApplication(): void
ABA_contactData
+phone: String
+email: String
+ABA_contactData(phone:String,email:String)
+getPhone(): String
+toString(): String


Fig. 12.1 **a** Class diagram. **b** Package diagram

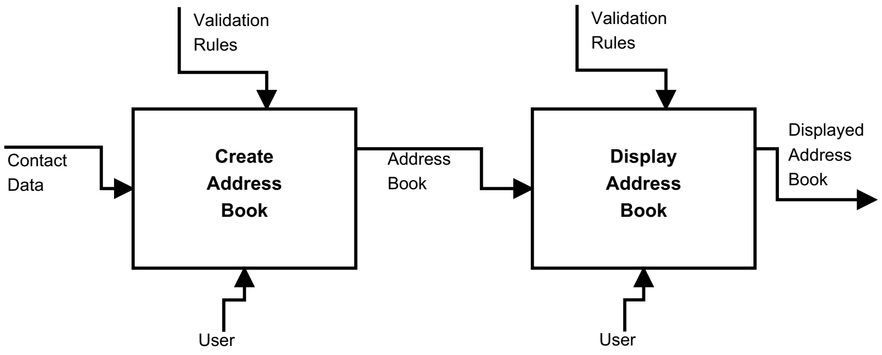


Fig. 12.2 IDEF0 function model for Chap. 9 ABA version B

- Input arrow: The data or objects that are transformed by the function into output. An input arrow points to the left side of an activity notation.
- Control arrow: The conditions required to produce correct output. Data or objects modeled as controls may be transformed by the function, creating output. A control arrow points to the top side of an activity notation.
- Output arrow: The data or objects produced by a function. An output arrow emanates from the right side of an activity notation.
- Mechanism arrow: The means used to perform a function. A mechanism arrow points to the bottom side of an activity notation.

In Fig. 12.2, *Contact Data* input by a *User* is used to create the memory-resident *Address Book* that is output from the first activity. The second activity then formats and displays this data to the *User*. *Validation Rules* are used by both activities to control the storage and display of valid *Address Book* data.

The definitive source of information on IDEF0 is the FIPS183 standard document [1]. As is typical of standards documents, FIPS183 is dense and difficult to read. Refer to Wikipedia [2] or idef.com [3] for additional information on IDEF0.

A UML communication diagram will show the structure and behavior of a design. A UML communication diagram shows interactions between objects via a sequence of messages (i.e., method calls). Figure 12.3 shows the UML communication diagram for the Chap. 9 ABA Version B solution. The ABA_OOP_B_solution object, whose class contains the main method, will construct a ABA_OOP_B object and then call its go method. The messages that start at the ABA_OOP_B object back to itself represent the logic performed by the go method. The numbering of the messages, 1.1, 1.2, 1.3, 1.6, and 1.7 for the messages sent from ABA_OOP_B to itself, and 1.4 and 1.5 for messages from ABA_OOP_B to the TreeMap<String,ABA_OOP_contactData> object, indicate the order in which these methods are called by the go method.

A UML communication diagram also shows relationships between the classes represented by the objects shown in the model. For example, the line connecting the ABA_OOP_B and TreeMap<String,ABA_OOP_contactData> objects, along with

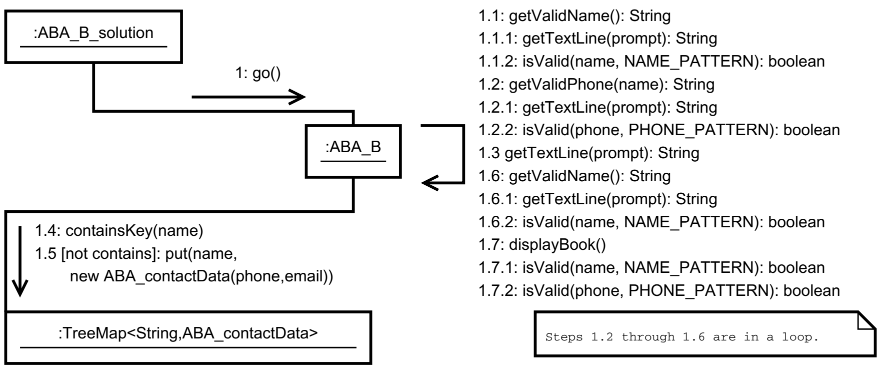

:ABA_B_solution
1: go()
:ABA_B
1.1: getValidName(): String
1.1.1: getTextLine(prompt): String
1.1.2: isValid(name, NAME_PATTERN): boolean
1.2: getValidPhone(name): String
1.2.1: getTextLine(prompt): String
1.2.2: isValid(phone, PHONE_PATTERN): boolean
1.3 getTextLine(prompt): String
1.6: getValidName(): String
1.6.1: getTextLine(prompt): String
1.6.2: isValid(name, NAME_PATTERN): boolean
1.7: displayBook()
1.7.1: isValid(name, NAME_PATTERN): boolean
1.7.2: isValid(phone, PHONE_PATTERN): boolean
1.4: containsKey(name)
1.5 [not contains]: put(name, new ABA_contactData(phone,email))
:TreeMap<String,ABA_contactData>
Steps 1.2 through 1.6 are in a loop.

