CS 246 Winter 2014 - Tutorial 8

November 12, 2014

1 Summary

- GDB
- Observer Pattern
- Decorator Pattern

2 GDB

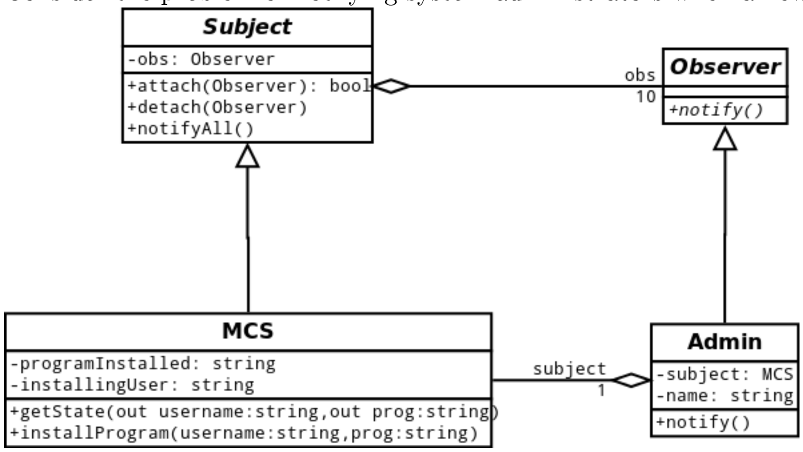
- As you've noticed from writing increasingly more complex programs, errors start to crop up all over the place
- Sometimes these errors are easy to identify and somtimes they are hard
- There are a variety of ways to try to find errors
  - A common debugging tool is the print statement
  - Throwing a bunch of print statments into your code that print out variable values can often find the problem
  - ...But not always
- Other times we need a tool that allows us to step through the execution of a program
- In first year, you might have used DrRacket's stepper.
- `gdb` is something like that for C/C++
- `gdb` allows you to print variables, set variables, watch variables, set breakpoints, step through execution, etc
- To use `gdb`, we need to compile our program with the `-g` option which provides debugging information
  - For example, it keeps variable and function names, line numbers, etc
- Some common commands include:

Command	Description
run [args]	run the program until it crashes or completes
backtrace|bt	print trace of current stack (list of called routines)
print var-name	print value of specified variable
break routine|[filename:]line-no	set breakpoint at routine or line of file
step [n]	execute next n lines (into routines)
continue [n]	skip next n breakpoints
watch var-name	print a message every time var-name is changed
quit	exit gdb

- By default, run will run the program until completion or a crash. So it is wise to set breakpoints before you begin.
- See `gdbex1.cpp`, and `gdbex2.cpp` for examples of buggy programs.

3 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 development and user interfaces (see `Model-View-Controller (MVC)`, a related pattern)
- 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 when internal state changes
- Consider the problem of notifying system administrators when a new program is installed




```cpp
#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(){
```

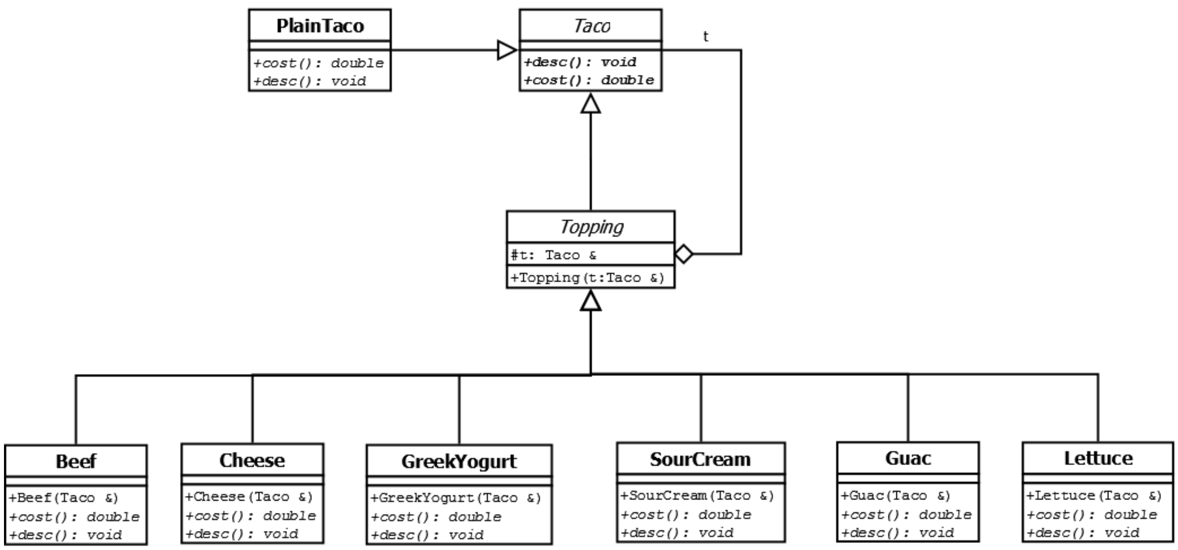
```cpp
    for(int i=0; i < obs_count; ++i)
      obs[i]->notify();
  }
  virtual ~Subject(){}
};

// 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");
}
```

4 Decorator

- Suppose we wanted to be able to dynamically add functionality to an object but still retain the original object
- For example:
  - 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
- We take this simple object and make it more complex by adding functionality over time
- Note that:
  - We could just create subclasses for every possibility but that can become tedious if there are many options
  - But this may also be impossible as there could be an infinite number of combinations
- Consider the case of adding toppings to tacos:



- Things to note:
  - Toppings are not strictly a Taco but when used in conjunction with a simple Taco instance, 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)
  - The result of the evaluation may be used by the Decorator in creation of it's own result or just pass it along
    * We use the the result when we compute a Taco's cost
    * We pass it along when we print the toppings of a Taco
  - Because we want to allow different tacos to share instances, the Decorator does not take ownership of the decorated object
  - Compare this to the Coffee Decorate example from class.
    * The main takeaway is to choose the approach that makes the most sense for your design.