UTCN
Computer Science Department
Software Design 2011/2012

CLASS AND PACKAGE DESIGN PRINCIPLES

==========

I. Theoretical Background

This section briefly describes the class and package design principles. The information presented here is based on [1, 2, 3, 4, 5].

1. Class Design Principles

Features of poor design

- *Rigidity* – the tendency for software to be difficult to change, even in simple ways. Every change causes a cascade of subsequent changes in dependent modules.
- *Fragility* – the tendency of the software to break in many places every time it is changed. Often the breakage occurs in areas that have no conceptual relationship with the area that was changed.
- *Immobility* **–** the inability to reuse software from other projects or from parts of the same project.

Causes of poor design

- Changing requirements
- Poor dependency management – when introducing improper dependencies between the modules of the software.

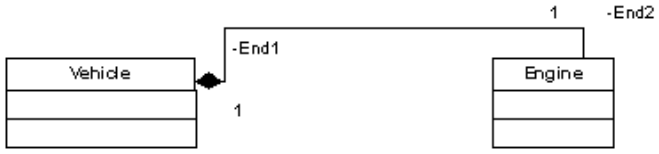
Principles of Object Oriented Class Design

a) The Open Closed Principle (OCP)

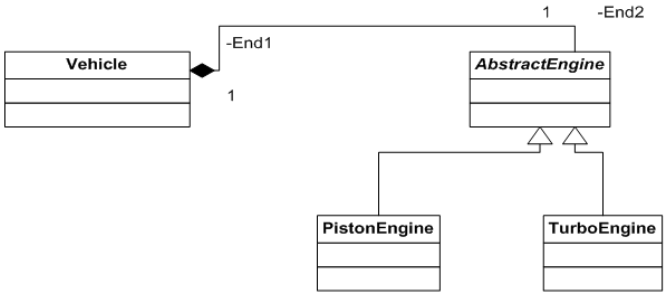
Main idea: Modules should be written so that they can be extended, without requiring them to be modified (*open for extension but closed for modification*).

Example:

- What if we want to change the Engine type... should we change the car?



- Use dependencies on abstract classes and not on concrete ones!



b) The Liskov Substitution Principle (LSP)

Main idea: Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.

Example*:* Consider the Rectangle class defined below:

```
class Rectangle
{
   public:
      void SetWidth(double w) {itsWidth=w;}
      void SetHeight(double h) {itsHeight=h;}
      double GetHeight() const {return itsHeight;}
      double GetWidth() const {return itsWidth;}
   private:
      double itsWidth;
      double itsHeight;
};
```

Consider the class Square which inherits Rectangle and overrides the *setWidth* and *setHeight* functions by duplicating the code.

Consider the following function:

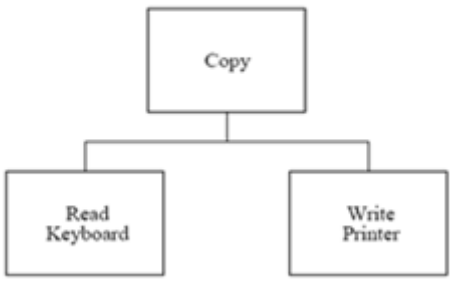
```
void g(Rectangle& r)
{
   r.setWidth(4);
   r.setHeight(5);
   assert(r.getWidth()*r.getHeight()==20);
}
```

This function invokes the *setWidth* and *setHeight* members of what it believes to be a *Rectangle*. The function works just fine for a *Rectangle*, but declares an assertion error if passed a *Square*.

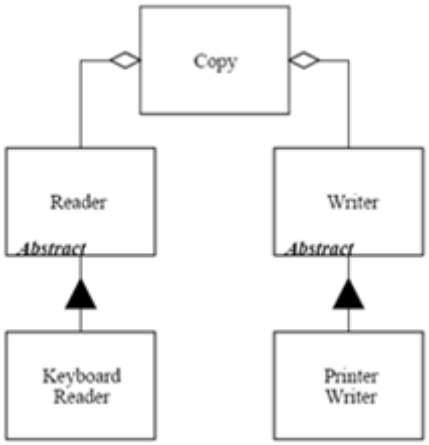
c) Dependency Inversion Principle

Main idea: Dependency Inversion is the strategy of depending upon interfaces or abstract functions and classes, rather than upon concrete functions and classes. In this context, high-level modules should not depend on low-level modules, and abstractions should not depend on details. A base class in an inheritance hierarchy should not know any of its subclasses.