Fig. 12.3 UML communication diagram for Chap. 9 ABA version B

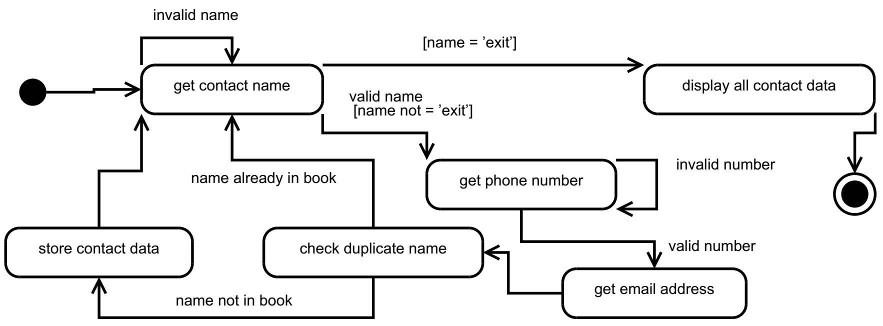


Fig. 12.4 Statechart for Chap. 9 ABA version B

the definition of the TreeMap, suggests that the ABA_OOP_B class is used as a container of many ABA_OOP_contactData objects.

12.2.3 OOD ABA Behavior Design Models

Figure 12.4 shows the statechart for the Chap. 9 ABA Version B solution. The statechart design model was introduced in Chap. 3 and used in Chaps. 6 and 9. Note that a statechart is also known as a *state diagram* or *state machine diagram* [4].

12.2.4 OOD Evaluate ABA Software Design

We'll use the six characteristics of a good software design described in Chap. 11 to evaluate the software design described above for the Chap. 9 ABA Version B solution.

12.2.4.1 Simplicity

In Chap. 11, simplicity is defined as *the amount and type of software elements needed to solve a problem*. We'll evaluate the ABA software design models described above based on this definition.

The UML class diagram in Fig. 12.1a contains only three classes. One of these classes contains the main method needed to run the Java program while another class is used as a container for a contact person's phone and email. There is one association and one composition relationship in this class diagram. From a class diagram perspective, this is a simple object-oriented design.

The UML package diagram in Fig. 12.1b shows one package containing the entire solution. From a package diagram perspective, this is a simple object-oriented design.

An IDEF0 function model includes five types of software elements: activity, input arrow, control arrow, output arrow, and mechanism arrow. Does the IDEF0 function model in Fig. 12.2 provide an accurate abstraction while minimizing the number of elements needed to express the design? With two activities and eight arrows showing inputs, controls, outputs, and mechanisms (ICOM), this IDEF0 function model describes two primary processing steps performed by the solution. In counting the ICOM arrows, note that both control arrows represent the same *Validation Rules*, both mechanism arrows represent the same User, and the *Address Book* arrow output from the first activity is then input into the second activity. Thus, there are only five distinct ICOM arrows in the diagram. Based on this brief analysis, this IDEF0 function model appears to be a simple abstraction of the solution.

The UML communication diagram in Fig. 12.3 shows the objects and messages involved in the Chap. 9 ABA Version B solution. The list of messages associated with the ABA_OOP_B object is rather extensive, and perhaps too detailed, resulting in this design diagram matching the details found in the code. Figure 12.5 shows a simpler diagram where the messages that are three-method-calls deep (e.g., 1.1.1, 1.1.2, 1.2.1, ...) have been removed. Removing this level of detail results in a design model that

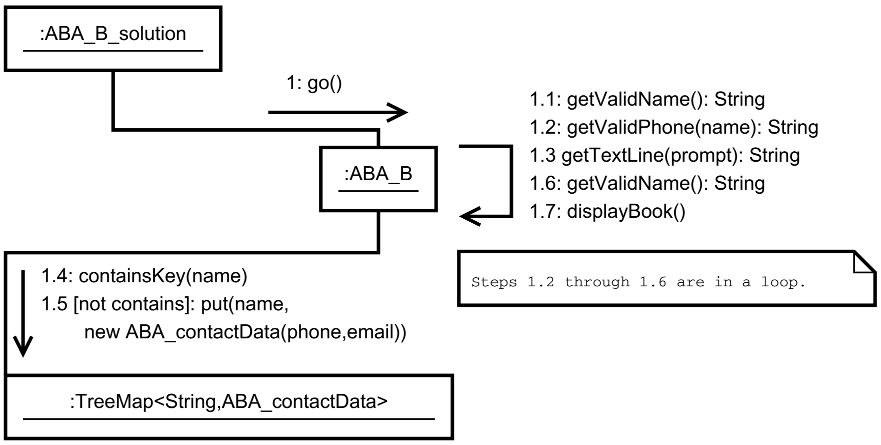

:ABA_B_solution
1: go()
:ABA_B
1.1: getValidName(): String
1.2: getValidPhone(name): String
1.3 getTextLine(prompt): String
1.6: getValidName(): String
1.7: displayBook()
Steps 1.2 through 1.6 are in a loop.
1.4: containsKey(name)
1.5 [not contains]: put(name,
new ABA_contactData(phone,email))
:TreeMap<String,ABA_contactData>

