OOP Case Study: Considering Security

9

The objective of this chapter is to apply the additional program design criteria—security—to the case study.

9.1 OOP Preconditions

The following should be true prior to starting this chapter.

- You understand four program design criteria: separation of concerns, design for reuse, design only what is needed, and performance. You have evaluated program code using these four 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.

9.2 OOP Simple Object-Oriented Programming Designs

The Address Book Application will be modified to improve the security of the ABA. The information security programming concepts described in Sect. 8.2.2 will be the focus of these improvements.

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_9

9.2.1 OOP Version A

We add two more requirements (identified in italics below) to specify the types of validation required for the input data. The ABA shall

- Allow for entry and (nonpersistent) storage of people's names.
- Store for each person a single phone number and a single email address.
- Use a simple text-based user interface to obtain the contact data.
- *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 limit on the size of the name entered.*
- *Ensure that each phone number contains only the digits 0 through 9. At least one digit must be entered for a phone number. There is no limit on the size of the phone number entered.*
- Prevent a duplicate name from being stored in the address book.
- Do not retrieve any information from the address book.

9.2.1.1 Python Solution

The Python code in Listing 9.1 shows how the new requirement to validate a person's name is implemented. Included in this listing are two methods: go(self) and getValidName(self). The addressBook() function and other methods of the ABA_OOP_A class are not shown. The *ABA_OOP_A.py* source code file contains the complete solution.

Listing 9.1 ABA_OOP_A.py

```
import re   #regular expression library

class ABA_OOP_A:
  def go(self):
     name = self.getValidName()
     while name != "exit":
       phone = self.getValidPhone(name)
       email = self.getTextLine("Enter email address for " +
             name + ": ")
       if name not in self.book:
         self.book[name] = (phone, email)
       name = self.getValidName()

       self.displayBook()

    #pre: Need to obtain a contact name from the user.
    #post: A valid contact name is returned.
    def getValidName(self):
      #A valid contact name may contain only spaces, uppercase
      #  letters, and lowercase letters.
      pattern = "^[ A-Za-z]+$"
      result = None
```

```
      #Continue asking for a contact name until
      #  valid data is entered.
      while result == None:
        name = self.getTextLine( \
             "Enter contact name ('exit' to quit): ")
        #Remove leading and trailing whitespace.
         name = name.strip()
         if len(name) == 0:
           #contact name must contain at least one letter.
            result = None
            errorMsg = "A contact name must contain at least \
one uppercase or lowercase letter."
         else:
           #Determine if the entered name is valid.
            result = re.match(pattern, name)
            errorMsg = "A contact name must contain only \
uppercase and lowercase letters and spaces."
      if result == None:
        print(errorMsg)

    return name
```

As seen in Listing 9.1, the go method calls getTextLine to obtain the email address, which does not have a validation requirement. To obtain the name and phone number, the go function calls getValidName and getValidPhone, respectively, to ensure that a valid name and phone number is entered by the user.

Implementing the two input validation requirements is done using the regular expression (*import re*) Python module. Looking at the getValidName method, the pattern variable is initialized to "^[A-Za-z]+$". The caret symbol "^" indicates the start of the string value while the dollar sign symbol "$" indicates the end of the string value. The symbols enclosed within square brackets "[" and "]" indicate a *set of characters* that includes space, any uppercase letter, and any lowercase letter. Specifying a set of characters in a pattern will match a single character value. The plus sign "+" after the set notation in the pattern indicates that one or more characters in the set must exist in order for the regular expression to match. The pattern and name entered by the user are passed to the *re.match* method. When this method returns None, the name value does not conform to the pattern specification. Note the use of the strip method to remove leading and trailing whitespace characters from the name entered by the user. This is done prior to checking the pattern specification. This ensures that a name of all spaces is not erroneously deemed a valid value.

The getValidPhone method (not shown in Listing 9.1) performs similar processing with two exceptions. It uses a different pattern value based on the validation requirement and the logic does not need to use the strip method since a space character is not valid in a phone number (i.e., see the validation requirement).

