6 OOP Case Study: Considering Performance

The objective of this chapter is to apply all four program design criteria—separation of concerns, design for reuse, design only what is needed, and performance—to the case study.

6.1 OOP Pre-conditions

The following should be true prior to starting this chapter.

- Evaluating the performance of a program design should consider both time and memory usage.
- You understand three program design criteria: separation of concerns, design for reuse, and design only what is needed. You've evaluated program code using these criteria.
- You know that a class diagram may be used to model the structure of your object-oriented program design.
- You know that a Nassi–Shneiderman diagram and a statechart are models used to illustrate the behavior of your program design.

6.2 OOP Simple Designs

We'll start with the ABA requirements as stated for Version C in Chap. 3. The ABA shall

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_6

- 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.

6.2.1 OOP Version A

The Version A solutions shown in Listings 6.1 and 6.2 are a result of changing the data structure being used to store contact names. These changes were made to improve the responsiveness of the ABA based on the time performance improvements described in Chap. 5.

Listing 6.1 ABA_OOP_A.py

```
#ABA_OOP_A.py: very simple object-oriented example.
# Entry and non-persistent storage of name,
# no duplicate names (using dictionary).

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

class ABA_OOP_A:
  def go(self):
    self.book = {}
    name = self.getName()
    while name != "exit":
      if name not in self.book:
        self.book[name] = None
      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 *if name not in self.book* conditional statement in Listing 6.1 results in Python using the built-in hash function __hash__() for the string (str) data type [1]. In a worst-case scenario, the hash function would generate the same hash table location for each contact name. The result would be a sequential search based on the use of the quadratic open addressing collision resolution algorithm. As noted in Chap. 5,

this worst case would result in linear $O(n)$ time performance which matches the performance of the solutions in Chap. 3.

The Java code for Version A is shown in Listing 6.2. This Java program design is significantly different since it is using a TreeSet instead of an ArrayList. As described in Chap. 5, this results in logarithmic time $O(\log_2 n)$ performance when determining if the contact name already exists in the ABA. Specifically, the add method of the TreeSet Java API class will not modify the TreeSet when the name is already in the data structure.

Listing 6.2 ABA_OOP_A.java

```
//ABA_OOP_A_solution.java: very simple object-oriented example.
// Entry and non-persistent storage of name,
// no duplicate names (using TreeSet).

import java.util.Iterator;
import java.util.Scanner;
import java.util.TreeSet;

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
{
  TreeSet<String> book;
  Scanner console;

  public void go()
  {
    book = new TreeSet<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): ");
```

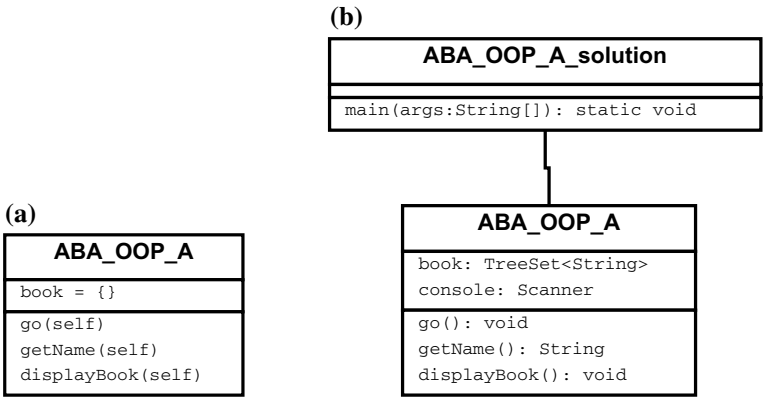

Fig. 6.1 **a** Python version A class diagram. **b** Java version A class diagram