Fig. 12.5 Simpler UML communication diagram for Chap. 9 ABA version B

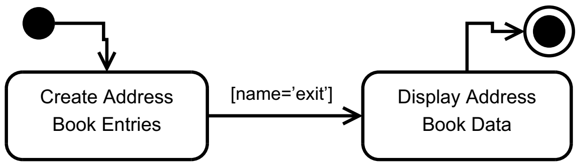


Fig. 12.6 More abstract statechart for Chap. 9 ABA version B

accurately reflects the code while abstracting away some of the details. Given the names of the messages (i.e., methods), this simpler design model clearly shows the overall design, where name and phone data are validated and stored in a TreeMap. The only vagueness in the 12.5 diagram is the message *1.3 getTextLine(prompt): String*, which does not clearly indicate the data value expected to be entered by the user. Note that creating another UML communication diagram that eliminates the second level of messages (e.g., 1.1, 1.2, ..., 1.7) to make the diagram even simpler would result in a diagram that does not convey any significant design information. The TreeMap object would be eliminated and the information about validating name and phone would no longer be expressed in the diagram.

A UML state machine diagram contains two types of software elements: states and transitions. Does the state machine in Fig. 12.4 provide an accurate abstraction while minimizing the number of elements needed to express the design? There are 6 states (8 if you count the initial and final states) and 11 transitions in the statechart, which shows the important behavior and user interactions of the ABA. Overall, this diagram seems fairly easy to understand. However, it does not use abstraction to hide many details. Figure 12.6 shows a state machine with only two states and three transitions. This diagram expresses a high-level design similar to the IDEF0 function model in Fig. 12.2. A benefit of the UML state machine modeling technique, when compared with the older statechart, is the option of creating substates (also known as sub-machines). In essence, we can create substates to represent a hierarchy of state machines. Figure 12.7 shows a substate diagram for the *Display Address Book Data* state that is shown in Fig. 12.6. The fact that a designer can create a hierarchy of state machines provides flexibility in how abstraction may be used to represent levels of software design. As shown in Figs. 12.6 and 12.7, a designer may choose to drill down on those states where it may be necessary to show a more detailed design.

12.2.4.2 Coupling

In Chap. 11, coupling is defined as *the degree to which each program module relies on other modules; the "connectedness" of a module to other modules; or the amount of interdependence between two or more modules*. We'll evaluate the ABA software design models described above based on this definition.

The UML class and package diagrams in Fig. 12.1a, b shows two relationships between the three classes. The association between the ABA_OOP_B_solution and ABA_OOP_B classes represents the main method constructing a ABA_OOP_B object and then calling it's go method. The composition relationship between the ABA_OOP_B and ABA_OOP_contactData classes represents the nonpersistent storage of address book data. These relationships represent the coupling in the Chap. 9 ABA Version B solution.

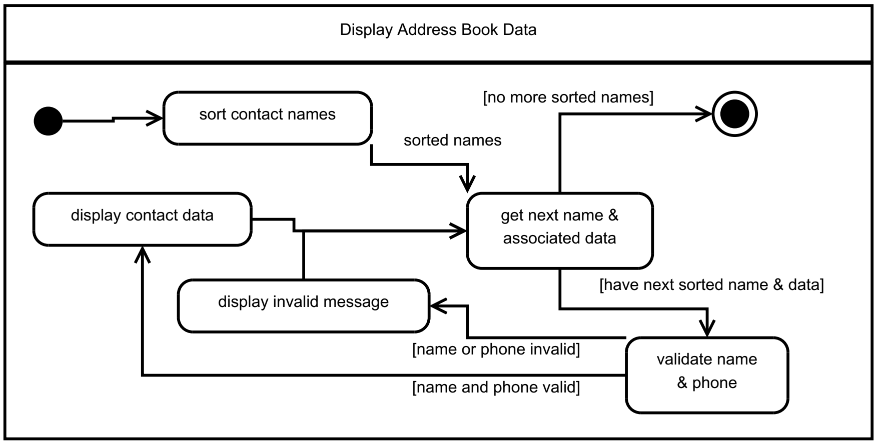


Fig. 12.7 Sub-Machine for Display Address Book Data for Chap. 9 ABA Version B

The IDEF0 function model in Fig. 12.2 shows one output flow (*Address Book*) from activity *Create Address Book* that flows as input into activity *Display Address Book*. This sharing of the address book data is the only coupling that exists between the two activities. Note that the control arrow *Validation Rules* and the mechanism *User* are associated with both activities. However, the *Validation Rules* are not transformed into the output, they are used to control the production of the output, and having the *User* associated with each activity simply denotes the role a person has when using the ABA.

