Concepts of Object-Oriented Programming
AS 2020

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 12

Encapsulation and Aliasing

December 11, 2020

Task 1

Suppose that the following Java classes are part of a package, to which an external user cannot add classes.

```
public abstract class BankAccount {
   ... boolean importantCustomer = false;
   ... int amount = 0;
   ... final int maxDebit = 1000;

   /// invariant amount >= -maxDebit &&
   ///   !importantCustomer => amount >= 0 &&
   ///   importantCustomer <=> this instanceof RichCustomer

   ... void deposit(int amount);
   ... void withdraw(int amount);
}

public final class PoorCustomer extends BankAccount {
   ... void deposit(int amount) {
      if(amount >= 0)
         this.amount += amount;
   }
   ... void withdraw(int amount) {
      if(amount <= this.amount)
         this.amount -= amount;
   }
}

public final class RichCustomer extends BankAccount {
   public RichCustomer() { importantCustomer = true; }
   ... void deposit(int amount) {
      if(this.amount + amount >= -maxDebit)
         this.amount += amount;
   }
   ... void withdraw(int amount) {
         if(-maxDebit <= this.amount - amount)
            this.amount -= amount;
   }
}
```

Provide the most permissive access modifiers for each field and method, such that the class invariant cannot be broken from outside the package. Assume that no integer over/underflow occurs.

In Scala, a class can be declared as sealed. That means that the class can be extended only by classes written in the same `.scala` file. Suppose that the class `BankAccount` is declared as sealed, and `PoorCustomer` and `RichCustomer` are part of the same `.scala` file. Does this allow you to choose more permissive access modifiers? Note that `PoorCustomer` and `RichCustomer` are still declared as final.

Task 2

Consider the following Java code:

```
package p;

public final class List {
   ///invariant 1: The list starting at head is acyclic
   ///invariant 2: The list starting at head is non-decreasing

   public void prepend(int x){
      if (head == null || x <= head.getValue())
         head = new Node(x, head);
   }

   public Node getHead() { return head; }
   public Node head = null;
}

public final class Node {
   Node(int x, Node n) {
      value = x;
      next = n;
   }

   public Node getNext() { return next; }
   public int getValue() { return value; }
   private Node next;
   private int value;
}
```

Assuming that we cannot modify the classes `List` and `Node`, we would like to see whether or not the invariants can be broken, either by adding classes to package `p`, or by clients outside of package `p`. Assume reflection is not used at all.

A) Can invariant 1 be broken by adding clients outside of package `p`? If yes, show code, that when run ends in a state in which the invariant is broken; if not explain why.

B) Can invariant 1 be broken by adding classes to package `p`? If yes, show code, that when run ends in a state in which the invariant is broken; if not explain why.

C) Can invariant 2 be broken by adding clients outside of package `p`? If yes, show code, that when run ends in a state in which the invariant is broken; if not explain why.

D) Can invariant 2 be broken by adding classes to package `p`? If yes, show code, that when run ends in a state in which the invariant is broken; if not explain why.

Task 3

Consider the following Java code:

```
public class Hour {
   public int h = 0;
}
```

```
public class Time {
   private Hour hour = new Hour();
   /// invariant hour.h >= 0 && hour.h < 24

   public void setHour(int h) {
      if (h >= 0 && h < 24) this.hour.h = h;
   }

   public Hour getHour() { return hour; }
}
```

A) Provide an example that breaks the invariant of `Time` without changing the code above and without using reflection.

B) There are two immediate ways to fix the problem. In one of them, signatures of methods are modified, while in the other they are not. What are these ways of fixing the problem?

C) Clearly, we would prefer to keep the signatures the same as before. Are there any drawbacks to this approach?

D) Would it be possible to introduce an interface with no mutator methods and use it to solve the problem? Explain how this approach would look and whether there would still be a way to break the invariant.

Task 4

Data structures often intentionally share aliases. For instance, consider the following Java class:

```
class ArrayList<T> {
   private T[] elements = ...;
   private int lastEl = 0;
   public T get(int i) { return elements[i]; }
   public int size() { return lastEl; }
   public void add(T el) { elements[lastEl++] = el; }
}
```

Imagine that this class is extended as follows

```
class Coordinates {
   int x, y;
   public Coordinates(int xx, int yy) { x = xx; y = yy; }
}

class CList extends ArrayList<Coordinates> {
   /// invariant ∀ i:int | 0 ≤ i ∧ i < size() ⇒ get(i).x > get(i).y
   public void add(Coordinates el) {
      if (el.x > el.y) super.add(el);
   }
}
```

A) Write a program that breaks the invariant of `CList`.

B) How can we fix this problem?

C) Is it possible to fix it without allocating new objects (either directly or indirectly), that is, without consuming additional memory? What new problems might arise from your changes?

D) Discuss the benefits and the drawbacks of using alias sharing in data structures.