```
        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());
    }
}
```

6.2.1.1 Design Models

The class diagrams for these two solutions are in Fig. 6.1a, b. The Python class diagram lists the same attributes and methods as shown in the Version B and C solutions in Sects. 3.3.2 and 3.3.3, respectively. The only difference is that the book attribute is initialized to an empty list in Chap. 3 while in this chapter it is a dictionary. Likewise, the Java class diagram lists the same attributes and methods as shown in the Version B and C solutions in Chap. 3. The only difference is that the book attribute is an ArrayList in Chap. 3 while in this chapter it is a TreeSet.

Both the Python and Java solutions exhibit the same behavior. The Nassi–Shneiderman diagram in Fig. 6.2 describes the algorithm used in either solution. The statechart diagram in Fig. 6.3 shows the behavior of the human–computer interaction, where each state represents a distinct interaction between the user and the software application.

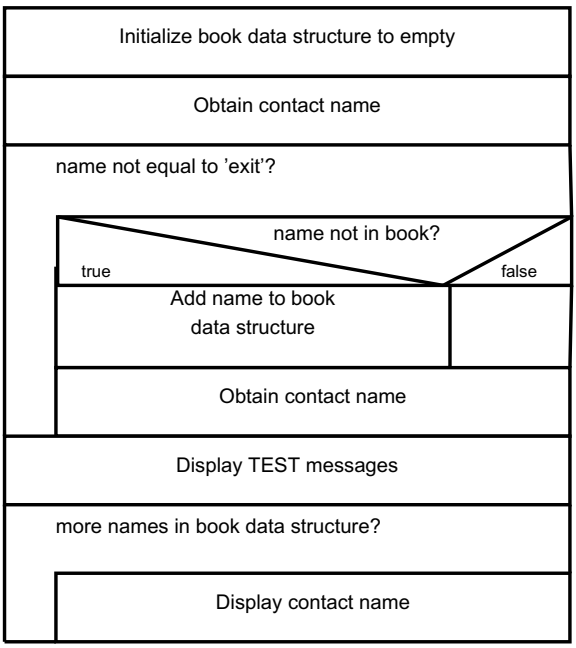

Fig. 6.2 Version A Nassi–Shneiderman diagram

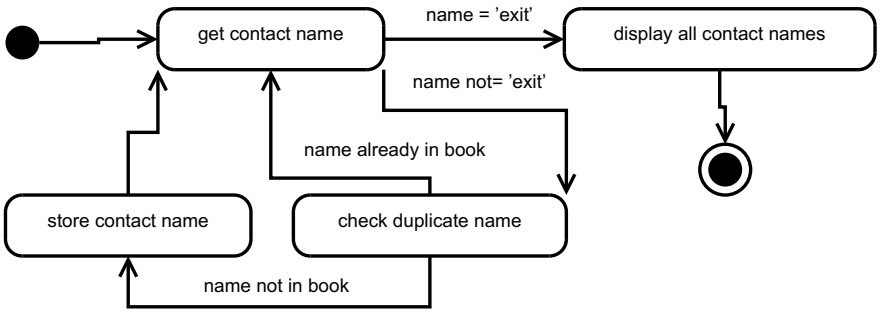

Fig. 6.3 Version A statechart

Note the Nassi–Shneiderman diagram shows a selection statement *name not in book?* inside the while loop. In the Python solution, we see this if statement in the code shown in Fig. 6.1. For the Java solution, the add method of the TreeSet class will not add the element when the element is already in the data structure. In effect, this add method contains a selection statement that prevents a duplicate name from being stored in the data structure.

The Python and Java solutions are now evaluated using the program design criteria described in Chaps. 2 and 5.

6.2.1.2 Program Design Criteria (from Chap. 2)

In terms of separation of concerns, design for reuse, and design only what is needed, changing the type of data structure used to store the address book data has no impact on these program design criteria. This solution exhibits the same program design characteristics as the Version C solution in Chap. 3.

6.2.1.3 Time Performance

Changing the type of data structure used to store the address book data will have a positive impact on the responsiveness of the ABA. To summarize, the time performance of checking to determine if the contact name is a duplicate was studied. Using a Python dictionary instead of a Python list produced a best case of constant time instead of linear time. Using a Java TreeSet instead of a Java ArrayList produced a best case of logarithmic time instead of linear time.

6.2.1.4 Memory Performance

As mentioned in Chap. 5, use of a Python dictionary means that a hash table is being used. Given the nonpersistent storage of contacts at this point in the ABA, it is quite likely that the hash table will contain a small number of entries that will not result in allocating a lot of memory that does not get used.

For the Java solution, the TreeSet data structure will contain a node only for those contacts that are currently in the ABA. Thus, the Java solution will only allocate memory in the TreeSet when it is needed to store contact data.

6.2.2 OOP Version B

Our address book is not very useful since it only stores contact names. Let's add one more requirement, identified in italics below. The ABA shall

- 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 contact data.
- Prevent a duplicate name from being stored in the address book.
- Not retrieve any information from the address book.

The Python solution is shown in Listing 6.3. This solution also uses a dictionary, and stores a phone number value instead of None as the mapped data value. Specifically, the statement *book[name] = None* in Version A has been replaced with the statement *book[name] = phone*. The displayBook method has also been modified to display the address book names in alphabetical order. This was done so the Python and Java programs for Version B exhibit the same behavior.

Listing 6.3 ABA_OOP_B.py