The UML communication diagram in Fig. 12.5 shows object coupling that mirrors the class coupling expressed in the class and package diagrams. The UML communication diagram shows the association relationship between ABA_OOP_B_solution and ABA_OOP_B. The composition relationship between ABA_OOP_B and ABA_OOP_contactData is implied by the definition of the TreeMap data structure and the *1.5 [not contains] put(...)* message. Message numbers 1, 1.4, and 1.5 show message coupling between the three classes/objects, which represents a low form of coupling.

The UML state machine diagram in Fig. 12.6 shows coupling between the *Create Address Book Entries* and *Display Address Book Data* states. The exact nature of this coupling is not as evident as it is in the IDEF0 function model.

12.2.4.3 Cohesion

In Chap. 11, cohesion is defined as *the degree to which a software module is strongly related and focused in its responsibilities; a module that has a small, focused set of responsibilities and single-mindedness; or an attribute of a software unit that refers to the relatedness of its components*. We'll evaluate the ABA software design models described above based on this definition.

The UML class diagram in Fig. 12.1a can be used to assess the cohesion of each class since it shows the attributes and operations of each class. Looking at the ABA_OOP_B class, we see that it has two public attributes—book and console—that are used to store the address book data and obtain data from the user, respectively. The operations in this class clearly show some processing related to obtaining data from the user and validating this data. It also shows that this class is responsible for displaying the address book data. Essentially, the ABA_OOP_B class is responsible for all of the requirements stated in Chap. 9 for ABA Version B. Thus, this class has low cohesion. In contrast, the ABA_OOP_contactData class is responsible for containing data (phone and email) for one contact name. Given the attributes and operations in this class, this class is highly cohesive.

The UML package diagram in Fig. 12.1b does not show cohesion at the class level since attributes and operations are not being shown. However, we do see that all three classes are part of a single package. So we can ask, is this one package highly cohesive? Since all three classes combined represent the entire solution for the Chap. 9 ABA Version B, we could argue that this package is highly cohesive. We will revisit the notion of cohesion in a UML package diagram in Chap. 15.

For the IDEF0 function model in Fig. 12.2, we consider each activity to be a separate module. We need to determine if each IDEF0 activity has a singular focus (i.e., has strong cohesion). One way to address this is to identify the Chap. 9 ABA Version B requirements that apply to each of the IDEF0 activities. The following requirements are part of the *Create Address Book* activity.

- Allow for entry and (nonpersistent) storage of people's names.
- Store for each person a single phone number.
- Use a simple text-based user interface to obtain the names and phone numbers.
- 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.
- 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.
- Prevent a duplicate name from being stored in the address book.

The *Create Address Book* activity implements all of the requirements stated for the Chap. 9 ABA Version B. We can react to this in one of two ways. We could argue that the IDEF0 function model is at such a high level of abstraction that it consolidates all of the requirements into a single activity. On the other hand, including user interactions, validation of data, and storage of valid data into one activity makes the activity have low cohesion. Given the above discussion on the cohesive properties shown in the class diagram, we conclude that we have a single IDEF0 activity that represents too much of the entire ABA.

Looking at the UML communication diagram in Fig. 12.5 results in an analysis of cohesion similar to the class diagram. The messages numbered 1.1 through 1.7 shows that the ABA_OOP_B object has methods that obtain data from the user, validate this data, and displays the address book data. This supports the conclusion from the class diagram that the ABA_OOP_B class exhibits low cohesion. The two messages (i.e., 1.4 and 1.5) going to the TreeMap data structure shows that the ABA_OOP_contactData class has a singular focus of containing (phone and email) for one contact name. This also supports the conclusion from the class diagram that the ABA_OOP_contactData class is highly cohesive.

The UML state machine in Fig. 12.6 results in an analysis of cohesion similar to the IDEF0 function model. A single state—*Create Address Book Entries*—represents all of the requirements stated for the Chap. 9 ABA Version B. In this case, we can develop a sub-machine for this state that would have shown all of the states and transitions that fulfill the ABA requirements. This sub-machine would consist of states that are individually more cohesive since they represent a more detailed view of the ABA processing being performed. Thus, the hierarchy of state machines as illustrated in Figs. 12.6 and 12.7 uses abstraction to show a high-level machine while also allowing a more detailed view using a sub-machine whose states will likely be more cohesive.

12.2.4.4 Information Hiding

In Chap. 11, information hiding is defined as *having a component hide its implementation details (i.e., algorithms and data) from the other components*.

The UML class diagram shows that all of the attributes and operations have public visibility. This is a poor design since this means that any of these class members may be referenced by code in any other class. For example, code can be written to directly reference the TreeMap data structure. This could result in adding contact data that has not been validated. Similarly, since all of the operations are public, these could be called by code in any other class. None of the processing performed by these methods are hidden.