Example: Consider a program that copies characters typed on a keyboard to a printer. The figure below shows that there are 3 modules, in the application: the "Copy" module which calls the „Read Keyboard" and „Write Printer" modules. The two low level modules are reusable and can be used in many other programs to access the keyboard and the printer. However, the "Copy" module is not reusable in any context which does not involve a keyboard or a printer.



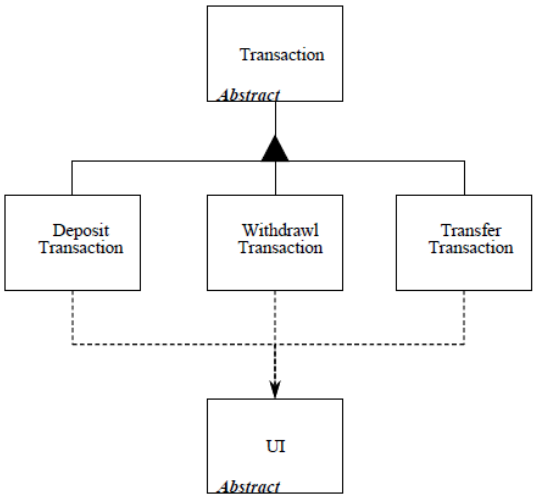
Consider the simple class diagram below where the “Copy” class contains an abstract “Reader” class and an abstract “Writer” class. The “Copy” class gets characters from its “Reader” and sends them to its “Writer”. In this way, the “Copy” class does not depend upon the “Keyboard Reader” nor the “Printer Writer” at all => the dependencies have been inverted as the “Copy” class depends upon abstractions, and the detailed readers and writers depend upon the same abstractions.



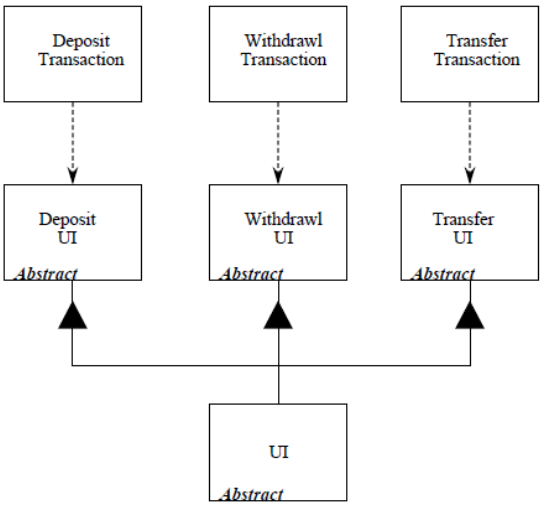
d) Interface Segregation Principle

- **Main idea:** Clients should not be forced to depend upon interfaces that they don’t use. Many client-specific interfaces are better than one general purpose interface.

- **Example:** Consider that each different transaction that the ATM can perform is encapsulated as a derivative of the class Transaction. Thus we might have classes such as DepositTransaction, WithdrawlTransaction, TransferTransaction, etc. Each of these objects issues message to the UI. For example, the DepositTransaction object calls the RequestDepositAmount member function of the UI class. Whereas the TransferTransaction object calls the RequestTransferAmount member function of UI. This is precicely the situation that the ISP tells us to avoid. Each of the transactions is using a portion of the UI that no other object uses. This creates the possibility that changes to one of the derivatives of Transaction will force coresponding change to the UI, thereby affecting all the other derivatives of Transaction, and every other class that depends upon the UI interface.