9.2.1.2 Java Solution

The Java code in Listing 9.2 shows how the new requirement to validate a person's name is implemented. Included in this listing are two methods: go() and getValidName(). Other methods of the ABA_OOP_A class are not shown. In addition, the ABA_OOP_A_solution and ABA_OOP_contactData classes are not shown in this code listing. The *ABA_OOP_A.java* source code file contains the complete solution.

Listing 9.2 ABA_OOP_A_solution.java

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

      while (! name.equals("exit"))
      {
        phone = getValidPhone(name);
        email = getTextLine("Enter email address for " +
              name + ": ");
        if (! book.containsKey(name))
          book.put(name, new ABA_OOP_contactData(phone, email));
        name = getValidName();
       }

      displayBook();
   }

   //pre: Need to obtain a contact name from the user.
   //post: A valid contact name is returned.
   public String getValidName()
   {
     //A valid contact name may contain only spaces,
     //  uppercase letters, and lowercase letters.
     String pattern = "[ A–Za–z]+";
     String name = "";
     String errorMsg = "";
     boolean okay = false;
     //Continue asking for a contact name until valid data
     //  is entered.
     do
     {
        name = getTextLine(
            "Enter contact name ('exit' to quit): ");
        //Remove leading and trailing whitespace.
        name = name.trim();
        if (name.length() == 0)
        {
          //contact name must contain at least one letter.
          okay = false;
          errorMsg = "A contact name must contain at least" +
```

```
            " one uppercase or lowercase letter.";
        }
        else
        {
            //Determine if the entered name is valid.
            okay = name.matches(pattern);
            errorMsg = "A contact name must contain only" +
               " uppercase and lowercase letters and spaces.";
         }
         if (! okay)
           System.out.println(errorMsg);
       } while (! okay);