The UML package diagram, IDEF0 function model, UML communication diagram, and UML state machine design models cannot be evaluated for how well they hide information. This is because these modeling techniques do not directly show how information is used by the design elements.

12.2.4.5 Performance

In Chap. 11, performance is defined as *the analysis of an algorithm to determine its performance in terms of time (speed) and space (memory usage)*.

The UML class and UML communication diagrams both show that a TreeMap is used to store the address book data. As discussed in Chap. 6, this data structure will give us logarithmic time performance. The other design models do not show the data structures being used or algorithmic details to assess performance.

Since the UML state machine diagram shows interactions between the user and application, we could evaluate the user's experience using the state machine. However, evaluating the performance of a human–computer interaction (HCI) design has not yet been discussed. Chapters 17–22 will discuss ways to evaluate an HCI design.

12.2.4.6 Security

In Chap. 11, security is defined as *a set of technical controls intended to protect and defend information and information systems*. We'll evaluate the ABA software design models described above based on this definition.

The class diagram, IDEF0 function model, and UML communication diagram shown in Figs. 12.1a, 12.2, and 12.3, respectively, show data being validated before stored in the address book. Since the case study includes specific requirements about how to validate address book data, and since these design models show an implementation of these requirements, the Chap. 9 ABA Version B design satisfies two security criteria: validate input data and validate output data.

12.2.5 OOD ABA Software Design: Summary

A summary of the OOD design models used along with an evaluation of these models using characteristics for a good software design follows.

12.2.5.1 OOD ABA Design Models

The UML class and package diagrams are used to show the structure of a design. These types of models show the classes involved in the design and how these classes are related to each other. The class relationship types that have been used in our case study are association and composition.

Two modeling techniques show both structure and behavior. A software designer may create IDEF0 function models to show abstractions of the software based on high-level activities and the information flows (i.e., input, control, output, and mechanism) that influence these activities. The UML communication diagram technique shows the interactions of object instances via messages (i.e., method calls). A software designer can ignore certain detailed messages to convey the overall processing flow instead of showing all of the details (which can be observed in the code).

A statechart (or UML state machine) is used to show the behavior of a design. This behavior is represented by states and transitions, and is typically used by a software designer to show the important processing steps without regard to the elements that are responsible for implementing these steps.

12.2.5.2 OOD ABA Software Design Characteristics

The design models that illustrate the structure and behavior of the Chap. 9 ABA Version B solution exhibit the following design characteristics.

- Simple (good),
- Low or weak coupling (good),
- Low or weak cohesion (bad),
- Does not hide information (bad),
- Logarithmic time performance (good), and
- Does input and output validation (good).

12.3 OOD Top-Down Design Perspective

To reinforce the software design concepts covered so far, this section introduces a second case study that will be discussed top-down. Architects in the information technology industry, whether they focus solely on software or on both hardware and software, tend to apply a top-down design when decomposing a large system into high-level design elements (aka subsystems, components). We will first discuss the requirements of a second case study and then present a software design that meets these requirements. The software design is described using models to present abstractions without first discussing code. The intent of this second case study is to demonstrate the development of a software design based solely on the problem domain (and not on any existing implementation details).

12.3.1 OOD Personal Finances: A Second Case Study

The second case study will develop a personal finance software application that keeps track of accounts and transactions. The requirements of this application are as follows.

- 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.

12.3.2 OOD Personal Finances: Structure Design Models

The package diagram in Fig. 12.8 shows a single named package *PersonalFinance* and the classes within this package. This single package represents the classes that describe the scope of the personal finance problem domain.