This unfortunate coupling can be avoided by segregating the UI interface into induvidual abstract base classes such as DepositUI, WithdrawUI and TransferUI.



2. Package Design Principles

Objective: *partition* the classes in an application according to some *criteria* and then *allocate* those partitions to packages.

Principles of OO High-Level Design

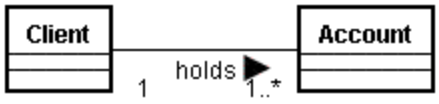
- ***Cohesion Principles***
    - *Common Reuse Principle (CRP)* - All classes in a package should be reused together. If you reuse one of the classes in the package, you reuse them all.
        - Criteria for grouping classes in a package: classes that tend to be reused together.
        - Packages have physical representations (shared libraries, DLLs, assembly)
            - Changing just one class in the package -> rerelease the package -> revalidate the application that uses the package.
    - *Common Closure Principle (CCP)* - The classes in a package should be closed against the same kinds of changes. A change that affects a package affects all the classes in that package.
- ***Coupling Principles***
    - *Acyclic Dependencies Principle (ADP)* - The dependency structure for released component must be a Directed Acyclic Graph (DAG). There can be no cycles.
    - *Stable Dependencies Principle (SDP)* - Depend in the direction of stability (Stability is related to the amount of work in order to make a change.)
    - *Stable Abstractions Principle (SAP)* - Stable packages should be abstract packages.

II. Problems

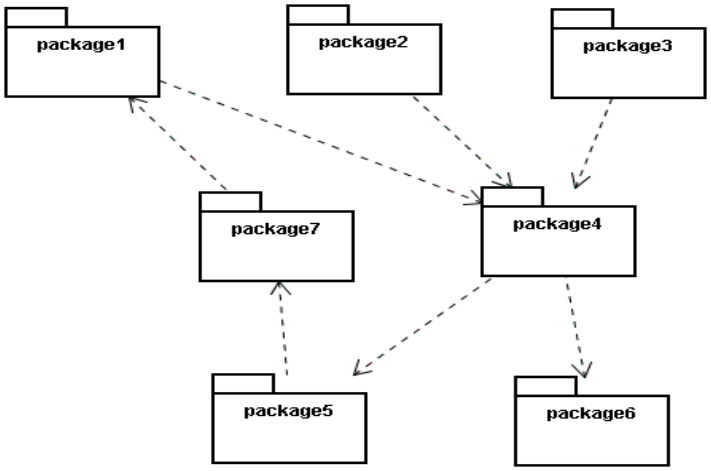
P1: Consider the classes below. What principle do the classes CurrentAccount and SpecialCurrentAccount not respect?

```
class CurrentAccount
{
      private double balance;
      private int period;
      //constructor, setters, getters

      public boolean closeAccount()
      {
            if(balance>0)
                  return true;
            else
                  return false;
      }
}

class SpecialCurrentAccount extends CurrentAccount
{
      private int minimumPeriod;
   //constructor, setters, getters

      public boolean closeAccount()
      {
            if((balance>0)&&(period>minimumPeriod))
                  return true;
            else
                  return false;
      }
}

class AccountTest{
…
     public void closeAnAccount(CurrentAccount ac)
     {
           System.out.println("Account close result: "+ac.closeAccount());
     }

     public static void main(String[] args)
     {
           AccountTest test = new AccountTest();

           CurrentAccount ac = new CurrentAccount(100,2);
           SpecialCurrentAccount sac = new SpecialCurrentAccount(200,5);
           test.closeAnAccount(ac);
           test.closeAnAccount(sac);
    }
}
```

P2: Consider the following design of a bank client and his bank accounts. Which class Design principle (LSP, OCP, DIP) is not respected?



P3: What package design principle is NOT respected in the following diagram?



References

[1] http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf
[2] http://www.objectmentor.com/resources/articles/lsp.pdf
[3] http://www.objectmentor.com/resources/articles/isp.pdf
[4] http://www.objectmentor.com/resources/articles/dip.pdf
[5] http://www.objectmentor.com/resources/articles/ocp.pdf