CS 246 Fall 2013 - Tutorial 8

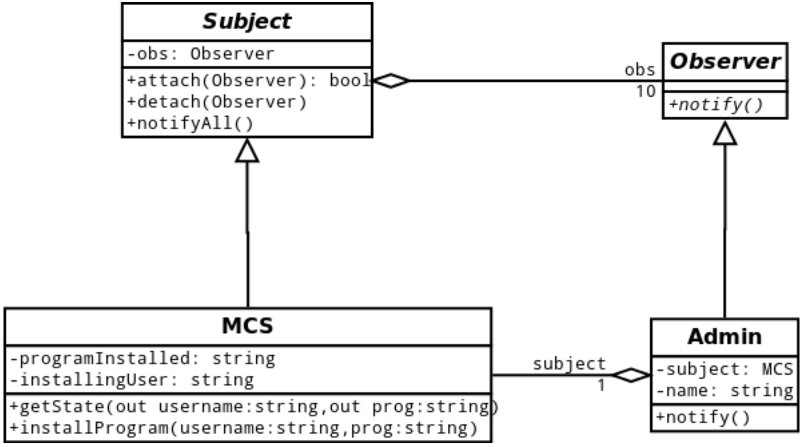
November 12, 2013

1 Summary

- Observer Pattern
- Decorator Pattern
- Make Overiew

2 Observer

- Often we have a desire for a subscription model of information propagation
  - We ask to be notified when something changes (e.g. new article on a website, a race is won, etc)
  - This task is common in web developed and user interfaces
- The **Observer** Pattern models this type of relationship
  - More specifically, it models the idea that there exists a many to one dependency with regards to notification
- Goal: Maintain a list of interested objects and notify them then internal state changes
- Consider the problem of notifying system administrators when a new program is installed

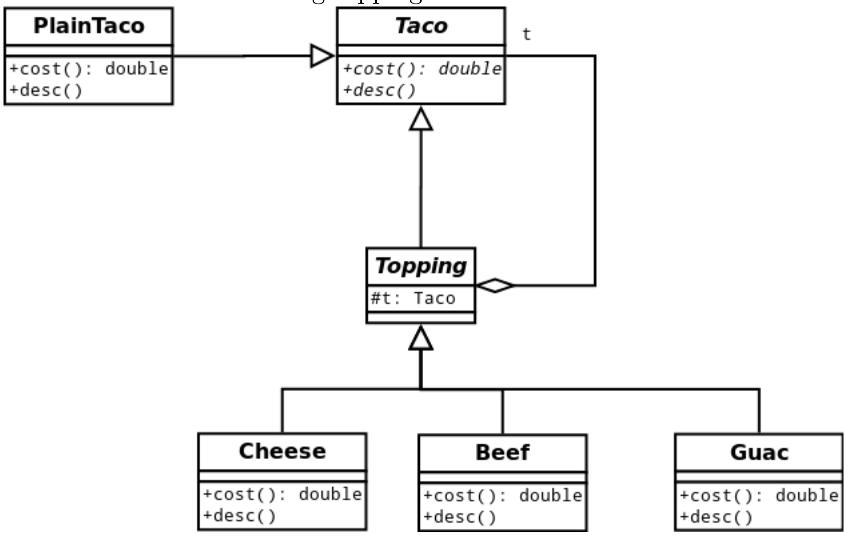


```
#include <iostream>
#include <string>
using namespace std;

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

const int max_obs= 10;
class Subject {
  Observer* obs[max_obs];
  int obs_count;
```

```
 public:
  Subject() : obs_count(0){}
  bool attach(Observer* o){
    if(obs_count != max_obs){
      obs[obs_count++] = o;
      return true;
    } // if
    return false;
  }

  void detach(Observer* o){
    for(int i=0; i < obs_count; ++i){
      if(obs[i] == o){
        for(int j=i; j < obs_count -1; ++j)
          obs[j]=obs[j+1];
          obs[obs_count--]=0;
      } // if
    } // for
  }

  void notifyAll(){
    for(int i=0; i < obs_count; ++i)
      obs[i]->notify();
  }
  virtual ~Subject(){}
};

// MasterControlSystem
class MCS : public Subject{
  string programInstalled;
  string installingUser;
 public:
  MCS(){}

  void getState(string& name, string& prog){
    name = installingUser;
    prog = programInstalled;
  }
  void installProgram(string name, string prog){
    programInstalled = prog;
    installingUser = name;
    notifyAll();
  }
};

class Admin : public Observer{
  MCS* subject;
  string name;
 public:
  Admin(MCS* mcs, string myname):subject(mcs),name(myname){
    subject->attach(this);
  }

  void notify(){
    string iu, ip;
    subject->getState(iu, ip);
    if(iu == "TRON"){
      cout << "TRON fights for the users! Allow " << ip << " to be installed.\n";
    } else if ( iu == "EL" && name=="ASH"){
      cout << "ASH has removed EL’s install permissions. Deny installation." << endl;
  } else {
   cout << name << " allows installation of " << ip << " by " << iu << "\n";
  }
 }

 ~Admin(){
  subject->detach(this);
 }
};
int main(){
 MCS mcs;
 Admin ash (&mcs, "ASH");
 Admin gvc (&mcs, "GVC");
 Admin clu (&mcs, "CLU");
 mcs.installProgram("TRON", "LightCycle");
 mcs.installProgram("BML", "alpine");
 mcs.installProgram("EL", "Quadris");
}
```

