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.

```java
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.

solution

For the fields of class `BankAccount`, the most permissive access modifiers are:

`importantCustomer`: default modifier. In this way, it would be accessible by other classes in the same package but not by subclasses. Otherwise, we may have a class that extends `BankAccount` and sets to true `importantCustomer` without being a `RichCustomer`.

`amount`: default, since we need to access it from the other classes of this package (e.g. `PoorCustomer` and `RichCustomer`), but we must prevent external attackers from modifying it.

`maxDebit`: public, since it is final and it cannot be modified by other classes.

Methods `withdraw` and `deposit` can be declared public, since they preserve the invariants.

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.

solution

If class `BankAccount` had been declared as sealed, we could choose protected as the access modifier of the `amount` and `importantCustomer` fields, since external classes would not be allowed to extend it and so would not be able to gain access to these fields. More generally, if a class is sealed, the default and protected levels are equivalent, since it is not possible to extend the current class outside the current package.

Task 2

Consider the following Java code:

```java
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.

solution

Invariant 1 cannot be broken by clients outside `p` because the field `Node.next` is private and can only be set in the constructor to an argument of the constructor, which must point to an already existing list that does not include the object currently being created.

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.

solution

Invariant 1 cannot be broken from inside `p` for the same reasons as above.

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.

solution

Invariant 2 cannot be broken from outside `p` because:
The invariant depends only on the fields `Node.next`, `Node.value`, and `List.head`.
Both `Node` fields are only written to in the constructor of `Node` and cannot be modified later as they are private.
The constructor of `Node` is of package access and so cannot be called directly by the client.
The only public method that calls it is `List.prepend`, which ensures invariant 2 - hence no decreasing list of nodes (whether or not attached to a `List`) can be created by clients of the package. So, although we can assign `List.head` any value, we cannot obtain a value (a `Node`) that would break the invariant.

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.

solution

Invariant 2 can be broken as follows (all code inside `p`):

```java
class Client{
   void client(){
      List list = new List();
      list.prepend(0);
      Node n = new Node(1, list.getHead());
      list.head = n;
   }
}
```

Task 3

Consider the following Java code:

```java
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.

solution

We can easily break the invariants through alias leaking. For instance, the following code breaks the invariant of class `Time`:

```java
Time t = new Time();
Hour h = t.getHour();
h.h = -1;
```

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?

solution

We can fix this in two ways. We have to avoid the alias leaking. We can reach this goal returning an integer value instead of an object, or a copy of the `Hour` object stored in the current `Time` object.

```java
public int getHour() { return hour.h; }
public Hour getHour() { return (Hour) hour.clone(); }
```

In general, it is simpler for reasoning, if possible, to return only primitive values, or to avoid exposing aliases of the local state of the object, by instead returning copies of the stored objects. In this way, we can avoid alias leaking, thus no external code can modify the values contained in the current object.

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

solution

The drawback of the second approach is that we are creating a new object and thus are using more memory. Additionally, client code that uses reference equality to check if the `Hour` object returned by `getHour()` is equal to another `Hour` object breaks if `getHour()` returns a new object on every call.

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.

solution

We could hide the `h` field of `Hour` by making `Hour` implement an interface `IHour` that has no mutator methods. `Time.getHour()` could then return this interface.

The client could still downcast from `IHour` to `Hour` and break the invariant but aside from that the invariant is protected. This could be prevented by making `Hour` a private inner class of `Time`.

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`.

solution

The invariant can be broken by exploiting the fact that `CList` captures and stores `Coordinates` objects.

```
CList list = new CList();
Coordinates c = new Coordinates(2, 1);
list.add(c);
c.x = 0;
```

B) How can we fix this problem?

solution

To fix `CList` we need two things

- We need to clone the `Coordinates` element before storing it.

```java
public void add(Coordinates el) {
   if (el.x > el.y) super.add((Coordinates) el.clone());
}
```

- We also need to clone the `Coordinates` element before returning it, as otherwise we leak a reference that could be modified.

