The Dependency-Inversion Principle

“High level modules should not depend on low-level modules. Both should depend on abstractions”
“Abstractions should not depend on details. Details should depend on abstraction”

- Functional software development describes higher level modules in terms of lower level modules
  - A change in a lower level module can trigger a change in the higher level module
  - Higher level modules are not independent of lower level modules and thus not re-usable
The Dependency-Inversion Principle

- Example: based on the requirement of “well-defined interfaces” one could design a program which looks as follows

  - Policy Layer sensitive to changes in the Utility Layer

- Better approach: abstract the interfaces of the lower layers out
  - The abstract interface is “owned” by the upper layer!
The Dependency-Inversion Principle

- DIP heuristic:
  - Do not depend on a concrete class, depend on an abstract class or abstract interface
    - No variables should hold a pointer or a reference to a concrete class
    - No class should be derived from a concrete class
    - No method should override an implementation method of any of its base classes
  - Heuristic is violated at least in main, since you need to create concrete instances of your class
  - No reason to follow this principle for non-volatile classes (e.g. class String)
  - DIP does not protect versus interface changes of the abstract class, but against implementation changes.

A simple example

- Consider a Button and a Lamp object
  - Wrong design violating DIP: Button depends directly on Lamp

```java
public class Button {
    private Lamp itsLamp;
    public void poll() {
        if (/* some condition */) {
            itsLamp.TurnOn();
        }...
    }
}
```
A simple example

- However, what happens if a Lamp would like to be controlled by an object different than Button, e.g. Switch?
  - Button Server independent of a Button, dependency is in the name only
  - Interface Button Server can be part of a separate package and does not have to be part of the Button object/class

The furnace example

- Consider a software controlling a furnace. The software can read the temperature through an I/O channel and turn the furnace on/off by sending commands to a different I/O channel

```c
void regulate ( double minTemp, double maxTemp ){
    for ( ; ; ) {
        while ( in(THERMOMETER) > minTemp )
            wait (1);
        out ( FURNACE, ENGAGE );
        while ( in(THERMOMETER) < maxTemp )
            wait (1);
        out ( FURNACE, DISENGAGE );
    }
}
```
The furnace example

- Design following DIP

```cpp
void Regulate ( Thermometer& t, Heater& h, double minTemp, double maxTemp )
{
    for ( ; ; ) {
        while ( t.read() > minTemp )
            wait (1);
        h.engage();

        while ( t.read() < maxTemp )
            wait (1);
        h.disengage();
    }
}
```

- Alternative solution for C++: usage of templates instead of abstract classes
  - Removes the possibility to add a new types of Thermometers and Heaters at runtime
  - Dynamic vs. Static Polymorphism
The Interface-Segregation Principle

“Clients should not be forced to depend on methods that they do not use”

- Main problem: how to deal with classes having ‘fat’ interfaces
  - Interfaces of the class can be broken up into groups of methods
  - Each group servers a different set of clients

Door-lock problem

```cpp
class Door {
  public:
    virtual void Lock() = 0;
    virtual void Unlock() = 0;
    virtual bool IsDoorOpen() = 0;
};
```

- Consider an implementation of Door called TimedDoor which needs to sound an alarm when the door has been left open
  - TimedDoor needs to communicate with another Object called Timer
Door lock problem

```cpp
class Timer {
    public:
        void Register(int timeout, TimerClient *client);
    
    Class TimerClient {
    public:
        virtual void TimeOut() = 0;
    
    
    ```

- When an object wishes to be informed about a time-out, it needs to register with the Timer
- How can a TimerClient communicate with the TimedDoor class?

Naïve solution: we force Door to inherit from TimerClient
- Problematic since every Door class now depends on TimerClient
- Timing free derivatives of Door will often provide degenerated implementations for TimeOut -> violation of LSP
- Timing free derivatives of Door will have to import TimerClient
- Interface polluted solely for the benefit of one subclass
- Door and TimerClient represent separate interfaces that are used by different clients
    - Should remain separate interfaces, do not merge them to a single interface
Separation through Delegation

- Introduction of a Door Timer Adapter object
  - Confirms to ISP, since Door and Timer are separated
  - Inelegant, since it requires the creation of a new object every time we wish to register a time-out