3 Decorator

- Suppose we wanted to be able to dynamically add functionality to an object but still retain the original object
  - Power ups in video games
  - Modifications to cars
  - Decorating a room
  - Implementing a user interface
- The **Decorator** pattern takes some relatively simple object and specializes it in some fashion
  - Note: We could just create subclasses for every possibility but that can become tedious if there are many options
- Consider the case of adding toppings to tacos:



```
class Taco {
 public:
  virtual double cost() = 0;
  virtual void desc() = 0;
  virtual ~Taco();
};
class PlainTaco : public Taco{
 public:
  double cost() { return 0.25;}
  void desc() { cout << "a corn flour tortilla";}
```

```
};
class Topping : public Taco{
  protected:
    Taco &t;
  public:
    Topping(Taco &t): t(t){}
};
class Lettuce : public Topping{
  public:
    Lettuce(Taco& t) : Topping(t){}
    double cost() { return t.cost() + 0.50;}
    void desc() { t.desc(); cout << ", lettuce";}
};
class Beef : public Topping{
  public:
    Beef(Taco& t) : Topping(t){}
    double cost() { return t.cost() + 0.50;}
    void desc() { t.desc(); cout << ", ground beef";}
};
class SourCream : public Topping{
  public:
    SourCream(Taco& t) : Topping(t){}
    double cost() { return t.cost() + 0.50;}
    void desc() { t.desc(); cout << ", sour cream";}
};
class Cheese : public Topping{
  public:
    Cheese(Taco& t) : Topping(t){}
    double cost() { return t.cost() + 0.75;}
    void desc() { t.desc(); cout << ", cheese";}
};
class Guac : public Topping{
  public:
    Guac(Taco& t) : Topping(t){}
    double cost() { return t.cost() + 1.00;}
    void desc() { t.desc(); cout << ", guac";}
};
class GreekYogurt : public Topping{
  public:
    GreekYogurt(Taco& t) : Topping(t){}
    double cost() { return t.cost() + 0.75;}
    void desc() { t.desc(); cout << ", greek yogurt";}
};
int main(){
    PlainTaco t;
    Cheese t2 (t);
    Guac t3 (t2);
    Cheese t4 (t3);
    Beef t5 (t4);
    Cheese t6 (t5);
    SourCream dream (t6);

    // Note that reuse of a previous decorator
    Beef t7(t2);
    GreekYogurt t8 (t7);
    Guac healthy (t8);
    cout << "Cost of dream taco: " << dream.cost() << ", which includes: ";
    dream.desc(); cout << endl;
    cout << "Cost of healthy taco: " << healthy.cost()  << ", which includes: ";
    healthy.desc(); cout << endl;
```

```
}
```

- Things to note:
  - Toppings are not strictly tacos but when used in conjunction with a taco, make a new type of taco
  - A decorator (Topping) always points to a “simpler” object and generally evaluates the “simpler“ object at some point (e.g. it relies on some value generated by it’s sub-object)

4 Make

- When our programs encompass many files (e.g. the Taco Decorater example requires many files) it becomes tedious and sometimes hard to manage compilation by hand
- Especially when there are intricate dependencies between files
- We use the `make` tool to assist us in the compilation of complex programs
- However, `make` is relatively simple and doesn’t understand complex dependencies
- But our compiler (g++) can help!
  - `-MMD` generates dependencies for user files
  - `-MD` generates dependencies for system and user files. Typically, we don’t use this as system files don’t change much.
- Both options output dependencies to a .d file for each .cc/.cpp file
  - For example: plaintaco.d contains plaintaco.o: plaintaco.cpp plaintaco.h taco.h
- Let’s consider a makefile for our Taco Decorator example

```
CXX = g++
CXXFLAGS = -Wall -MMD
OBJECTS = main.o plaintaco.o tacodecorator.o lettuce.o greekyogurt.o guac.o beef.o cheese.o sourcream.o
DEPENDS = ${OBJECTS:.o=.d}
EXEC = tacos

${EXEC} : ${OBJECTS}
    ${CXX} ${OBJECTS} -o ${EXEC}

-include ${DEPENDS}

PHONY : clean
clean :
    rm -rf ${DEPENDS} ${OBJECTS} ${EXEC}
```

- Recall that CXX and CXXFLAGS are special variables that make understands
- Remember commands must always be indented 1 tab character (not spaces)
- .PHONY says that the target isn’t real, we just want to run some commands
- -include will include the contents of the given files (like #include in C/C++)