OOP Case Study: Use Program Design Criteria and Simple Models

3

The objectives of this chapter are to introduce the Address Book Application case study and to apply the program design criteria and object-oriented design models to the case study.

3.1 OOP Preconditions

The following should be true prior to starting this chapter.

- You understand that an object-oriented program contains methods grouped into classes. An entire solution may consist of many classes, where each class contains many methods.
- Program design should be evaluated using at least three criteria: separation of concerns; design for reuse; and design only what is needed.
- Object-oriented design models include: class diagrams, Nassi–Shneiderman diagrams, and statechart diagrams.

3.2 OOP Case Study and Bottom-Up Approach

The same case study is used throughout the remainder of this book to help illustrate software design thinking and alternatives. This case study is a simple address book application (ABA) that will store contact information of people that you know. In this first bottom-up design chapter, the focus is on an ABA that does not store contact information persistently. The contact data is stored only when the application is running; each time the application is started, the address book contains no data.

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_3

Storing data non-persistently was chosen as the first bottom-up design to avoid having to introduce new technologies (e.g., XML, relational database). This should allow you to use the material in this chapter as a review of prior programming knowledge, which hopefully includes program design concepts.

3.2.1 OOP Very Simple Address Book Application (ABA)

This very simple ABA shall

- Allow for entry and nonpersistent storage of people's names.
- Use a simple text-based user interface to obtain the names.
- Not check to determine if a duplicate name was entered.
- Not retrieve any information from the address book.

Both Python and Java code will be used to introduce OOP design concepts.

3.3 OOP Simple Object-Oriented Programming Designs

3.3.1 OOP Version A

A Python solution for the case study requirements is shown in Listing 3.1. We can execute this Python solution in a Python interpreter environment by loading the source code file and then entering addressBook() at the interpreter prompt. This Python solution has one class named ABA_OOP_A and one function named addressBook.

Listing 3.1 ABA OOP solution—Python Version A

```
#ABA_OOP_A.py: very simple object–oriented example.
# Entry and non–persistent storage of name.

def addressBook():
  aba = ABA_OOP_A()
  aba.go()

class ABA_OOP_A:
  def go(self):
    book = []
    name = input("Enter contact name ('exit' to quit): ")
    while name != "exit":
      book.append(name)
      name = input("Enter contact name ('exit' to quit): ")

    print()
    print("TEST: Display contents of address book")
    print("TEST: Address book contains the following contacts")
    for name in book:
      print(name)
```

A Java solution for the case study requirements is shown in Listing 3.2. We can execute this Java solution using any Java development environment by loading the source code file, compiling the code, and then executing the program from within the development environment. This Java solution has two classes.

Listing 3.2 ABA OOP solution—Java Version A

```
//ABA_OOP_A_solution.java: very simple object–oriented example.
//  Entry and non–persistent storage of name.

import java.util.ArrayList;
import java.util.Iterator;
import java.util.Scanner;

public class ABA_OOP_A_solution
{
   public static void main(String[] args)
   {
      ABA_OOP_A aba = new ABA_OOP_A();
      aba.go();
   }
}

class ABA_OOP_A
{
  public void go()
  {
     ArrayList<String> book = new ArrayList<String>();
     Scanner console = new Scanner(System.in);

     System.out.print("Enter contact name ('exit' to quit): ");
     String name = console.nextLine();

     while (! name.equals("exit"))
     {
       book.add(name);
       System.out.print("Enter contact name " +
         "('exit' to quit): ");
       name = console.nextLine();
      }

      System.out.println();
      System.out.println("TEST: Display contents of address book");
      System.out.println("TEST: " +
        "Address book contains the following contacts");
      Iterator<String> iter = book.iterator();
      while (iter.hasNext())
        System.out.println(iter.next());
    }
}
```

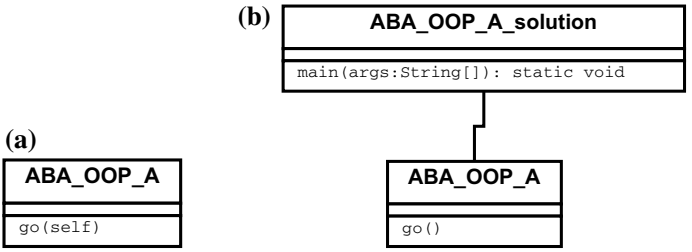

Fig. 3.1 **a** Python version A class diagram **b** Java version A class diagram

The Python solution has one function and one class. When evaluating this program design, we will ignore the addressBook function since its purpose is simply to construct an ABA_OOP_A object and then execute the go() method. The Java solution has two classes. We will evaluate this program design based on these two classes. These two OOP solutions are about as simple you can get with object-oriented programming.