```cpp
class TimedDoor: public Door {
    public:
        virtual void DoorTimeOut(int timeoutID);
    }

class DoorTimeAdapter: public TimerClient {
    public:
        DoorTimeAdapter(TimedDoor& theDoor) :
            itsTimedDoor(theDoor) {}
        virtual void TimeOut(int timeout) {
            itsTimedDoor.DoorTimeOut(timeoutID);
        }
        private:
            TimedDoor& itsTimedDoor;
    }
```

Separation through Delegation
Separation through Multiple Inheritance

- TimedDoor can inherit from both Door and TimerClient

```cpp
class TimedDoor: public Door, public TimerClient {
    public:
        virtual void TimeOut(int timeoutID);
};
```

Let’s talk first about something different

```cpp
vector<Shape*>::iterator i;
for (i=list.begin(); i!= list.end(); i++ ) {
    (*i)->Draw();
}
```

- We started to discuss the last time while having a function in the termination condition of a loop is not good for the performance
An example

for ( i=1000; i > 0 ; i-- ) {
    x[i] = x[i] + s;
}

Loop: L.D F0, 0(R1)
ADD.D F4, F0, F2
S.D F4, 0(R1)
DADDUI R1, R1, #8
BNE R1, R2, Loop

Assumptions

<table>
<thead>
<tr>
<th>Instruction producing results</th>
<th>Instruction using results</th>
<th>Latency in clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU</td>
<td>FP ALU</td>
<td>3</td>
</tr>
<tr>
<td>FP ALU</td>
<td>Store</td>
<td>2</td>
</tr>
<tr>
<td>Load FP</td>
<td>FP ALU</td>
<td>1</td>
</tr>
<tr>
<td>Load FP</td>
<td>Store</td>
<td>0</td>
</tr>
</tbody>
</table>

• 1 cycle branch delay
An example (III)

Loop: L.D  F0, 0(R1)  1
          s          2
ADD.D    F4, F0, F2  3
          s          4
          s          5
S.D      F4, 0(R1)   6
DADDUI   R1, R1, #-8  7
          s          8
BNE      R1, R2, Loop 9
          s          10

Rescheduling the code

Loop: L.D  F0, 0(R1)  1
        DADDUI  R1, R1, #-8  2
        ADD.D  F4, F0, F2  3
        s          4
        BNE      R1, R2, Loop 5
        S.D      F4, 8(R1)  6  ↓

Delayed branch slot

- Each loop iteration consists of 3 instructions of actual work (load, add, store) and 3 cycles loop overhead
Loop unrolling

Loop:  
L.D F0, 0(R1)  
ADD.D F4, F0, F2  
S.D F4, 0(R1) /* Drop DADDUI & BNE */  
L.D F6, -8(R1)  
ADD.D F8, F6, F2  
S.D F8, -8(R1) /* Drop DADDUI & BNE */  
L.D F10, -16(R1)  
ADD.D F12, F10, F2  
S.D F12, -16(R1) /* Drop DADDUI & BNE */  
L.D F14, -24(R1)  
ADD.D F16, F14, F2  
S.D F16, -24(R1)  
DADDUI R1, R1, #-32  
BNE R1, R2, Loop

Loop unrolling (II)

- Eliminates three branches and three decrements  
  - Reduces loop overhead
- The previous code sequence still contains many stalls, since many operations are still dependent on each other
- Requires a large number of registers
- Increase of the code size
- For general application of loop unrolling:
  - Two consecutive loops
  - First executes \((n \mod k)\) times, not unrolled
    - \(n\): number of loop iterations
    - \(k\): unroll depth of the loop
  - Second executes \((n/k)\) times, unrolled
Scheduled version of the unrolled loop

Loop:  
L.D   F0, 0(R1)  
L.D   F6, -8(R1) 
L.D   F10, -16(R1)  
L.D   F14, -24(R1)  
ADD.D F4, F0, F2  
ADD.D F8, F6, F2  
ADD.D F12, F10, F2  
ADD.D F16, F14, F2  
S.D   F4, 0(R1)  
S.D   F8, -8(R1) 
DADDUI R1, R1, #-32  
S.D   F12, 16(R1)  
BNE R1, R2, Loop  
S.D   F16, 8(R1)