```
#ABA_OOP_B.py: very simple object-oriented example.
# Entry and non-persistent storage of name,
# no duplicate names, a phone number.

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

class ABA_OOP_B:
  def go(self):
    self.book = {}
    name = self.getName()
    while name != "exit":
      phone = self.getPhone(name)
      if name not in self.book:
        self.book[name] = phone
      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")
      sortedNames = sorted(self.book.keys())
      for name in sortedNames:
          print(name, self.book[name])

    def getPhone(self, name):
      phone = input("Enter phone number for " + name + ": ")
      return phone
```

The Java solution in Listing 6.4 uses a map instead of a set, since a map stores a key–value pair whereas a set only stores keys. Specifically, the data type *TreeMap<string, string>* indicates the map will store a key that is a string (i.e., contact name) along with a value related to the key (i.e., phone number) which is also a string.

Listing 6.4 ABA_OOP_B.java

```
//ABA_OOP_B_solution.java: very simple object-oriented example.
// Entry and non-persistent storage of name,
// no duplicate names, a phone number.

import java.util.Collection;
import java.util.Iterator;
import java.util.NavigableSet;
import java.util.Scanner;
```

```
import java.util.TreeMap;

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
{
  TreeMap<String,String> book;
  Scanner console;

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

    while (!name.equals("exit"))
    {
       phone = getPhone(name);
       if (!book.containsKey(name))
         book.put(name, phone);
       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");

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

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

```
        while (iterKey.hasNext() && iterValue.hasNext())
          System.out.println(iterKey.next() + " " + iterValue.next());
    }

    public String getPhone(String name)
    {
      System.out.print("Enter phone number for " + name + ": ");
      String phone = console.nextLine();
      return phone;
    }
}
```

The program designs have been modified by adding another method. Below are updated design models and a look at the four program design criteria.

6.2.2.1 Design Models

The getPhone method has been added to each solution. Figure. 6.4a, b shows this change in the class diagrams.

The Nassi–Schneiderman chart and statechart diagrams would need to be changed to show the entry of a phone number by the user. The updates to these models are not shown.

Version B solutions are now evaluated using the four program design criteria.

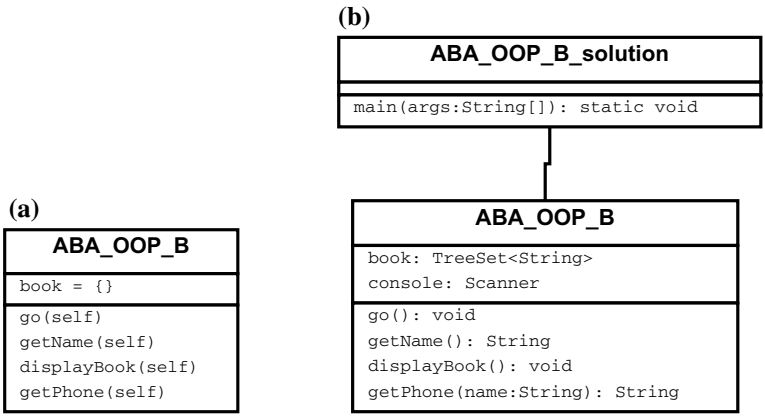

Fig. 6.4 **a** Python Version B class diagram. **b** Java Version B class diagram

6.2.2.2 Separation of Concerns

The additional requirement results in the getPhone method being added to the solution. This new method is called by the go method. This updated program design adheres to the principle of separation of concerns.

