CS 246 Fall 2013 - Tutorial 9

November 15, 2013

1 Summary

- GDB
- Visitor Pattern
- Coupling and Cohesion

2 GDB

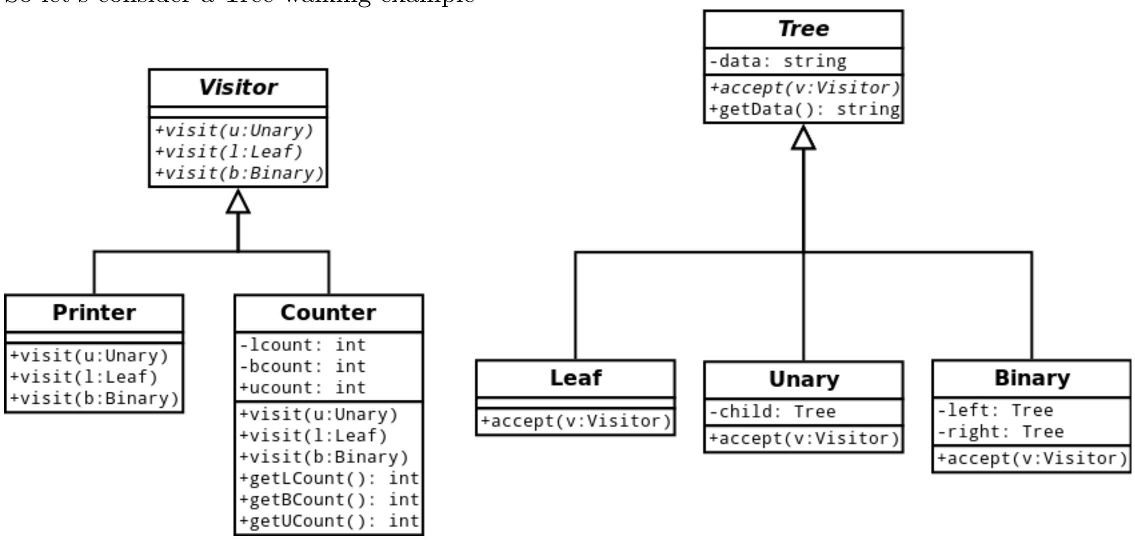
- As we begin to write increasingly complex programs, errors start to crop up
- Sometimes these errors are easy to identify and somtimes they are hard
- There are a variety of ways to try to find errors
  - A common debugging tool is the print statement
  - Throwing a bunch of print statments into your 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 had 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 `gdbex0.cpp`, `gdbex1.cpp`, and `gdbex2.cpp` for examples of buggy programs.

3 Visitor Pattern

- The Visitor Pattern is used when we want to implement/simulate double dispatch
  - That is, we want to use the runtime type of two objects to determine what to do
- The Visitor Pattern combines overloading and overriding methods
- Visitor Pattern allows us to add functionality to existing classes without requiring extensive changes or recompilation
  - This is because our Elements only need to know about the abstract Visitor base class and not any concrete classes
- So let's consider a Tree walking example




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

class Leaf;
class Binary;
class Unary;

class Visitor{
  public:
    virtual void visit(Unary &) = 0;
    virtual void visit(Binary &) = 0;
    virtual void visit(Leaf &) = 0;
    virtual ~Visitor() = 0;
};

Visitor::~Visitor(){}

class Tree{
  protected:
    string data;
  public:
    Tree(string data) : data(data){}
    virtual void accept(Visitor& v) = 0;
    string getData(){return data;}
    virtual ~Tree() = 0;
};
Tree::~Tree(){}

class Leaf : public Tree{
  public:
```

```
    void accept(Visitor &v){
      v.visit(*this);
    }
    Leaf(string data) : Tree(data){};
    ~Leaf(){}
};

class Unary : public Tree{
  protected:
    Tree * child;
  public:
    void accept(Visitor& v){
      child->accept(v);
      v.visit(*this);
    }

    Unary(string data, Tree *t): Tree(data), child(t){}
    ~Unary(){delete child;}
};

class Binary : public Tree{
  protected:
    Tree * left, * right;
  public:
    void accept(Visitor& v){
      left->accept(v);
      v.visit(*this);
      right->accept(v);
    }

    Binary(string data, Tree * t1, Tree *t2):Tree(data), left(t1), right(t2){};
    ~Binary(){delete left; delete right;}
};

class Counter: public Visitor{
    unsigned int lcount, ucount, bcount;
  public:
    void visit(Leaf& l){
      lcount+=1;
    }
    void visit(Unary& u){
      ucount+=1;
    }
    void visit(Binary& b){
      bcount+=1;
    }

    Counter():lcount(0), bcount(0), ucount(0){}
    int getLCount(){return lcount;}
    int getUCount(){return ucount;}
    int getBCount(){return bcount;}
    ~Counter(){}
};

class Printer: public Visitor{
  public:
    void visit(Leaf& l){
      cout << "Leaf : " << l.getData() << endl;
    }
    void visit(Unary& u){
```

```
        cout << "Unary: " << u.getData() << endl;
    }
    void visit(Binary& b){
        cout << "Binary: " << b.getData() << endl;
    }
    ~Printer(){}
};

int main(){
    Tree * tp = new Unary("foo", new Binary("bar", new Unary("baz", new Leaf("taco")), new Binary("bat", r
    Counter c;
    tp->accept(c);
    cout << c.getLCount() << endl;
    cout << c.getUCount() << endl;
    cout << c.getBCount() << endl;
    Printer p;
    tp->accept(p);
}
```

- Note the forward declaration of Unary, Binary, and Leaf, which is requried by Visitor to compile.

4 Coupling and Cohesion

- At times we need to compare possible design choices beyond efficiency/memory concerns
- To do this we need some kind of new metric
- Two of the most common are **coupling** and **cohesion**

4.1 Coupling

- **Coupling** measures the degree of interdependence among programming "modules" (e.g. classes, libraries, functions)
- The aim is to achieve the lowest coupling (equivalently the highest independence)
- We want it to be the case that any change to a particular module should minimze recompilation and changes necessary in other modules
- When we program to an interface, we exhibit low coupling
- When we program to an implementation, we exhibit high coupling
- For example. C++ strings are not really char* underneath. There is an underlying layer that we are not exposed to and don't need to be.

4.2 Cohesion

- **Cohesion** measures the degree of association among elements within a module
  - These elements could singular statements, groups of statements, etc
- A highly cohesive module has strongly and genuinely related elements
- Typically (but not always), low cohesion implies high coupling
- Similarly, high cohesion implies low coupling
- The C++ <algorithm> library has low cohesion (it's just a bunch of unrelated algorithms)
- The C++ <string> library has relatively high cohesion (all the C++ string stuff is contained here)

4.3 Coupling and Cohesion

- Ultimately, we have that:
  - **Low coupling** is a sign of good structure and design
  - **High cohesion** supports readability and maintainability
- Accordingly, we try to pick design that best exhibit **low coupling AND high cohesion**
- However, this cannot always be done (sometimes efficiency may rule out good design)