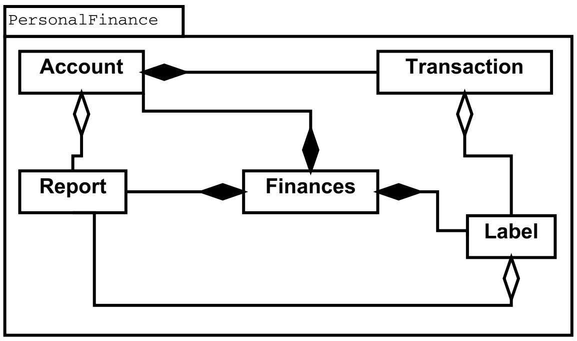


Fig. 12.8 Personal finance package diagram

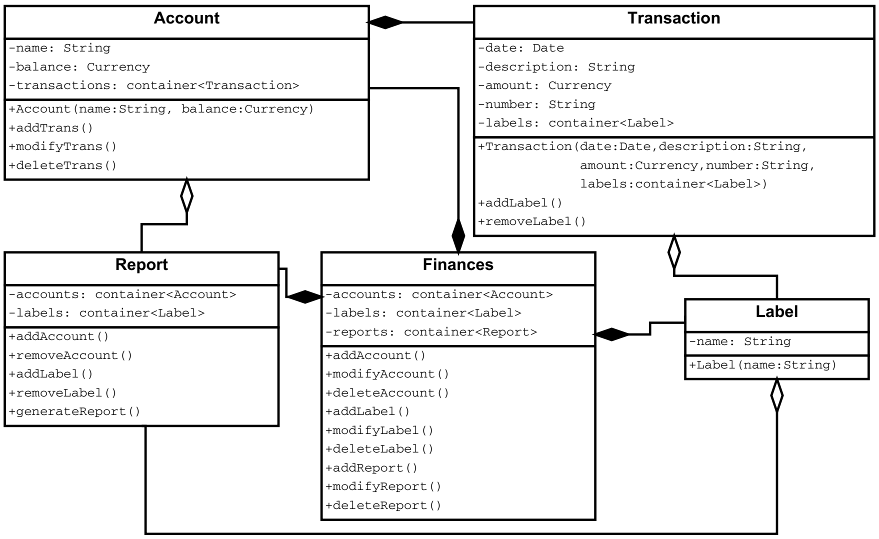


Fig. 12.9 Personal finance class diagram

The class diagram in Fig. 12.9 shows the attributes and public methods for each class shown in the package diagram. This class diagram is at a higher level of abstraction than what we anticipate needing to do with a detailed design or with the implementation. Examples of this abstraction include the following.

- Some attributes use the generic term *container* to represent a data structure that will need to be used to store multiple data values.
- None of the classes 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.

- The classes show the primary objects needed for the personal finance problem domain. At this high-level of abstraction, it is unclear whether these classes are responsible for a user interface or for persistent storage of personal finance data.

Another term that may be used to describe the class diagram in Fig. 12.9 is *domain model*. As just explained, the classes in this diagram represent the important aspects of the personal finance domain as delineated in the requirements listed above. As such, this class diagram represents knowledge about personal finance by showing the key concepts and their relationships to each other. As introduced in Chap. 1, software development processes (SDPs) describe a sequence of steps that, when performed, will produce software artifacts. As you may recall, the generic steps in an SDP include: gather needs, produce design, and implement design. The creation of the domain model class diagram is an example of a software model that accomplishes two things—it explains the needs/requirements and begins to show how the needs may be translated into a design. This illustrates the role of an architect in creating a high-level design, which may also be called an architecture. These high-level design models tend to convey needs/requirements while also serving as a starting point for creating more detailed design models. As we'll see in Chap. 15, while a domain model may represent a high-level view of a software design, these models tend to ignore important technical details that, if left out of more detailed designs, would result in an implementation that is not as robust and flexible in meeting future needs. While the term domain model has been used to describe the class diagram, the terms *business model* or *analysis model* may also be used to describe Fig. 12.9.

12.3.3 OOD Personal Finances: Structure and Behavior Design Models

A data-flow diagram (DFD) is a modeling technique that shows both the structure and behavior of a design. Figure 12.10 shows a high-level data-flow diagram for the personal finance case study. This figure illustrates the use of four notations within a DFD:

Process	A rounded-rectangle shape with a label that explains the processing being performed by this process. Figure 12.10 shows three processes—Account Management, Transaction Management, and Report Management.
Data store	A rectangle shape with a label that describes a persistent data store, e.g., database. Figure 12.10 shows one data store—Personal Finance Database.
External entity	A 3D-square that identifies an entity that exists outside the scope of the system being described. Figure 12.10 shows two external entities—User and Report.

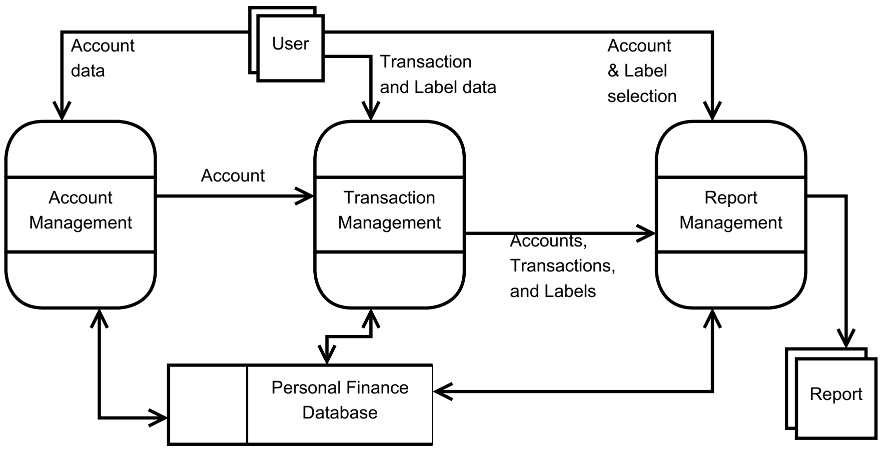

User
Account data
Transaction and Label data
Account & Label selection
Account Management
Account
Transaction Management
Accounts, Transactions, and Labels
Report Management
Personal Finance Database
Report

Fig. 12.10 Personal finance data-flow diagram

Data flow A directed line that connects two other notations in the DFD, showing the flow of information between these two notations. A data flow may have an arrow on one or both ends and may include an optional label. Figure 12.10 shows five data flows each with a label and four data flows that are not labeled.