       return name;
    }
}
```

As seen in Listing 9.2, the go method calls getTextLine to obtain the email address, which does not have a validation requirement. To obtain the name and phone number, the go method calls getValidName and getValidPhone, respectively, to ensure that a valid name and phone number is entered by the user.

Implementing the two input validation requirements is done using regular expressions. Looking at the getValidName method, the pattern variable is initialized to "[A-Za-z]+" which specifies a *set of characters* that includes space, any uppercase letter, and any lowercase letter. Specifying a set of characters in a pattern will match a single character value. The plus sign ("+") after the set notation in the pattern indicates that one or more characters in the set must exist in order for the regular expression to match. The matches method, which is part of the String class, is called to validate the name given a pattern that it must match. When this method returns false, the name value does not conform to the pattern specification. Note the use of the trim method to remove leading and trailing whitespace characters from the name entered by the user. This is done prior to checking the pattern specification. This ensures that a name of all spaces is not erroneously deemed a valid value.

The getValidPhone method (not shown in Listing 9.2) performs similar processing with two exceptions. It uses a different pattern value based on the validation requirement and the logic does not need to use the trim method since a space character is not valid within a phone number (i.e., see the validation requirement).

9.2.1.3 Object-Oriented Programming Design Models

The class diagrams in Fig. 9.1a, b shows the structure of the Version A solutions. Both solutions continue to use constructor methods to initialize the instance variables. The Python solution is using a tuple to store the phone number and email address for each contact person. The Java solution has a third class named ABA_OOP_contactData used to store the phone number and email address for each contact person. Given this third class, the Java class diagram uses a composition relationship (the filled-in dia-

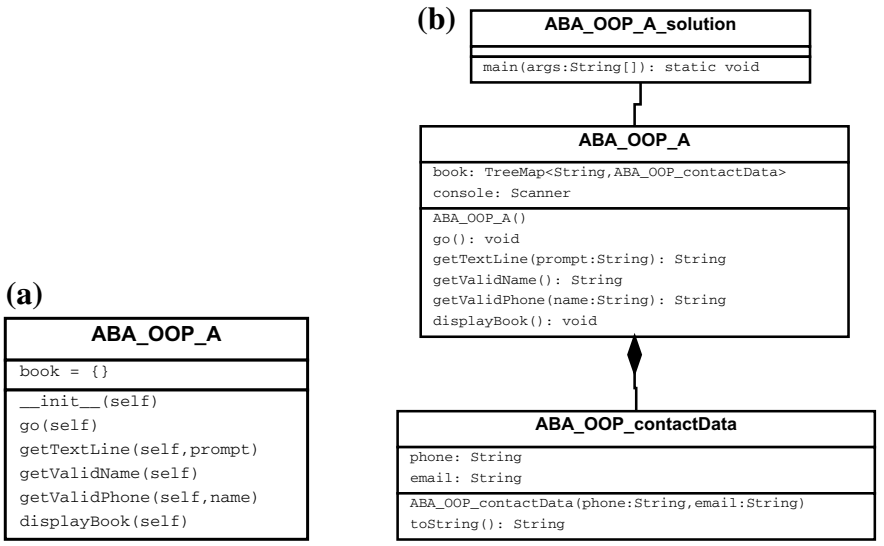

Fig. 9.1 **a** Python Version A class diagram and **b** Java Version A class diagram

mond shape) to indicate the ABA_OOP_A class contains ABA_OOP_contactData instances, where these object instances only exist inside the ABA_OOP_A object. Object-oriented class relationships will be discussed in more detail in Chap. 12.

Both the Python and Java solutions exhibit the same behavior. The Nassi–Shneiderman diagram in Fig. 9.2 describes the algorithm used in either solution while the statechart in Fig. 9.3 shows the behavior of the human–computer interaction, where each state represents a distinct interaction between the user and software application. Both of these diagrams show that the algorithmic solution now includes the validation of two input values entered by the user.

9.2.2 OOP Version B: A More Secure ABA

The version A solution validates data input by the user, an important security feature to reduce vulnerabilities through an input channel. It may be just as important to validate data output by software. This is especially true when saving data to a persistent data store or sending data to another application. Software applications that retrieve data from a persistent data store often assume that this data is valid. Similarly, receiving data from another application via a network connection is often done with the assumption that the other application will give us only valid data. To create more secure software, it is best to validate any data input into or output by an application.

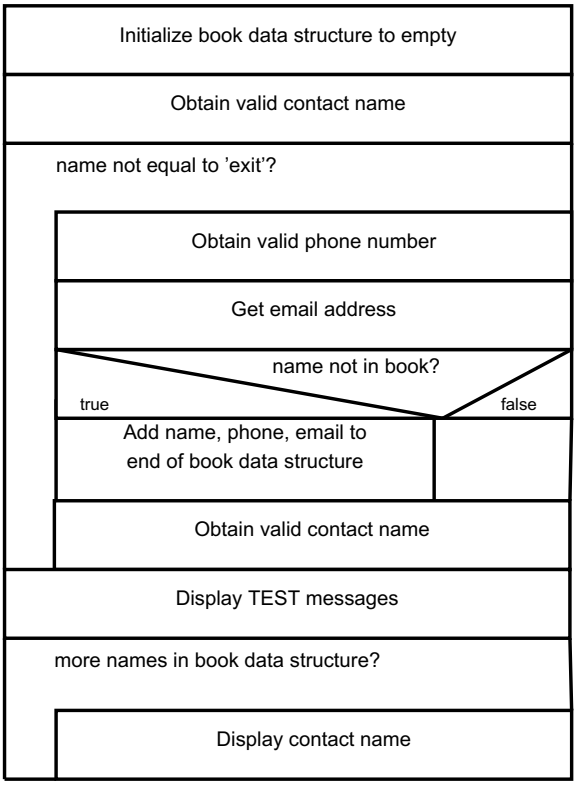

Fig. 9.2 Version A Nassi–Shneiderman diagram

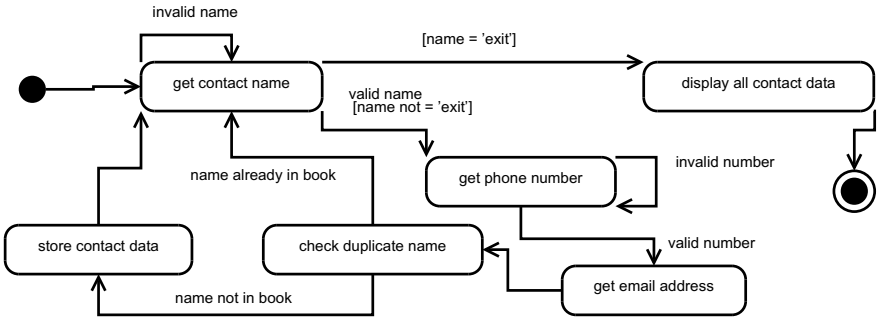

Fig. 9.3 Version A statechart

Many programming languages will raise exceptions during runtime to indicate an error condition has occurred. Handling these runtime exceptions will produce a more secure software application.

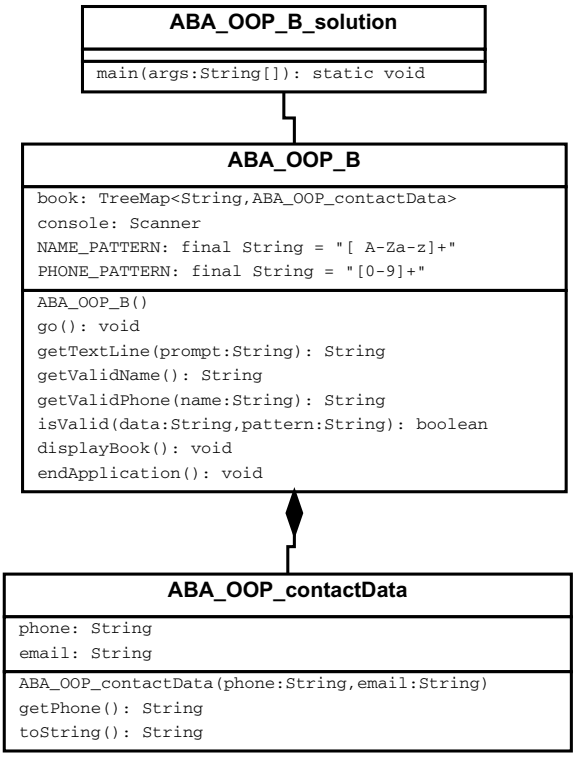

Fig. 9.4 Java Version B class diagram

Finally, recognizing that a software failure (or exception) has occurred is important to do, but this is incomplete unless the design also ensures that the failure does not lead to inappropriate access to the data.

A Java solution (Version B) is described below that includes code that implements input validation, output validation, exception handling, and fail-safe defaults.

9.2.2.1 Object-Oriented Programming Design Models

The class diagram in Fig. 9.4 shows the structure of the Version B solution. The class diagram uses the same two relationship types shown in the Version A Java solution.

The two behavior diagrams—Nassi–Shneiderman and statechart—are not shown for Version B. Instead, sample code listings are included below to explain how each security feature has been added to the solution.

9.2.2.2 Data Input Validation

No changes were made to the validation of data input by the user.

9.2.2.3 Data Output Validation

The displayBook method shown in Listing 9.3 now includes validation of the name and phone values prior to this data being displayed to the user. When either of these values is not valid, neither data value is displayed to the user even though one of the values may be valid! The NAME_PATTERN and PHONE_PATTERN identifiers are two constants defined within the ABA_OOP_B class. See the ABA_OOP_B.java source code file for details on these constants and other portions of the version B solution not described in this section.

Listing 9.3 ABA_OOP_B displayBook method

