Exercise 1:
The Commandments of Software Engineering

Hand-out: 2 April 2004
Due: 16 April 2004

Master Solution

1. Summary: Modularity principles

Uniform Access principle
Facilities managed by a module are accessible to its clients in the same way whether implemented by computation or by storage.

Information Hiding principle
The designer of every module must select a subset of the module’s properties as the official information about the module, to be made available to authors of client modules.

Open-Closed principle
Modules should be open (may be extended) and closed (usable by clients).

2. Uniform Access principle
Here is a possible implementation of a class BankAccount in Java:

```java
import java.util.*;

class BankAccount {
    ...
    public int balance() {
        int depositSum = 0;
        int withdrawalSum = 0;

        for (int i = 0; i < deposits.size(); i++) {
            depositSum = depositSum + ((Integer)deposits.get(i)).intValue();
        }
    }
}
for (int i = 0; i < withdrawals.size(); i++) {
    withdrawalSum = withdrawalSum + ((Integer) withdrawals.get(i)).intValue();
}
return depositSum – withdrawalSum;

protected ArrayList deposits;
protected ArrayList withdrawals;

To do

- Does this style observe the principle of Uniform Access?
- What are the benefits of applying the Uniform Access principle?

Hint

Examine where you need parentheses in Java.

To hand in

Hand in your answers to the two questions above (in the “To do” section).

Solution

- Does this style observe the principle of Uniform Access?

When looking at the code deposits.size(), the client can immediately see that deposits is an attribute (a “field” in Java terminology) and that size is a routine of class ArrayList (a “method” in Java terminology) because there is no opening and closing parentheses to access attributes whereas parentheses are compulsory for routine calls. This policy breaks the principle of Uniform Access; clients should not know whether a service is implemented by computation (routine) or by storage (attribute).

- What are the benefits of applying the Uniform Access principle?

Hiding implementation from the client (i.e. hiding the fact a service is implemented by computation or by storage) provides flexibility. It is possible to change the implementation (for example, decide to use an attribute rather than a routine) at no cost for the clients. The supplier does not have to tell its clients about this change; the client code will continue to work. It would not be the case in the Java world where the client needs to know whether an operation is implemented as a “field” or as a “method” because he needs to add the corresponding parentheses. If the supplier changes the implementation, the client code would not compile anymore.
3. Information Hiding principle

Here is a bad application written in Java:

```java
import java.lang.*;

class BadApplication {
    ....
    public void changeBalance() {
        account.balance = 1000;
    }
    ...
    protected BankAccount account;
}
```

using the following class BankAccount:

```java
import java.lang.*;

class BankAccount {
    ....
    public int balance;
}
```

To do

- Why does this example violate the principle of Information Hiding?
- How would you solve the problem?
- Write the equivalent example in Eiffel, applying standard style rules.

Hint

The Eiffel code corresponding to the above Java classes would not compile.

To hand in

Hand in your answers to the two questions above (in the “To do” section) and the text of your Eiffel classes APPLICATION and BANK_ACCOUNT.

Solution

- Why does this example violate the principle of Information Hiding?

The attribute balance of class BankAccount is public. As a consequence, the client class BadApplication can modify the account’s balance directly without using a setter procedure (account.balance = 1000). In Eiffel, such code would be forbidden. Indeed, it is dangerous to let clients modify attribute values without the supplier knowing about it. This could break fundamental properties and consistency of the object.
• How would you solve the problem?

Make the attribute balance private in class BankAccount. Define a procedure setBalance that enables clients to change the balance value, i.e.:

```java
public void setBalance (int aValue) {
    balance = aValue;
}
```

Use this procedure setBalance in routine changeProcedure of the application class.

• Write the equivalent example in Eiffel, applying standard style rules. (Contracts are not included in the Eiffel code – although it would be important to express them – because this notion has not been seen in class yet.)

```eiffel
class APPLICATION
...
feature -- Access
    account: BANK_ACCOUNT
        -- Bank account

feature -- Element change
    change_balance is
        -- Set the `account' balance to 1000.
        do
            account.set_balance (1000)
        end
...
end

class BANK_ACCOUNT
...
feature -- Access
    balance: INTEGER
        -- Account balance

feature -- Element change
    set_balance (a_value: INTEGER) is
        -- Set `balance' to `a_value'.
        do
            balance := a_value
        end
...
end
```
4. Open-Closed principle

To do
Give an example showing the importance of the **Open-Closed principle**. (Explain the benefits of this principle.)

Hints
- Think about inheritance and client relationships.
- You may view the question as: what would *not* be possible without this principle?

To hand in
Hand in the example and accompanying explanations.

Solution
- Give an example showing the importance of the Open-Closed principle. (Explain the benefits of this principle.)

Let’s take the example of the bank account again. Say we have a class `BANK_ACCOUNT` as before. To be usable by a client, this class must be “closed”, i.e. it must compile and have a clear specification (typically under the form of contracts, but this notion will be introduced later in the course). If these conditions are met, the class `BANK_ACCOUNT` may have one client, say `APPLICATION`.

![Diagram of APPLICATION to BANK_ACCOUNT](attachment:diagram.png)

Now, let’s suppose that the financial company for which you developed the class `BANK_ACCOUNT` would like to have a more precise representation of bank accounts, in particular it would like to have a class `NUMBERED_BANK_ACCOUNT`. If the class `BANK_ACCOUNT` that you developed before is not “open”, than you have to develop the new class `NUMBERED_BANK_ACCOUNT` from scratch. It works and client applications, say `APPLICATION_2` could use it.

![Diagram of APPLICATION_2 to NUMBERED_BANK_ACCOUNT](attachment:diagram2.png)

But it is likely that the implementation of `NUMBERED_BANK_ACCOUNT` resembles the one of `BANK_ACCOUNT` a lot. To avoid code duplication, the idea is to make class `NUMBERED_BANK_ACCOUNT` a descendant of class `BANK_ACCOUNT`, i.e. allow the class to reuse the features of class `BANK_ACCOUNT` (inheritance has not been introduced in class yet).
We say that the class `BANK_ACCOUNT` is “open” because it can be reused to build a new class `NUMBERED_BANK_ACCOUNT`; but it is also “closed” because it can be used by clients, like class `APPLICATION`, and the introduction of a new class `NUMBERED_BANK_ACCOUNT` has no consequence on the client `APPLICATION`. 