This DFD shows that Account Management creates/updates accounts which are then used by Transaction Management to record transactions within the account. Report Management would then create reports using the account, transaction, and label data. The data flows to/from the Personal Finance Database shows that all of the data created/updated by the three processes are stored in a persistent data store.

For more information about data-flow diagrams, refer to [5,6], or do a web search to find more recent articles.

The UML communication diagram in Fig. 12.11 shows the structure and behavior of the personal finance case study based on the domain model class diagram shown in Fig. 12.9. This communication diagram shows a typical flow of public method calls that express how the classes/objects interact with each other.

12.3.4 OOD Personal Finances: Behavior Design Models

The statechart diagram in Fig. 12.12 shows some of the user interactions with the personal finance system. This statechart was developed to emphasize the fact that transactions must be associated with an existing account.

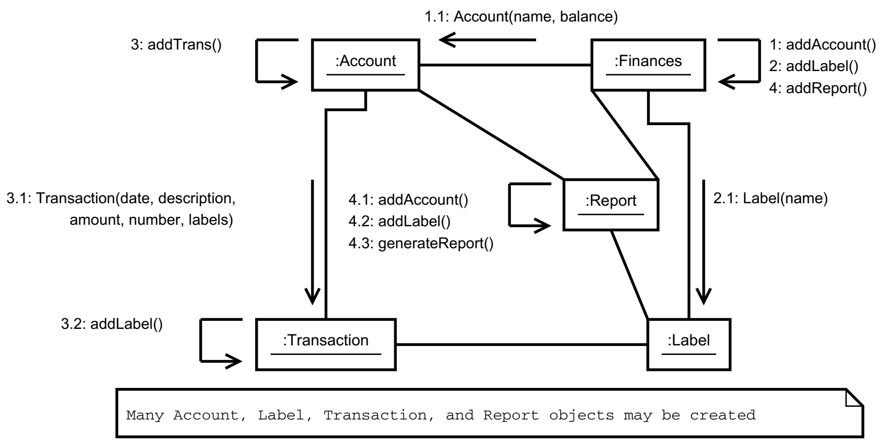


Fig. 12.11 Personal finance communication diagram

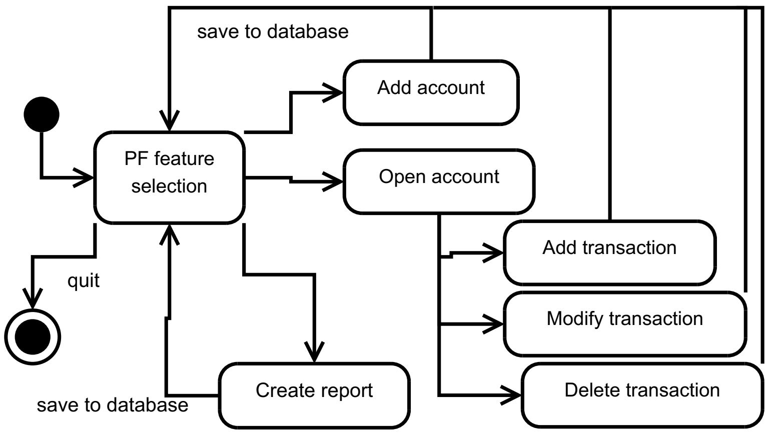


Fig. 12.12 Personal finance statechart

12.3.5 OOD Personal Finances: Summary

A summary of the OOD design models used along with an evaluation of these models using characteristics for a good software design follows.

12.3.5.1 Design Models

The design models shown in Figs. 12.8, 12.9, 12.10, 12.11 and 12.12 show a high-level design that represents an initial understanding of the needs/requirements and how they may be represented within the personal finance domain. Depending on what an architect wants to emphasize, some of these models may be omitted in the early stages of expressing knowledge of the domain. 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, showing interactions between the personal finance system and a user, or showing use of persistent storage, may be important. In these cases, the DFD and statechart would be important to develop.

12.3.5.2 Evaluation

An evaluation of the personal finance design, as shown in Figs. 12.8, 12.9, 12.10, 12.11, and 12.12, 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 that, perhaps, should not be included in a very high-level design.

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 domain classes as described by the requirements. These couplings appear to be necessary given the list of requirements. The DFD shows an abstraction of the class diagram and emphasizes the three primary processes involved in the personal finance software system. Again, the interactions between these three processes appear to be necessary given the case study requirements.

