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

Lecture 9: November 10, 2016

Lecturer: Brad Lushman *Notes By: Harsh Mistry*

9.0.1 Observer Pattern

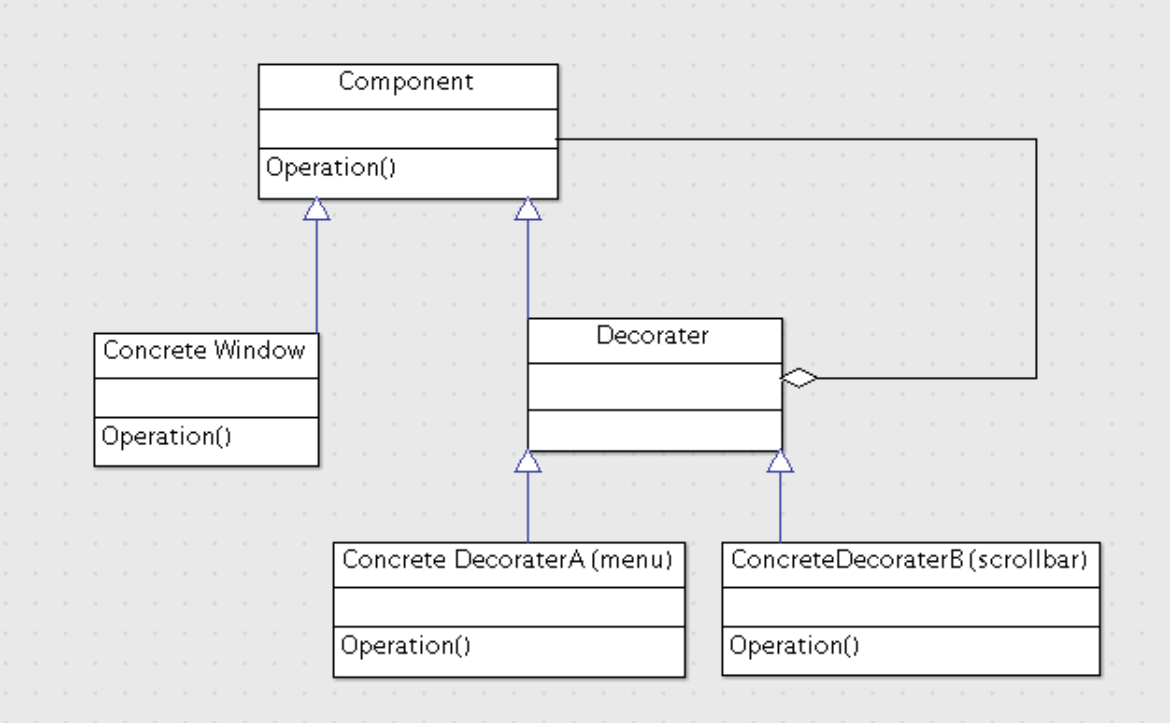
Example 9.1 *Horse Race*

```
Class Subject{
    vector <observer *> observers;
  public :
    void attach (observer *o){ observer.emplace_back(0);}
    void detach (observer *o);
    void notify Observers (){
      for (auto &ob:observers) ob->notify();
    }
    virtual ~Subject() = 0;
};

Class Observer {
  public :
    virtual void notify() = 0;
    virtual ~observer();
};

//Concrete Subject
Class HorseRace : public Subject {
    ifstream in; //source of data
    string lastWinner;
  public:
    HorseRace(const string &source) : in{source} {}
    bool runRace();  //true if successful
    string getState () { return lastWinner;}
};

//Concrete Observer
Class Better : public Observer {
    HorseRace *subject;
    string name, myHorse;
  public :
    Better ( ...) : ... {subject->attach(this);}
    ~Better() {subject->detach(this);}
    void notify () {
      string winner = subject->getState();
      if (winner == myhorse) {
        cout << "yipee!";
      }
      else {
        cout << "Double or nothing";
      }
    }
};

int main () {
  HorseRace hr;
  Better Larry (&hr, "Larry", "Runslikeacow");
  ...

```

```
51    while (hr.runRace()){
52      hr.notifyObservers();
53    }
54  }
55
56  Subject::~subject() {} // Must be defined, even though its pure virtual.
```