```
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");

  NavigableSet<String> keySet = book.navigableKeySet();
  Collection<ABA_OOP_contactData> valueColl = book.values();

  Iterator<String> iterKey = keySet.iterator();
  Iterator<ABA_OOP_contactData> iterValue = valueColl.iterator();

  String name, phone, data;
  while (iterKey.hasNext() && iterValue.hasNext())
  {
     try
     {
       name = iterKey.next();
       ABA_OOP_contactData contactData = iterValue.next();
       data = contactData.toString();
       phone = contactData.getPhone();
       if (! isValid(name, NAME_PATTERN) ||
         ! isValid(phone, PHONE_PATTERN))
       {
          name = "[InvalidName]";
          data = "[InvalidContactData]";
       }
     }
     catch (Exception ex)
     {
       name = "[InvalidName]";
       data = "[InvalidContactData]";
     }
     System.out.println(name + " " + data);
  }
}
```

9.2.2.4 Exception Handling

The displayBook method shown in Listing 9.3 includes a try-catch block that will capture exceptions. The expressions *iterKey.next()* and *iterValue.next()* will cause a NoSuchElementExcetion when the iterator object has no more elements. Under normal operation, it would be considered a logic error if this exception was to occur. However, it may be that the memory that contains the book TreeMap has been corrupted in some way, perhaps maliciously. Additionally, any exception that is raised as a result of executing the other statements inside the try block will be caught by the catch block of code.

The getTextLine method shown in Listing 9.4 now has a try-catch block to capture exceptions caused by the Scanner method nextLine. Specifically, holding the Ctrl key down while striking the C key will cause a NoSuchElementException. When this or any other exception is thrown by the nextLine method, an error message is displayed and the program is terminated.

Listing 9.4 ABA_OOP_B getTextLine function

