CS 246 - Object Oriented Programming **Fall 2016**

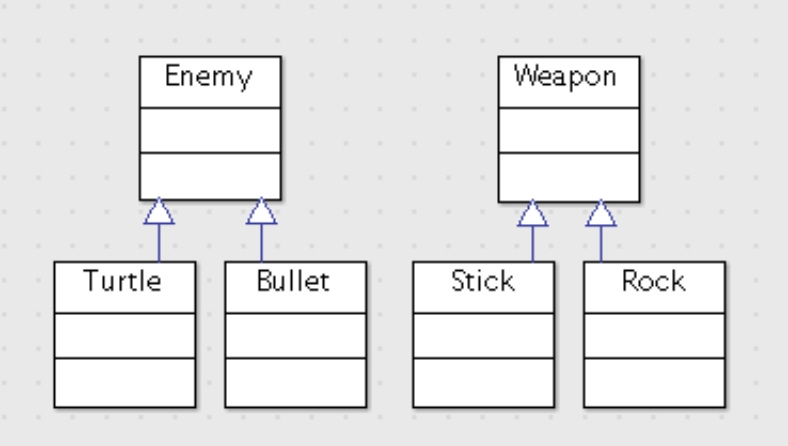
Lecture 10: November 17, 2016

Lecturer: Brad Lushman *Notes By: Harsh Mistry*

10.1 Visitor Pattern

The visitor pattern is often used to implementing a concept called **double dispatch**. Using this method, we can implement a program where action can be based upon two different objects.To achieve this goal we can combine overriding and overloading.

Example 10.1 *Basic video game with enemies and weapons using the visitor pattern*



We want a different action to occur when different weapons are used on different enemies. So we introduce a overloaded strike method.

```
1  class Enemy{
2      virtual void beStruckBy(Weapon &w) = 0;
3  };
4
5  class Turtle : public Enemy{
6      void beStruckBy(Weapon &w) override {
7        w.strike(*this);
8      }
9  }
10
11 class Bullet : public Enemy {
12     void beStruckBy(Weapon &w) override{
13       w.strike(*this); //Different then previous subclass, due to different parameter types
             of *this
14     }
15 };
16
17 class Weapon {
18     //Overloads
```

```
19      virtual void strike(Turtle &t) = 0;
20      virtual void strike(Bullet &t)  = 0;
21  };
22
23  class  Stick : public Weapon{
24      void strike (Turtle &t) override {
25        //Strike turtle with stick
26      }
27
28      void strike (Bullet &t) override {
29        //Strike bullet with stick
30      }
31  };
32
33  class Rock : Weapon {
34      void strike (Turtle &t) override {
35        //Strike turtle with rock
36      }
37
38      void strike (Bullet &t) override {
39        //Strike bullet with rock
40      }
41  };
42
43  //Client Code
44  Enemy *e = new Bullet(...);
45  Enemy *w = new Rock(...);
46
47  e->beStructBy(w);
```

Visitor pattern can also be used to add functionality to existing classes without changing or recompiling them.

Example 10.2 *Adding a visitor to the book hierarchy*

```
1   class Book{
2     public:
3       virtual void accept (BookVisitor &v) {v.visit(*this);}
4   };
5
6   class Text : public Book {
7       Void accept (BookVisitor &v)  override {v.vist(*this);}
8   };
9
10  class BookVisitor{
11      virtual void visit (Book &b) = 0;
12      virtual void visit (Text &b) = 0;
13      virtual void visit (Comic &b) = 0;
14  }
15
16  // Now we want to Track how many of each type of book is available
17  // - Books are sorted by author
18  // - Texts are sorted by topic
19  // - Comics are sorted by hero
20  // This can be done using a map<string, int>
21
22  class Catalogue : BookVisitor {
23      map<string, int> theCatalogue;
24    public :
25      map<string, int> getResult() {return theCatalogue;}
26      void visit (Book &b) override { ++ theCatalogue[b.getAuthor()];}
```

```
27      void visit (Text &t) override { ++theCatalogue[t.getTopic()];}
28      void visit (Comic &t) override { ++theCatalogue[t.getHero()];}
29  };
```

10.2 Compilation Dependencies - include vs forward declare

Consider :

```
1  Class A {...};
2  Class B : public A { ...}
3  Class C {
4      A myA();
5  };
6  Class D {
7      A *myAP
8  };
9  Class E {
10     A f(A a);
11 };
```

- Class B and C require include statements as the compiler needs to know more information about A before it can allocate the necessary space and construct the fields.
- Class D and E do not require include statements as the size of A does not need to be known to construct the object.