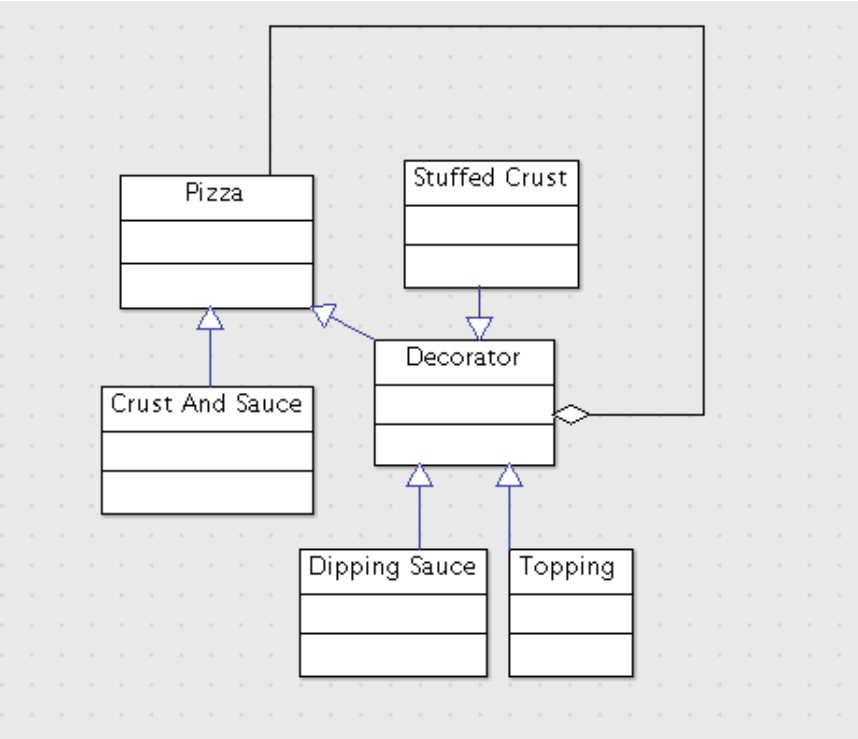
9.1 Decorator Pattern

The decorator pattern is used to add functionality or features during runtime.



- Class Component
  - Defines the interface operations objects will provide
- Concrete Component - Implements the interface
- Decorator - all components inherit from Decorator, which inherits from component. So, every decorator is a component and every decorator has a component.

Example 9.2 *Pizza*




```
1  Class Pizza {
2    public :
3      virtual float price () const =0;
4      virtual string desc() const = 0;
5      virtual ~pizza();
6  );
7
8  Class CrustAndSauce : public Pizza {
9    public:
10     float price() const override { return 5.99;}
11     string desc() const override { return "Pizza";}
12 };
13
14 Class Decorator : public Pizza {
15   protected :
16     Pizza component;
17   public :
18     Decorator (PIzza *p) : Component {p}{}
19     virtual ~Decorator() {delete component;}
20 };
21
22 Class Topping : public Decorator {
23     string theTopping;
24   public :
25     topping (string t, pizza *p) : Decorator {p}, theTopping{t}{}
26     float price () const override {
27       return component->price() + .75;
28     }
29     string desc() const override {
30       return component->desc() + " with " + theTopping;
31     }
```

```
};

int main () {
  Pizza *p1 = new CrustandSauce;
  p1 = new Topping ("Cheese", p1);
  p1 = new Topping ("Cheese", p1);

  cout << p1->desc() << p1->price();

  delete p1;
}
```

9.2 Inheritance and copy/move

Consider :

```
Class Book {
  //Defines copy/move
};

Class Text : public Book {
  //Does not define copy/move
};

Text t {"Algoithms". "CLRS", 500. "CS"};
Text tz = t; // No copy Ctor in Text
```

There is no copy constructor so what happens is

- Book's copy constructor is called
- then goes field-by field for the text part

Write your own operations :

```
Text::Text(const Text &other) : Book {other}, topic{other.topic} {}

Text &text:: Operator= (const Text &other) {
  Book::Operator=(other);
  topic = other.topic;
  return *this
}

Text::Text(Text &&other) : Book{std::move(other)}, topic{std::move(other.topic)} {}

Text &Text::operator=(Text &&other) {
  Book::Operator=(std::move(other));
  topic = std::move(other.topic);
  return *this;
}
```

Note : even though other points to an rvalue, it is it self a lvaue. `std::move` forces an lvalue to be treated as a rvalue, so that "move" versions of the operators can be called.

In addition, these custom operators are equivalent to the default operations, so they do not need to be defined.

Now Consider:

```
Text t1 { ... }, t2 {...} ;
Book *p1 = &t1, *p2 = &t2;
*p1 = *p2; // What happens?
```

When assigning a book point to a book pointer, we get a concept called **Partial Assignment**, as it only copies the book portion of the object. To fix this, we can set `operator=` to be virtual .

```
Class Book {
  public :
    virtual Book &operator=(const Book &other) ;
};

Class Text : Public Book {
  ...
  public
    Text &operator=(const Book  &other) override;
};
```

Different return types for virtual functions are ok (as long as you return a subclass reference/pointer), but different parameters must be the same or its not an override/compile and violates is-a. As a result, copying a book to a text in the example above, is technically legal, but can be a program breaking.

Recommendation : All superclasses should be abstract.

