Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 9

Information hiding, encapsulation and object structures

November 21, 2014

Task 1

Java allows an object of a class `C` to access the private fields of other objects declared in `C`. Discuss the resulting level of information hiding, its advantages and limitations, and provide some examples.

What is the policy concerning the visibility of protected fields of other objects?

Task 2

Consider the following Java code:

```
file A.java:
package p;
public class A {}

file B.java:
package p;
class B extends A {public int x;}

file C.java:
package p;
public class C extends B {          //1
  public B get(){...}               //2
  public void set(B d){...}         //3
}

file client.java:
package client;
import p.*;
class Client
  void f(){
    C c = new C();

    c.set(c.get());     //4
    c.set(c);           //5

    A a = c.get();      //6
    a = c;              //7
    c.x=1;              //8
  }
  void g(C c){
    c.get().x=1;        //9
  }
```

The code of `f` is accepted by the Java compiler, while `g` is rejected.

A) What seems inconsistent with information hiding in this code? Which lines would you expect the compiler to reject? Could allowing this still be useful in some cases?

B) Does this problem exist in C++? C#? Scala? If not explain which line the respective compiler would complain about.

C) Suggest a rule the Java compiler could enforce in order to be more consistent with information hiding - the rule should reject all marked lines.

D) Now suppose we wanted to support also lines 1 and 7 (could you suggest why this might be useful?) how can we relax our new rule to allow this?

Task 3

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.

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 4

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;
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 `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`.