In addition, the implementations of each object can require the use of include statements, as the actual definition may require more information on additional classes. If this is the case, write the include within the .cc file, as it can never be apart of the include cycle.

So, If there is no compilation dependency necessitated by the code, **DO NOT** create one with unnecessary includes.

Now Consider the XWindow class :

```
1  Class XWindow {
2      Display *d;
3      Window w;
4      int s;
5      Gc gc;
6      unsigned long colours [10];
7    public :
8      ...
9  };
```

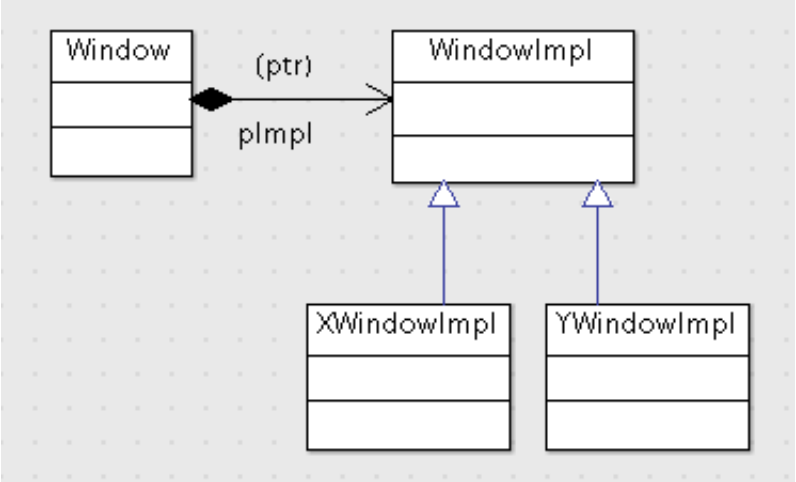
If any of the private data fields were to change, all clients must recompile. To avoid having to recompile the client, we can use the **pimpl idom** (Pointer to implementation) .

```
1  //Included Header (Window.h)
2  Class XWindowInput; //Forward Declaration
3
4  Class XWindow {
5      XWindowInput *pimpl;
6    public :
7      ...
8  };
9
```

```
10 //Separate Header File (xwindowImpl.h)
11 #include(X11/Xlib.h)
12
13 struct XWindowImpl{
14     Display *d;
15     Window *d;
16     window q;
17     int s;
18     GC gc;
19     unsigned long colours[10];
20 };
21
22 //Window.cc
23 #include "window.h"
24 #include "xwindowImpl.h"
25
26 Xwindow::Xwindow(): pimpl{new XWindowInpml {...{ } {...}
27
28 //In other methods replace d, w, s, etc with pimpl->d, pimpl->w, ...
```

If you confine all private fields to XWindowImpl, only window.cc needs to be recompiled if you change XWindow's implementation.

Generalization : If there is multiple X window implementations, then we can make the Impl struct a superclass



Implementing a class hierarchy with implementations is called the **Bridge Pattern**.

10.3 Measuring of Design Quality

10.3.1 Coupling and Cohesion

- Coupling : Coupling is the degree to which distinct modules depend on each other
  - Low Coupling
    - Modules communicate via function calls with basic parameters and results

  - Relative Amount of Coupling
    - Modules pass array/structs back and forth
    - Modules affect each other's control flow
    - Modules share global data
  - High Coupling
    - Modules have access to each other's implementations (friends)

- Cohesion : Cohesion is how closely elements of the module are related to each other
  - Low Cohesion
    - Arbitrary grouping of unrelated elements (e.g `<utility>`)
  - Relative Amount of Cohesion
    - Elements share a common theme, but are otherwise unrelated. Perhaps also share some base code (e.g `<algorithm>`)
    - Elements manipulate state over the lifetime of an object (e.g `open/read/close` files)
    - Elements pass data to each other
  - High Cohesion
    - Elements cooperate to perform exactly one task.

High coupling implies changes to one module requires changes to be made to other modules. This results in a implementation that is harder to reuse.

Low cohesion implies poorly orginized code and make its hard for the client to complete task and include classes which may not be needed.