6.2.2.3 Design for Reuse

The getPhone method is conceptually similar to the getName method. This is a good example of the more challenging aspect of design for reuse—identifying code segments (or complete methods) that are similar but not the same. To help identify and generalize two methods into a single method, you can write a description of the logic as a series of algorithmic steps. In this case, both methods: display a prompt to the user; obtain data entered by the user; and return this data to the calling function. Given the similar processing performed by the two methods, there is likely a better program design that would eliminate the need to have both of these methods.

6.2.2.4 Design Only What is Needed

The additional method is necessary to fully implement the additional requirement.

6.2.2.5 Time Performance

For the Python solution, there is no change in performance for Version B when compared to Version A since both use a dictionary. The Java program uses a TreeMap instead of a TreeSet. Since maps are typically implemented the same way that sets are (i.e., using a red-black binary search tree algorithm), there is no change in time performance for Version B when compared to Version A.

6.2.2.6 Memory Performance

There is no change in memory performance for these Version B solutions when compared to the Version A solutions.

6.2.3 OOP Version C

We'll add one more requirement to our Address Book Application, identified in italics below, as the last ABA version presented in this chapter. 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.
- Prevent a duplicate name from being stored in the address book.
- Not retrieve any information from the address book.

The program designs in Listings 6.5 and 6.6 address the problem noted above in the design for reuse criteria for Version B. Instead of having getName and getPhone methods (and a getEmail method for the new requirement), we now have one method getTextLine. The getTextLine method has one parameter variable so the correct prompt value can be passed to it for display to the user.

Listing 6.5 ABA_OOP_C.py

```
#ABA_OOP_C.py: very simple object-oriented example.
# Entry and non-persistent storage of name,
# no duplicate names, a phone number and email(better).

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

class ABA_OOP_C:
  def __init__(self):
    self.book = {}

  def go(self):
   name = self.getTextLine(
       "Enter contact name ('exit' to quit): ")
   while name != "exit":
    phone = self.getTextLine(
        "Enter phone number for " + name + ": ")
    email = self.getTextLine(
        "Enter email address for " + name + ": ")
    if name not in self.book:
      self.book[name] = (phone, email)
    name = self.getTextLine(
        "Enter contact name ('exit' to quit): ")

   self.displayBook()

  def getTextLine(self, prompt):
    return input(prompt)

  def displayBook(self):
    print()
    print("TEST: Display contents of address book")
    print("TEST: Address book contains the following contacts")
    sortedNames = sorted(self.book.keys())
    for name in sortedNames:
      print(name, self.book[name])
```

Listing 6.6 shows the Java solution.

Listing 6.6 ABA_OOP_C.java

```
//ABA_OOP_C_solution.java: very simple object-oriented example.
// Entry and non-persistent storage of name,
// no duplicate names, a phone number and email (better).
```

```
import java.util.Collection;
import java.util.Iterator;
import java.util.NavigableSet;
import java.util.Scanner;
import java.util.TreeMap;

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_contactData
{
    String phone;
    String email;

  public ABA_OOP_contactData(String phone, String email)
  {
     this.phone = phone;
     this.email = email;
  }

  public String toString()
  {
    return "(" + phone + ", " + email + ")";
  }
}

class ABA_OOP_C
{
  TreeMap<String,ABA_OOP_contactData> book;
  Scanner console;

  public ABA_OOP_C()
  {
     book = new TreeMap<String,ABA_OOP_contactData>();
     console = new Scanner(System.in);
  }

  public void go()
  {
     String name, phone, email;
     name = getTextLine("Enter contact name ('exit' to quit): ");

     while (! name.equals("exit"))
     {
       phone = getTextLine("Enter phone number for " +
             name + ": ");
       email = getTextLine("Enter email address for " +
              name + ": ");
       if (! book.containsKey(name))
         book.put(name, new ABA_OOP_contactData(phone, email));
       name = getTextLine("Enter contact name ('exit' to quit): ");
     }
```

```
      displayBook();
    }

    //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 = console.nextLine();
      return textLine;
    }

    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();

       while (iterKey.hasNext() && iterValue.hasNext())
         System.out.println(iterKey.next() + " " + iterValue.next());
    }
}
```

6.2.3.1 Design Models

The class diagrams in Fig. 6.5a, b show the structural changes to the Version C solution. The Python solution uses a tuple to store a phone number and email address for a contact name. In contrast, the Java solution now has a third class used to store a phone number and email address for a contact name. Both solutions now use constructor methods to initialize the instance variables. Python constructor methods are always named __init__ while Java constructor methods have the same name as the class and do not have a method return type.