3.3.1.1 Design Models

The Python and Java solutions can be described using the design modeling techniques introduced in Sects. 2.2.3 and 2.2.4. The class diagrams for these two solutions are in Fig. 3.1a, b. The Python class diagram shows the single class while the Java class diagram shows an association relationship between the class that contains the main method (which is where execution starts) and the class that contains the ABA logic.

Both the Python and Java solutions exhibit the same behavior. The Nassi–Shneiderman diagram in Fig. 3.2 describes the algorithm used in either solution. The statechart diagram in Fig. 3.3 shows the behavior of the human–computer interaction, where each state represents a distinct interaction between the user and the software application.

The Python and Java solutions are now evaluated using the program design criteria described in Sect. 2.2.2.

3.3.1.2 Separation of Concerns

All four ABA requirements (see Sect. 3.2.1) are implemented in a single Python class. Note that we ignore the AddressBook function as this is used to simply construct an object then invoke the go method in the ABA_OOP_A class. Given this, one could argue that the Python program meets the separation of concerns criteria. However, the first requirement—allow for the entry and nonpersistent storage of people's names—states two distinct concerns. One, the ABA needs to obtain people's names and two, the ABA needs to non-persistently store these names. Given this, the Python program could be better designed.

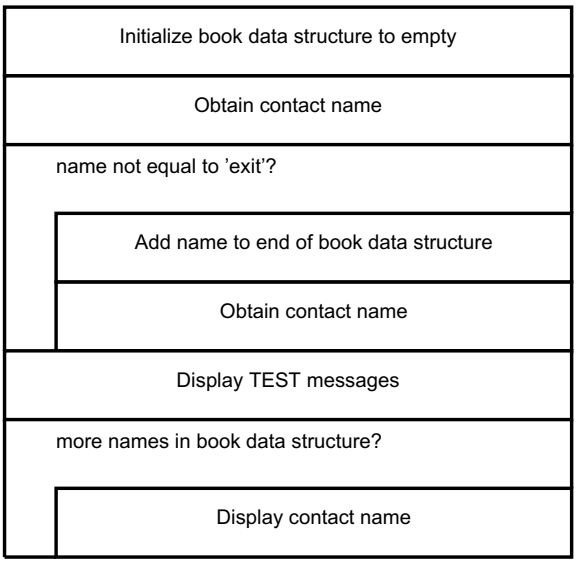

Fig. 3.2 Version A Nassi–Shneiderman diagram

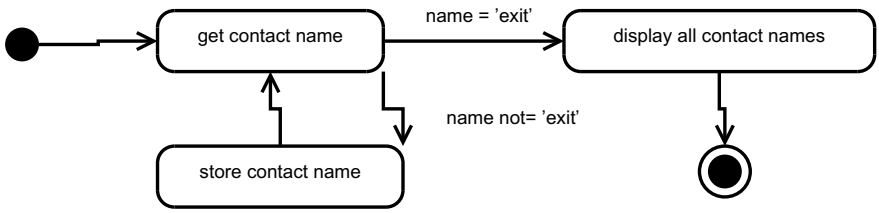

Fig. 3.3 Version A statechart

Similarly, we ignore the Java class that contains the main method, resulting in all four ABA requirements implemented in a single Java class. Like the Python solution, the Java solution could be better designed since the first ABA requirement expresses two distinct concerns.

3.3.1.3 Design for Reuse

Both the Python and Java versions contain duplicate code. In Python the *name = input(...)* statement is repeated, while in Java the two statements *System.out.print ("Enter ...");* and *String name = console.nextLine();* are repeated. Eliminating this redundant code will improve the quality of the program design.

3.3.1.4 Design Only What is Needed

Both programs include code to display the contents of the address book. This code is not necessary as it is not stated as a requirement. However, this code is included in the programs to ensure each name entered was in fact stored (not persistently) in the address book. In essence, the code displaying the address book is test code. Given this, a better design would be to put this test code in a separate function. This provides more flexibility—we could then call this test function whenever and wherever we want.