Ultimately the goal is to have low coupling and high cohesion.

10.3.2 Decoupling the Interface (MVC)

Primary program classes should not be printing things.

Example 10.3 *Bad Design Example*

```
class ChessBoard {
    ...
    cout <<< "Your move" << endl;
};
```

Example 10.3 is a bad design habit as it ties the output to cout and prevents you from reusing the class for other purposes.

One solution is to give the class stream objects with which it can do Input/Output operations.

```
Class ChessBoard{
    std::istream &in;
    std::ostream &out;
  public :
    ChessBoard(std::istream &i, std::ostream &o) : in{i}, out{o} {}

    ...
    out << "your move" << endl;
};
```

This solution works, but still limits you to text output and prevents you from implementing different types of outputs (i.e different languages). A better implementation is to have the class not communicate at all.

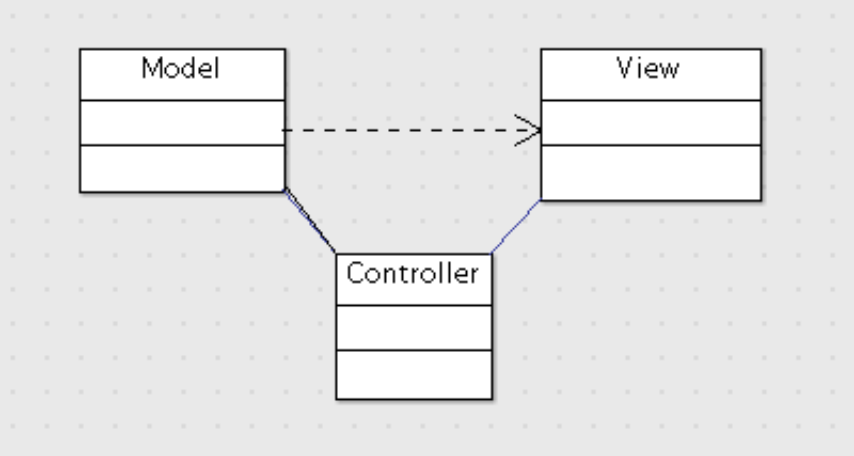
Single Responsibility Principle : "A class should have only one reason to change."

In essence, the idea for communicating with the chess board without printing is to simply make use of parameters and results. So , ensure use communication is excluded from main program classes.

Expanding further, main should not be used either to communicate with the user, as main is difficult to reuse. Ideally, main should be fairly short and a separate class should be used to handle communication. This idea is known as the **Model-View-Controller Pattern (MVC)**.

The goal of MVC is to separate out the distinct portions of the application

- Model - Data you are manipulating (e.g game state)
  - Can have multiple views
  - Classic observer pattern if you choose to communicate with the view directly
- View - How the model is displayed to the user
  - Indicates control flow through model and view
  - May encapsulate data
  - May fetch user input
- Controller - How the model is manipulated.



10.4 Exceptions Safety

Consider :

```
1 void f() {
2   MyClass *p =  new MyClass;
3   MyClass mc;
4   g();
5   delete p;
6 }
```

Within the example there are no leaks, but if g() raises an exception there may be leaks. During stack unwinding all stack allocated data is cleaned up, but heap allocated memory is not destroy. So, within the example, p would be leaked.

A better implementation :

```
void f() {
  MyClass *p =  new MyClass;
  MyClass mc;
  try {
    g();
  }
  catch (...) {
    delete p;
    throw;
  }

  delete p;
}
```

The above implementation works, but involved code duplication. In some languages, there is a "finally" clauses. Unfortunately, C++ is not one of those languages. In c++ the only thing you can count on in c++ is that destructor for stack allocated data will run. So, use stack allocated data as much as possible and use this to your advantage.

This leads to the popular c++ idiom : **RAII - Resource Acquisition is Initialization**. RAII states that every resource should be wrapped in a stack-allocated object whose destructor frees it.

```
ifstream f {name};
//The file is guaranteed to be released when f is popped from the stack (f's destructor is
    run)
```

This concept, can be applied to delete any heap allocated data by using `class std::unique_ptr <t>` which takes a T* in the constructor and will delete the pointer when the item is popped from the stack.

So an ideal implementation for the example with g() would be :

```
void f() {
  auto p = std::make_unique<MyClass> ();
  MyClass mc;
  g();
}
```