The Nassi–Schneiderman chart and statechart diagrams would need to be changed to show entry of an email address by the user. The updates to these models are not shown.

Version C solutions are now evaluated using the four program design criteria.

6.2.3.2 Separation of Concerns

The Python solution uses a tuple to store multiple values (e.g., phone number and email address) for a contact name. This results in no impact on the separation of concerns criteria.

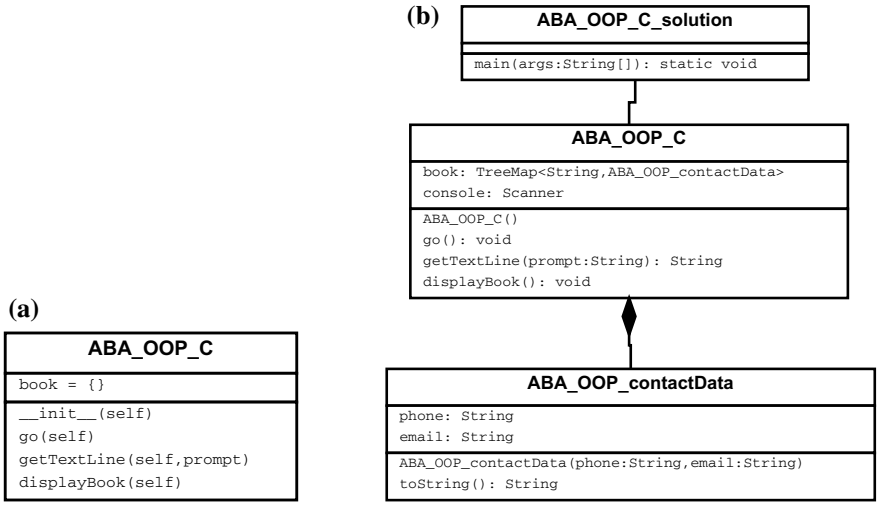

Fig. 6.5 **a** Python Version C class diagram. **b** Java Version C class diagram

The Java solution now has a third class named ABA_OOP_contactData used to store a phone number and email address for each contact name in the TreeMap. This third class is necessary since the TreeMap requires a data type for the key value and the associated data value. This updated program design adheres to the principle of separation of concerns.

6.2.3.3 Design for Reuse

The getTextLine method can now be used whenever there is a need to display a prompt to the user, obtain data entered by the user, and return this data to the calling method as a string value. We have generalized the getName and getPhone methods into a single method, resulting in the getTextLine method being more useful to us. This is evident by the fact that the additional requirement of obtaining an email address was implemented without needing to define a new method.

6.2.3.4 Design Only What is Needed

The Python and Java program designs are performing only those processing steps that are required.

6.2.3.5 Time Performance

There is no change in performance for Version C when compared to Version B since the Python and Java solutions continue to use the same type of data structure.

6.2.3.6 Memory Performance

There is no change in memory performance for these Version C solutions when compared to the Version B solutions.

6.3 OOP Post-conditions

The following should have been learned after completing this chapter.

- Evaluating a program design should consider at least five criteria: separation of concerns, design for reuse, design only what is needed, time performance, and memory performance.
- When adding new methods to a solution, a software engineer should determine whether the new code is similar to an existing method. When similar code segments are found, a more general method should be developed that serves the needs of both code segments.

Exercises

Hands-on Exercises

1. Modify the Version A Nassi–Shneiderman diagram so that it accurately describes the behavior of Version C.
2. Modify the Version A statechart so that it accurately describes the behavior of Version C.
3. Use an existing code solution that you've developed and identify portions of the program design that could be changed to improve its time performance.
4. Use an existing code solution that you've developed and identify portions of the program design that could be changed to improve its memory performance.
5. Select different types of data structures supported by a programming language that you use and identify the types of operations that would result in poor performance. For example, for a given data structure what would the time performance be for accessing a single value via a search? For accessing the first, middle, or last value in the data structure?

Reference

1. Python Software Foundation: The python standard library, version 3.3.5. https://docs.python.org/3.3/library/functions.html. Accessed 4 Jun 2014