Rewrite Book Hierarchy :

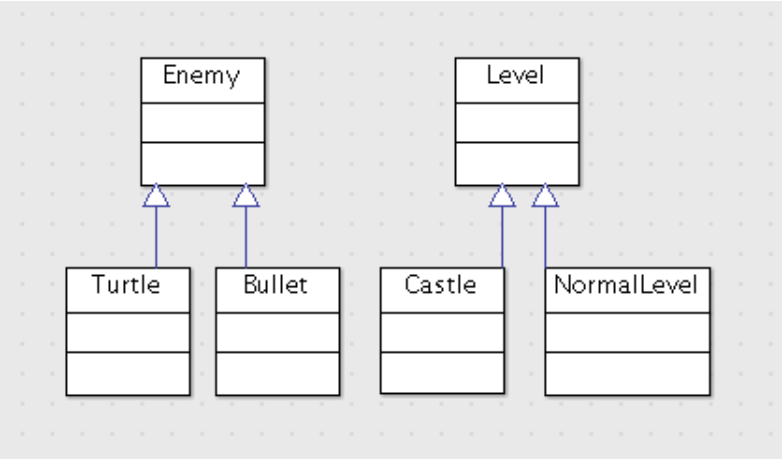
```
Class AbstractBook{
    string title, author;
    int numPages;
  protected:
    AbstractBook &operator=(const AbstractBook &other);
  public :
    ...
};

Class NormalBook : public AbstractBook {
  ...
  public :
    ...
    NormalBook &operator =(const NormalBook &other) {
      AbstractBook::operator(other);
      return *this;
    }
};
```

Now with the partial assignment example, assignment will be prevented for use by the base class. As a result, if partial assignment is done with this implementation, the application will not compile.

9.3 Factory Method Pattern

Factory Method pattern lets a class defer instantiation to subclasses

Example 9.3 *Basic Video Game - A video game is created with 2 kinds of enemies (turtles and bullets), but bullets become more frequent later in the game.*




```
1  Class Level {
2      ...
3    public :
4      virtual Enemy *createEnemy() = 0;
5  };
6
7  Class NormalLevel : pubic Level {
8    public :
9      Enemy *createEnemy () override { /* create turtles */ }
10 };
11
12 Class Castle : public Level {
13   public :
14     Enemy *createEnemy() override {/* mostly bullets */}
15 };
16
17 Level *l = new NormalLevel;
18 Enemy *e = l->createEnemy;
```

9.4 Template Method Pattern

The template method pattern is used in any siutuition when you want subclasses to override superclass behaviour, but some aspects must remain the same.

Example 9.4 *There are red turtles and green turtles.*

```
1  Class Turtle {
2    public :
3      void draw () {
4        drawHead();
5        drawShell();
6        drawFeet();
7      }
8    private :
9      void drawHead();
10     void drawFeet();
11     virtual void drawShell() = 0;
12 };
13
14 Class RedTurtle : public Turtle {
```

```
15      void drawShell() override {/* draw red shell */}
16  };
17
18  Class GreenTurtle : public Turtle {
19      void drawShell () override { /*draw green shell */}
20  };
```

- *Most of the functions are defined an cannot, be overridden, but the functions listed as virtual allow for certain functionality to be modified. In the given example, sub classes can only change how the shell is drawn while everything else is already defined.*

9.4.1 Extension : The Non-Virtual Interface (NVI) Idiom

- A public virtual method is really two things :
  - An interface to the client, which indicates provided behaviour, with pre/post conditions
  - An interface to subclasses, which has a hook to insert specialized behaviour.

It is hard to separate, these ideas if they are tied to the same function.

The NVI idiom says : all public methods should be non-virtual and all virtual methods should be private or atleast protected, with the exception of the destructor.

Example 9.5 *Digital Media*

```
1   // Without NVI
2   Class DigitalMedia {
3     public :
4       virtual void play () = 0 ;
5   };
6
7   //With NVI
8   Class DigitalMedia {
9     pubic :
10      void play() { doPlay(); }
11    private :
12      virtual void doPlay () = 0;
13  };
```

This generalizes template method, by putting every virtual function call inside a template method.

9.5 STL Map

Example 9.6 *"arrays" that map string to int*

```
1   #include <map>
2
3   //Creating a map
4   map<string, int> m;
5
6   //Add Values
7   m["abc"] = 1;
8   m["def"] = 4
```

```
9
10 //Printing
11 cout << m["ghi"]; // if key is not present, it is inserted with a default constructed value.
12 cout << m["abc"]; //1
13
14 //Delete entry
15 m.erase("abc");
16
17 //Check if a key exists
18 if (m.count("abc") ==   1) { ... }
19
20 //Iterating over a map -> sorted in order
21 for (auto &p : m){
22   cout << p.first << " " << p.second << endl; //p's type is std::pair<string, int>
23 }
```