Cohesion As discussed above for the class diagram, these classes represent the domain objects needed to design and implement personal finance software. If any of the classes include user interface or persistent data storage, then the class is responsible for too many distinctly different types of processing. Given this unknown, this high-level object-oriented design appears to have low cohesion (which is bad).

Information hiding Based on this high-level design, and given the lack of clarity about where the user interface and persistent storage design elements reside, one can argue this design effectively hides all implementation details.

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 information security. However, designers and implementers should always consider security even when this is not explicitly stated in the requirements. The high-level design models above do not address data input validation, data output validation, exception handling, fail-safe defaults, or type-safe languages.

12.4 OOD Post-conditions

The following should have been learned when completing this chapter.

- 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 that may be used to illustrate the structure of your software design.
- You understand that an IDEF0 function model, data-flow diagram, UML communication diagram, and UML state machine are design models that may be used to illustrate the behavior of your software design.
- You've created and/or modified design models that describe an object-oriented software design. To improve your learning of the topics presented in this chapter, you should create/modify design models using a bottom-up approach and also using a top-down approach. In a bottom-up approach, start with an implementation and develop design models that accurately reflect the code. In a top-down approach, start with requirements and develop high-level models that express these requirements.

Exercises

Discussion Questions

1. What changes would you make to the Chap. 9 ABA Version B design so that high cohesion is achieved while still having low coupling?
2. Describe a problem statement (i.e., scenario) and have students discuss use of the six characteristics for a good design. Are there situations where a balance (i.e., trade-off) needs to be done between two or more of the characteristics?

Hands-on Exercises

1. Using an existing code solution that you've developed, create design models that abstract away certain implementation details. Apply the characteristics for a good software design 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 improve your evaluation of the six characteristics for a good software design.
3. Select a sample problem domain from the list below, or choose a problem domain of interest to you, and develop a software design that describes structural and behavioral aspects of a potential solution.

Airline reservation and seat assignment An individual may reserve one or more flights where each flight has a from-airport with a departure date and time, and a to-airport with an arrival date and time. For each flight reservation, the individual may reserve one or more seats. Each flight reservation represents a one-way ticket. To complete the purchase of the flight reservations, the individual must provide the name of each passenger, a billing address and phone number, and credit card information.

Automated teller machine (ATM) An individual may deposit or withdraw funds from a bank account, transfer funds between two bank accounts, or obtain the balance of a bank account. An individual may perform as many transactions as they like, and may request a receipt that shows the results of each transaction. Before transactions can start, the individual must provide a valid bank card and pin number. The individual indicates when no more transactions are needed.

Bus transportation system A public transportation system needs to maintain their routes, times, drivers, and fares. This information may be different for weekdays, weekends, and for holidays.

Course-class enrollment An individual may register for one or more class sections. Each class section meets on one or more weekdays and times, has a classroom location, an instructor, and a course description. An individual may register for or drop a class section.

Digital library An individual may reserve or checkout items from an electronic library. The library contains books, movies, songs, and magazines. An individual may change a reservation and may extend their checkout due-date once.

Inventory and distribution control An organization keeps track of their product inventory and coordinates distribution of these products to customers. Each product has a unique identifier, description, dimensions, and weight.

Online retail shopping cart An individual may add and remove items from an electronic shopping cart. Each item in the shopping cart has a price and quantity purchased. To complete the purchase of the items in the shopping cart, the individual must identify the type of shipping requested, and provide their name, shipping address and phone number, billing address and phone number, and credit card information.

Personal calendar An individual wants to maintain their schedule on a calendar. The calendar may be used to show a single day, a week, or a month. The calendar allows the schedule of events to overlap each other.

Travel itinerary An individual needs to create a travel plan to visit N cities or destinations. The individual will identify travel time, distance, and route to get to each destination based on visiting the N locations in a specified order. The individual will also indicate the duration they will stay in each destination. An individual may have the need to create many distinct travel plans.

References

1. National Institute of Standards and Technology: Draft Federal Information Processing Standards Publication 183: Integration Definition for Function Modeling (IDEF0). NIST (1993). http://www.idef.com/wp-content/uploads/2016/02/idef0.pdf. Accessed 21 June 2017
2. Wikipedia.org: IDEF0. In: Wikipedia the free encyclopedia. Wikimedia Foundation (2017). https://en.wikipedia.org/wiki/IDEF0. Accessed 21 June 2017
3. Knowledge Based Systems: IDEF0 Function Modeling Method. Knowledge Based Systems, Inc (2017). http://www.idef.com/idefo-function_modeling_method/. Accessed 21 June 2017
4. Wikipedia.org: State diagram. In: Wikipedia the free encyclopedia. Wikimedia Foundation (2017). https://en.wikipedia.org/wiki/State_diagram. Accessed 23 June 2017
5. Gane C, Sarson T (1977) Structured systems analysis: tools and techniques. McDonnell Douglas Systems Integration Company
6. Yourdon E, Constantine L (1979) Structured design: fundamentals of a discipline of computer program and systems design. Prentice Hall, Upper Saddle River