3.3.2 OOP Version B: Same Simple ABA, Better Program Design

Version B is an improved program design based on the evaluation just described for Version A. The Python code in Listing 3.3 still shows a single class, but now this class has three methods.

Listing 3.3 ABA OOP solution—Python Version B

```
#ABA_OOP_B.py: very simple object–oriented example.
# Entry and non–persistent storage of name (better).

def addressBook():
  aba = ABA_OOP_B()
  aba.go()

class ABA_OOP_B:
  def go(self):
    self.book = []
    name = self.getName()
    while name != "exit":
        self.book.append(name)
        name = self.getName()

     self.displayBook()

    def getName(self):
      return input("Enter contact name ('exit' to quit): ")

    def displayBook(self):
      print()
      print("TEST: Display contents of address book")
      print("TEST: Address book contains the following contacts")
      for name in self.book:
          print(name)
```

The Java code in Listing 3.4 still has two classes, with one class containing a very simple main method and the other class now having three methods.

Listing 3.4 ABA OOP solution—Java Version B

```
//ABA_OOP_B_solution.java: very simple object–oriented example.
//  Entry and non–persistent storage of name (better).

import java.util.ArrayList;
import java.util.Iterator;
import java.util.Scanner;

public class ABA_OOP_B_solution
{
    public static void main(String[] args)
    {
      ABA_OOP_B aba = new ABA_OOP_B();
      aba.go();
    }
}

class ABA_OOP_B
{
  ArrayList<String> book;
  Scanner console;

  public void go()
  {
     book = new ArrayList<String>();
     console = new Scanner(System.in);
     String name;
     name = getName();

     while (! name.equals("exit"))
     {
       book.add(name);
       name = getName();
     }

       displayBook();
  }

  public String getName()
  {
     System.out.print("Enter contact name ('exit' to quit): ");
     String name = console.nextLine();
     return name;
  }

  public void displayBook()
  {
    System.out.println();
    System.out.println("TEST: Display contents of address book");
    System.out.println("TEST: " +
      "Address book contains the following contacts");
```

```
        Iterator<String> iter = book.iterator();
        while (iter.hasNext())
            System.out.println(iter.next());
    }
}
```

The go() methods in each version B solution controls the overall flow of execution. However, we now have two additional methods that are called by the go() method.

3.3.2.1 Design Models

The class diagrams in Fig. 3.4a, b show the structure of the Version B solutions. The Python class diagram shows the ABA_OOP_B class having a single instance variable—self.book—representing the address book data structure. This instance variable is needed since the go method stores names in self.book and the displayBook method uses this same data structure to display all of the names stored in the data structure.

The Java class diagram shows the ABA_OOP_B class having two instance variables—book and console—representing the address book data structure and a Scanner object, respectively. The book instance variable is needed since the go method stores names in book and the displayBook method uses this same data structure to display all of the names stored in the data structure. Similarly, the console instance variable is needed since the Scanner object is constructed in the go method and then used to obtain user-supplied input data via the getName method.

The Nassi–Shneiderman and statechart diagrams for the version B solutions are the same as the version A solutions.

3.3.2.2 Separation of Concerns

The first requirement—allow for the entry and non-persistent storage of people's names—is now split between two methods. The go method is responsible for storing

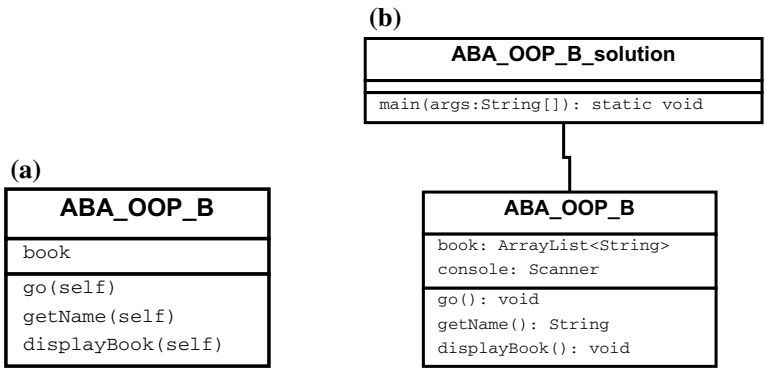

Fig. 3.4 **a** Python version B class diagram and **b** Java version B class diagram

each person's name while the getName method is responsible for obtaining a name from the user.

