ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 9

Information hiding, encapsulation and object structures

November 22, 2013

In-class Assessment: A subset of the questions from this sheet will be used for the in-class assessment. No notes are allowed during the assessment.

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?

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`.

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

`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.

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

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

C++ developers often talk about binary and source compatibility. Assume we have some class `C` and we make a change to it so that, now we have a newer version of the class `C'`:

- If all clients of `C` work just as well with `C'` **without even recompiling the clients**, then `C` and `C'` are *binary compatible*.
- If all clients of `C` work just as well with `C'` **without modifying the clients, but after recompiling them**, then `C` and `C'` are *source compatible*.
- If clients of `C` may need source code adjustments to work with `C'`, then `C` and `C'` are *not compatible*.

Well encapsulated C++ classes have a much higher chance to be binary compatible with many modifications. This is desirable, since it makes it possible to update the classes without breaking clients.

Consider the following C++ class:

```
// Point.h
class Point {
   private:
      int x_,y_;
   public:
      Point(int x, int y);
      int x();
      int y();
};
```

```
// Point.cpp
#include "Point.h"
Point::Point(int x, int y)
   : x_{x}, y_{y} {}
int Point::x() { return x_;}
int Point::y() { return y_;}
```

And the following client of that class:

```
// Client.cpp
#include <iostream>
#include "Point.h"

int main() {
   Point points[3] = {{1,2},{3,4},{5,6}};
```

```
    for(auto p : points) std::cout << p.x() << p.y();
    return 0;
}
```

The `main` function allocates three objects of type `Point` on *the stack*. Since `main` allocates the memory, it needs to know precisely how much space an object of type `Point` takes. This illustrates the fact that in C++, the memory layout of objects (also known as the object model) is part of the public interface of a class. Knowing this, and without modifying the client in any way, do the following:

- Find two different ways to add something to the private part of `Point`, such that the new code is not binary compatible with the client, but it is still source compatible. Do not change existing member declarations in any way. For both cases write only the new code and provide a brief explanation why this breaks the client.
- Describe a better design of `Point` that will make the class more encapsulated. Your design should allow you to implement at least one of the changes above in a binary compatible way. Do not make any changes to the client code.

Hint: If you want to experiment, you can use the following to compile the `Point` class and the `main` function separately:

- `Point` on Windows:
  ```
  g++ -std=c++11 -shared -fPIC -Wl,--export-all-symbols
  -o libpoint.so Point.cpp
  ```
- `Point` on Linux:
  ```
  g++ -std=c++11 -shared -fPIC -o libpoint.so Point.cpp
  ```
- `Client`:
  ```
  g++ -std=c++11 -L. -Wl,-rpath,'.' Client.cpp -o client -lpoint
  ```

Solution

- We can add a new field:

  ```
  private: int z_,x_,y_;
  ```

  or a new virtual private method with an implementation:

  ```
  private: virtual void transform() {
      x*=2;
      y*=2;
  };
  ```

  In both cases we change the memory layout of the `Point` class and the client needs to be recompiled to account for this. Otherwise the stack of the `main` function will be corrupt. In the first case each object will need a space for 3 integers, but `main` will only allocate 2 integers per object. In the second case for each object we need 2 integers, plus a pointer to the virtual table for class `Point`. A virtual table is needed whenever a class has at least one virtual method and is used for dynamic dispatch - the table depends on the dynamic type of the receiver, and it determines what method should be called.

- It is possible hide the internal fields of a class from its clients using *opaque pointers* [1], [2]:

  ```
  // Point.h
  class PointData;  // This is only declared, the actual
  // implementation is completely hidden from the clients
  ```

[1] http://en.wikipedia.org/wiki/Opaque_pointer

[2] http://techbase.kde.org/Policies/Binary_Compatibility_Issues_With_C++#Using_a_d-Pointer

```
class Point {
   private:
      PointData* data_;
   public:
      Point(int x, int y);
      ~Point();
      int x();
      int y();
};

// Point.cpp
#include "Point.h"
class PointData {
   public:
      PointData(int x, int y): x{x}, y{y} {}
      int x;
      int y;
};

Point::Point(int x, int y) : data_{new PointData{x,y}} {}
Point::~Point() {delete data_;}
int Point::x() { return data_->x;}
int Point::y() { return data_->y;}
```

The `data_` pointer isolates the clients of `Point` from its internal representation. The key here is that when a new `Point` is created, even if it is on the stack, its data will be placed on the heap and will be created by the library (not by the client like before).

If the `Point` class had been written in the way above, it is easy to just add the extra field to the `RectData` class.

The drawback here is that access to the data members of `Point` now requires one level of indirection which might have a negative effect in performance critical applications.

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.

- 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.
- Is it possible to fix the problem by:
  - 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?

  In each of these cases, explain how you can prevent the malicious login or why it is not possible.

Solution

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

```
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 registeted before user u
        }
    }
```

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

    ```
    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.

  - 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.

    ```
    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.

  - 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.

  - 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 an invalid, since `current` was not updated yet. This is no longer the case.

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`.

Solution

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

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

(c)
```
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 original `Cell` also modifies the other.

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