```
//pre: prompt contains a message (typically instructions) to be
//   displayed to user.
//post: returns value entered by user as a String.
public String getTextLine(String prompt)
{
  System.out.print(prompt);
  String textLine;
  try
  {
    textLine = console.nextLine();
  }
  catch (Exception ex)
  {
    textLine = "";
    //Entering Ctrl–Z causes a NoSuchElementException
    System.out.println("\nConsole input has been terminated," +
        " ending application.\n");
    endApplication();
  }
  return textLine;
}
```

The getValidName method shown in Listing 9.5 no longer includes the logic to validate the contact name. Instead, the isValid method (also shown in Listing 9.5) has been added for the purpose of validating a string data value using a regular expression pattern. The try-catch block in the isValid method will detect the use of an invalid regular expression.

Listing 9.5 ABA_OOP_B getValidName and isValid methods

```
//pre: Need to obtain a contact name from the user.
//post: A valid contact name is returned.
public String getValidName()
{
```

```
  String name = "";
  boolean okay = false;
  //Continue asking for contact name until valid data is entered.
  do
  {
     name = getTextLine("Enter contact name ('exit' to quit): ");
     //Remove leading and trailing whitespace.
     name = name.trim();
     okay = isValid(name, NAME_PATTERN);
     if (! okay)
       System.out.println("A contact name must contain only" +
         " uppercase and lowercase letters and spaces.");
  }  while (! okay);

  return name;
}

//pre: User has entered a data value that can be validated
//   using a regular expression.
//post: Return true when name is valid. Otherwise, return false.
public boolean isValid(String data, String pattern)
{
   boolean okay = false;
   if (data.length() > 0)
   {
      try
      {
        //Determine if the entered name is valid.
        okay = data.matches(pattern);
      }
      catch (PatternSyntaxException ex)
      {
         System.out.println("\nLogic error in pattern matching," +
           " ending application.\n");
         endApplication();
      }
   }

   return okay;
}
```

Similarly, the getValidPhone function no longer includes the logic to validate the phone number. Instead, the isValid function is called for the purpose of validating a phone number.

9.2.2.5 Fail-Safe Defaults

The displayBook method shown in Listing 9.3 is an example of fail-safe defaults. When a software failure occurs, in this case when either the name or phone value

stored in the address book data structure is not valid, a generic "invalid data" value is displayed instead of the data currently stored in the book dictionary.

9.3 OOP Post-conditions

The following should have been learned when completing this chapter.

- Developing a safe (more secure) software application involves consideration of a number of factors, including
  - Validate data input into an application.
  - Validate data output by an application.
  - Include an exception handler in code to react in a safe (more secure) way when an exception may occur. In Python, use try-except blocks to react to runtime exceptions. In Java, use try-catch blocks to react to runtime exceptions.
  - Use fail-safe defaults when an error condition or exception is detected, to ensure data is not used in an inappropriate (less secure) manner.
- Making your code more secure often involves adding some complexity to your solution. Testing your code needs to include specific test cases to determine whether your code responds correctly to both proper and improper use of your application.

Exercises

Hands-on Exercises

1. The Java Version B solution has two functions, getValidName and getValidPhone, that appear to contain very similar code. Develop a solution that combines these two functions into one generic function that is able to validate a name or a phone value. Evaluate your solution using the program design criteria.
2. Use an existing code solution you have developed and identify portions of the program design that could be changed to improve its security. Implement these security changes in your solution.