3.3.2.3 Design for Reuse

The duplicate code (found in version A) is now put in the getName method. Prompting for and obtaining a contact name is now done in one method; any changes to how this should be done is now localized to a single method.

3.3.2.4 Design Only What is Needed

The test code to display the contents of the address book is now all in one method. If we want to remove this test code, we can simply comment out the displayBook() call statement contained in the go method.

3.3.3 OOP Version C: No Duplicate Names

We'll now change one of the requirements (identified in italics). The ABA shall

- Allow for entry and nonpersistent storage of people's names.
- Use a simple text-based user interface to obtain the names.
- *Prevent a duplicate name from being stored in the address book.*
- Not retrieve any information from the address book.

The Python code shown in Listing 3.5 takes advantage of the *in* operator and the fact that book is a list data structure. The changed requirement is implemented in the go method via the if statement found inside the while statement.

Listing 3.5 ABA OOP solution—Python Version C

```
#ABA_OOP_C.py: very simple object–oriented example. Entry and
 non–persistent storage of name, no duplicate names (i.e., doing a
 sequential search).

def addressBook():
  aba = ABA_03c()
  aba.go()

class ABA_OOP_C:
  def go(self):
    self.book = []
    name = self.getName()
    while name != "exit":
      if name not in self.book:
        self.book.append(name)
      name = self.getName()
```

```
        self.displayBook()

    def getName(self):
      return input("Enter contact name ('exit' to quit): ")

    def displayBook(self):
      print()
      print("TEST: Display contents of address book")
      print("TEST: Address book contains the following contacts")
      for name in self.book:
          print(name)
```

The Java code shown in Listing 3.6 takes advantage of the ArrayList class and its *contains* method. The changed requirement is implemented in the go method via the if statement found inside the while statement.

Listing 3.6 ABA OOP solution—Java Version C

```
//ABA_OOP_C_solution.java: very simple object-oriented example.
//  Entry and non-persistent storage of name, no duplicate names
//  (i.e., doing a sequential search).

import java.util.ArrayList;
import java.util.Iterator;
import java.util.Scanner;

public class ABA_OOP_C_solution
{
  public static void main(String[] args)
  {
     ABA_OOP_C aba = new ABA_OOP_C();
     aba.go();
  }
}

class ABA_OOP_C
{
   ArrayList<String> book;
   Scanner console;

   public void go()
   {
     book = new ArrayList<String>();
     console = new Scanner(System.in);
     String name;
     name = getName();

     while (! name.equals("exit"))
     {
        if (! book.contains(name))
          book.add(name);
        name = getName();
```

```
      }

      displayBook();
    }

    public String getName()
    {
       System.out.print("Enter contact name ('exit' to quit): ");
       String name = console.nextLine();
       return name;
     }

     public void displayBook()
     {
       System.out.println();
       System.out.println("TEST: Display contents of address book");
       System.out.println("TEST: " +
         "Address book contains the following contacts");
       Iterator<String> iter = book.iterator();
       while (iter.hasNext())
          System.out.println(iter.next());
     }
}
```

3.3.3.1 Design Models

Neither Version C solution results in any changes to the class diagrams shown for Version B.

The two behavior diagrams—the Nassi–Shneiderman diagram in Fig. 3.5 and the statechart in Fig. 3.6—need to show the additional logic associated with not allowing duplicate names to be stored in the address book data structure.

3.3.3.2 Separation of Concerns

The go method in both solutions are now implementing two requirements. This method implements the first requirement—allow for entry and nonpersistent storage of people's names—via the use of the append (Python) or add (Java) method. The third requirement—prevent a duplicate name from being stored in the address book—is implemented in the if statement. However, since the third requirement is implemented in Python using a built-in operator and in Java using the Java ArrayList application programming interface, this program design satisfies this criteria.

3.3.3.3 Design for Reuse

There is no duplicate code in this solution. Thus, design for reuse is satisfied in this program design.

3.3.3.4 Design Only What is Needed

Besides the displayBook method used for testing purposes, this solution contains no extra processing.

3.4 OOP Post-conditions

The following should have been learned after completing this chapter.