```java
public Coordinates get(int i) {
   return (Coordinates) super.get(i).clone();
}
```

The drawback of such an approach is that we create a copy of all the elements stored in the list. It is not possible to make sure the invariant is preserved without creating objects that are only in the current `CList` object.

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?

solution

A possible solution would be to have final fields in class `Coordinates`. This would ensure that the invariant cannot be broken, but it requires the allocation of new objects each time we want to modify the fields. For instance, the following code:

```java
Coordinates c = new Coordinates(2, 1);
c.x = 0;
```

would have to be re-written to

```java
Coordinates c = new Coordinates(2, 1);
c = new Coordinates(0,1);
```

which allocates a new object even though this is not necessary (since the object pointed by `c` is not shared, and so changing its fields cannot break the invariants of other objects).

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

solution

The main benefit of alias sharing in data structures is to minimize the consumption of memory. In addition, we may want to share aliases on data structures, for instance, in order to further update the content of an element in a list. The main drawback is that alias sharing does not allow us to reason locally about the objects stored in the data structure, since clients could retain references to objects they store in the data structure, and might therefore modify the contents of these objects after they were stored.

Task 5

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

```java
package security;

public class User {
   public String name;
   public String password;
```

```java
   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:

```java
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.

solution

The body of the `malicious` method could look like this:

```java
void malicious(Login l) {
   User u = new User("user","pass");
   l.registerUser(u);
   u.name = null;

   try {
      l.login(u);
   }
   catch(LoginException e) {
      boolean success = l.login(e.problemUser);
      // Logged in as the user that was registered before user u
   }
}
```

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?

solution

- We could make both fields of `User` have the default (package) access:

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

Therefore, code outside the package will not be able to change existing `User` objects and the `malicious` method could not cause the exception as before.

- only modifying the `LoginException` class?

solution

The `LoginException` class currently captures the value of the problematic user. Instead it could create a new user that has the same name as `problemUser` but hides the password.

```java
public class LoginException extends RuntimeException {
   public User problemUser;
   public LoginException(String message, User problemUser) {
      super(message);
      this.problemUser = new User(problemUser.name, "****");
   }
}
```

This way, even if an exception is thrown, that refers to the wrong user name, the user's password will not be leaked.

- only modifying the `registerUser` method?

solution

We can change the `registerUser` method so that it does not capture its argument:

```
public void registerUser(User u) {
   if (u == null || u.name == null || u.password == null
      || u.name.isEmpty() || u.password.isEmpty()) return;
   users.add(new User(u.name, u.password));
}
```

Now we would not be able to modify the internal structure of the `Login` class by modifying the user we just registered in the `malicious` method.

- only modifying the body of the `for` loop inside the `login` method?

solution

This for loop actually contains a bug which allows the exploit to work. To fix it we must move the assignment to the `current` variable to the beginning of the loop:

```
for(User registered : users) {
   current = registered;
   boolean nameEqual = registered.name.equals(u.name);

   ...
}
```

In the original code we were able to cause an exception regarding a particular user, but report the previous user as invalid, since `current` was not updated yet. This is no longer the case.

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.

solution

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

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.

solution

```
/// requires c1 != c2;
void setCells(Cell c1, Cell c2)
...
```

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.

solution

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

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

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

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

   private Cell(CellInt ci) { value = ci; }

   private CellInt value;

   private class CellInt {
      CellInt(int newValue) { value = newValue; }
      int get() { return value; }
      void set(int newValue) { value = newValue; }
      private int value;
   }
}
```

The `clone` method now creates a new `Cell` that shares the representation (the `CellInt`), and so modifying the cloned or the original `Cell` also modifies the other.

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.

solution

```
   /// requires reach(c1) disjoint reach(c2);
   void setCells(Cell c1, Cell c2)
   ...
```

Now the reach of the arguments `c1` and `c2` are disjoint, so modifying one cannot affect the other in any way.

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`.

solution

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

Strengthening the postcondition of `Cell.clone` like that has the following consequences:

- The implementation of `Cell.clone` from subtask C) can no longer be verified since it does not guarantee the new postcondition (the `reach` sets won't be disjoined)
- The bodies of the methods `left` and `right` should therefore verify (modularly), and indeed will: `Cell.clone`'s stronger postcondition now establishes the precondition of `setCells`