Task 5

The following Java classes, all part of the `security` package, were written by an unexperienced programmer and contain a number of issues:

```
package security;

public class User {
   public String name;
   public String password;
   public User(String name, String password) {
      this.name = name;
      this.password = password;
   }
}

public class LoginException extends RuntimeException {
   public User problemUser;
   public LoginException(String message, User problemUser) {
      super(message);
      this.problemUser = problemUser;
    }
}

public class Login {
   private List<User> users = new LinkedList<User>();
   public void registerUser(User u) {
      if (u == null || u.name == null || u.password == null
         || u.name.isEmpty() || u.password.isEmpty()) return;
      users.add(u);
   }

   // Returns true if the user 'u' was successfully logged in.
   // Otherwise returns false or throws an exception.
   public boolean login(User u) throws LoginException {
      if (u == null) return false;
      User current = null;
      try {
         for(User registered : users) {
            boolean nameEqual = registered.name.equals(u.name);
            current = registered;

            if (nameEqual) {
               if (registered.password.equals(u.password))
                  return true;
            }

            if (nameEqual)
               throw new LoginException("Invalid password for user", u);
         }

         return false;
      }
      catch(Exception e) {
         throw new LoginException("Invalid user", current);
      }
   }
}
```

The `malicious` method is in a different package:

```
void malicious(Login l) { ... }
```

Assume the `Login` object that is passed into the method already has registered users.

A) Complete the body of the `malicious` method so that you manage to log-in as an already existing user. You do not know any names or passwords of existing users. Do not use reflection. You are not allowed to call `login` more than a constant number of times.

B) Is it possible to fix the problem under the following restrictions? In each of these cases, explain how you can prevent the malicious login or why it is not possible.

- only modifying the `User` class?
- only modifying the `LoginException` class?
- only modifying the `registerUser` method?
- only modifying the body of the `for` loop inside the `login` method?

Task 6 (from a previous exam)

In answering this task, do not use reflection, inheritance, and static fields or methods.

This task is concerned with reasoning about *non-modification* in a modular setting in the presence of aliasing.

Consider the following code:

```
package cell;
public class Cell {
   /// ensures get() == newValue
   public Cell(int newValue) { value = newValue; }

   /// ensures get() == newValue
   public void set(int newValue) { value = newValue; }

   /// pure
   public int get() { return value; }
   private int value;
}

package client;
import cell.*;
class Client{
   /// requires c1 != null
   /// requires c2 != null
   void setCells(Cell c1, Cell c2) {
      c1.set(1);
      c2.set(2);
      assert(c1.get() == 1);
   }

   void setCellsClient() {
      Cell c1 = new Cell(5);
      Cell c2 = new Cell(5);
      setCells(c1, c2);
   }
}
```

The objective of this task is to make sure that the assertion in the method `setCells` does not fail, using modular reasoning. The potential problem is that of determining whether the call `c2.set(2)` can affect the return value of `c1.get()`.

A) Modify the second line in method `setCellsClient` (the initialization of `c2`) so that the assertion in method `setCells` fails. The precondition of `setCells` must still be satisfied by the modified version.

B) Add a precondition to `setCells` that will make the call from your version of the method `setCellsClient` illegal. The precondition should be such that the original call is legal. Remember that the precondition can only refer to the arguments of the method and to public fields and methods.

C) We now add a `clone` method to the `Cell` class:

```
/// ensures result != null
/// ensures result != this
/// ensures result.get() == get()
/// ensures get() == old(get())
public Cell clone() { return new Cell(value); }
```

We also add to the client the methods `left` and `right`, which use the `clone` method:

```
void left() {
   Cell c1 = new Cell(5);
   Cell c2 = c1.clone();
   setCells(c1, c2);
}
```

```
void right() {
   Cell c1 = new Cell(5);
   Cell c2a = new Cell(5);
   Cell c2 = c2a.clone();
   setCells(c1, c2);
}
```

Modify only the `Cell` class so that a call to `left` causes the assertion in `setCells` to fail, while a call to `right` does not cause the assertion to fail. You can add private and default access members and methods to the `Cell` class and add private classes to the `cell` package, and also modify the implementation of existing methods, but not change the public interface in any way. Your implementations must satisfy the existing contracts, including the one from task B.

D) Strengthen the precondition of the method `setCells` so that, with your modified `Cell`, the call from `left` would fail the precondition check, while the call from the method `right` would satisfy the precondition.

You can use the concept of the reach of an object, where, for an object `x`, `reach(x)` is defined as the the set of objects which are reachable from `x` — the set of objects which can be described by an access path `x.f1.f2. ... .fn` for some n and some sequence of field names `f1..fn` (we do not consider arrays in this task). All fields are considered, regardless whether they are public or private. You can also use set operations in your precondition.

Remember that the precondition of a method can refer only to the `this` object and the method's arguments, dereferencing of public fields, and call public pure methods.

E) In order to prove the correctness of the body of the methods `left` and `right`, when `setCells` has the stronger precondition from section D, we would have to strengthen the postcondition of the `clone` method of class `Cell`. Write a stronger postcondition to the method `Cell.clone` so that the bodies of the methods `left` and `right` can be proven modularly — i.e., without knowing the implementation of the `clone` method and other private details of the class `Cell`.