- Program design should be evaluated using at least three criteria:
  - Separation of concerns

    Each named element of code (e.g., method) should do one, and only one, thing.
    When methods are grouped into a class, each class should have one, and only one, responsibility.

  - Design for reuse

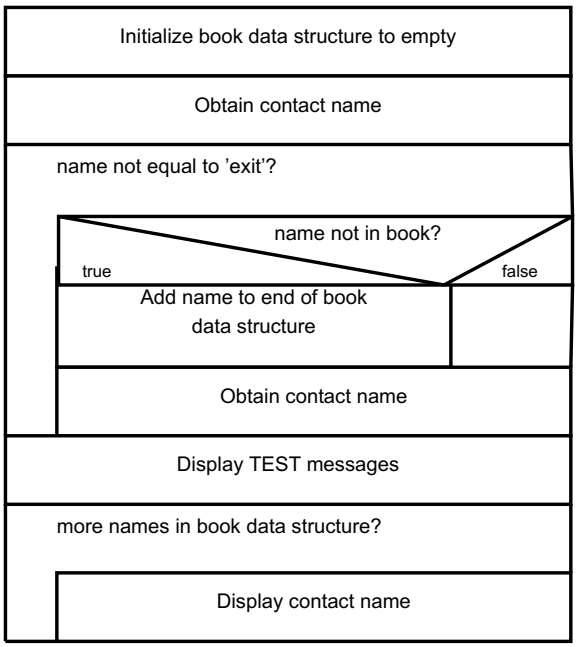

Fig. 3.5 Version C Nassi–Shneiderman diagram

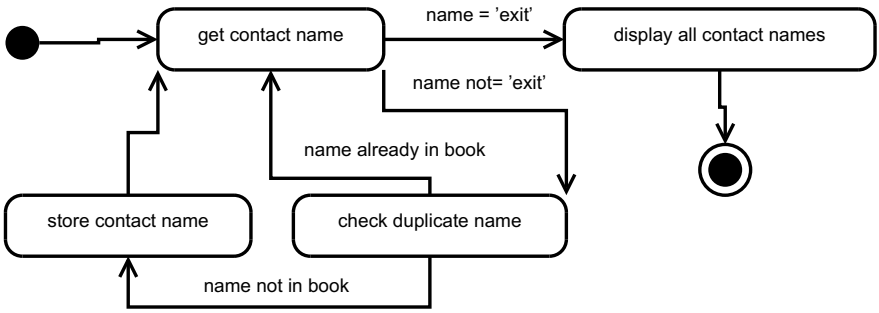

Fig. 3.6 Version C statechart

Redundant code should be eliminated. Parameters should be used to generalize a method so it may be called in situations that are similar (but not exactly the same).

– Design only what is needed

Adding code for processing that is not required will increase the time needed to test the software. This extra cost should be avoided whenever possible.

- Object-oriented program design models include the following.
  - A class diagram is used to show the structure of a solution by identifying the attributes and methods for each class in an object-oriented programming solution.
  - A Nassi–Shneiderman diagram is used to show the behavior of a solution by explaining the algorithmic processing.
  - A statechart diagram is used to show the behavior of a solution by describing the states the software is in during execution and how the solution transitions to different states during execution.

Exercises

Discussion Questions

1. What are the benefits to applying separation of concerns to a program design?
2. Can you think of a situation where you would want to ignore separation of concerns when developing a program design?

3. Identifying similar code segments that can be generalized into a single method is challenging.

   a. What types of code features should you look for when trying to identify multiple code segments that are similar enough to generalize into a method?
   b. Can you explain the steps one might take to generalize code segments into a single method?

Hands-On

1. Using an existing code solution that you've developed:

   a. Draw an appropriate model that describes the structure of your solution.
   b. Draw a Nassi–Shneiderman diagram that depicts the algorithmic processing of your solution.
   c. Draw a statechart diagram that depicts the user interactions in your solution.
   d. Improve its program design based on the criteria introduced in Sect. 2.2.2 and used in the case study in this chapter.

2. Select an application domain from the list below or select an application domain based on your interests. Write a shortlist of requirements for this application domain and then create program design models and code that meet your list of requirements. For this exercise, it is important that you keep the list of requirements short and simple. You want to focus on thinking about program design models instead of trying to figure how to implement a requirement.

   - A simple dice game like Bunco, Pig, or Poker Dice.
   - A simple card game like Concentration/Memory, Snap, or War.