CS 246 Fall 2018 — Tutorial 10

November 21, 2018

Summary

1 Measures of Design Quality

- There are two commonly accepted measures of design quality: **cohesion** and **coupling**.
- **Coupling** measures the degree of interdependency among "modules" (e.g. classes, libraries, functions).
  - The aim is to achieve low coupling (i.e. the modules are as independent as possible).
  - Any changes made to a particular module should minimize recompilation and changes necessary in other modules.
  - When we program according to an interface, we exhibit low coupling.
  - When we program according to an implementation, we exhibit high coupling.
- **Cohesion** measures the degree of association among elements within a module.
  - The aim is to achieve high cohesion, a highly cohesive module has strongly and genuinely related elements.
  - The C++ `<utility>` library has low cohesion (it's just a library for unrelated functionality that doesn't fit elsewhere).
  - The C++ `<string>` library has relatively high cohesion, because the only purpose of this library is to introduce the `string` class and related functions.
- In general, we would like our design to exhibit **high cohesion** and **low coupling**. However, this cannot always be done.

  For example, in some rare cases (i.e. not in this course) you need to achieve maximum efficiency by trading in good design.

2 Visitor Pattern

- Earlier in the course we saw *dynamic dispatch*, which allows selecting the intended method to call in a class hierarchy based on the runtime type of the object.
- What if we want to select the correct method to call based on two different hierarchies?
- Consider a Tree with nodes of various degrees:

```
class TreeVisitor {
    ...
public:
    virtual int visit(const UnaryTreeNode &t) = 0;
    virtual int visit(const BinaryTreeNode &t) = 0;
};
class PrettyPrinter : public TreeVisitor {
    ...
public:
    int visit(const UnaryTreeNode &t) override {...}
    int visit(const BinaryTreeNode &t) override {...}
};
class NodeCounter : public TreeVisitor {
    ...
public:
    int visit(const UnaryTreeNode &t) override {...}
    int visit(const BinaryTreeNode &t) override {...}
};
```

- However this solution does not take polymorphism into account: what if we have a `TreeNode` reference?
- To solve this problem, we have to make use of dynamic dispatch in both hierarchies. This technique is called **double dispatch**.

```
// Complete example can be found in 10/visitor
class TreeVisitor {
    ...
public:
    virtual int visit(const UnaryTreeNode &t) = 0;
    virtual int visit(const BinaryTreeNode &t) = 0;
};
class PrettyPrinter : public TreeVisitor {
    ...
public:
    int visit(const UnaryTreeNode &t) override {...}
    int visit(const BinaryTreeNode &t) override {...}
};
```

```
class NodeCounter : public TreeVisitor {
    ...
public:
    int visit(const UnaryTreeNode &t) override {...}
    int visit(const BinaryTreeNode &t) override {...}
};

class TreeNode {
    int data;
public:
    virtual int accept(TreeVisitor &v) = 0;
};
class UnaryTreeNode : public TreeNode {
    TreeNode *child;
public:
    ...
    int accept(TreeVisitor &v) override {
        return v.visit(*this);
    }
};
class BinaryTreeNode : public TreeNode {
    TreeNode *left;
    TreeNode *right;
public:
    ...
    int accept(TreeVisitor &v) override {
        return v.visit(*this);
    }
};
```

- The client code calls `t.accept(v)` where `t` is a reference to a `TreeNode` and `v` is a reference to a `TreeVisitor`.
- In `accept()` methods, the call to the virtual method `v.visit()` makes sure that `v` is the correct visitor.
- It seems like `UnaryTreeNode::accept()` and `BinaryTreeNode::accept()` have the same code; however, the types of `*this` are different in those methods. This will make sure the compiler select the correct overloaded method to call.
- Adding a new visitor does not require any implementation change for the `TreeNode` hierarchy. This is called the **visitor pattern**.
- This solution promotes low coupling and high cohesion. How?

3 Model-View-Controller (MVC)

In class, we discussed the single responsibility principle which states that each class should have one job. Keeping this in mind, consider how we should implement an application which interacts with a user. Logically, if the program interacts with a person, it must be able to accept input and display information to the user. Additionally, aside from input and output, there must be some interpretation of what the input does and how that effects the display. Thus, we've come up with three responsibilities: input, output, and logic.

Model-View-Controller (MVC) is a design pattern which explains how these responsibilities should be divided amongst classes and how these classes should interact. Specifically, we should separate these jobs into classes:

- Model: The "logic" behind the application. This stores the state of the application and logic of how interactions should be handled.
- View: The "ouput"/user interface. Whatever the person using the application "sees". This could be a terminal, a GUI, sound, etc.
- Controller: The "input". How the user interacts with the application. This could be any input object: clicking with a mouse, typing on a keyboard, a touch screen, a joy stick, voice control, etc.

Each of these classes should be responsible for one thing: the model is the maintainer of logic and state, the view presents data to the user, and the controller receives input from the user.

Consider how information would flow between these classes:

1. The model will receive input from the controller.
2. Determine if the input logically makes sense given the current state and update the state.
3. Notify the view to update to represent the change of state.
4. Repeat.

Note that step 3 of the flow was to "notify" the view. The relationship between the model and the view is typically implemented using an observer pattern: when something happens which results in the state of the model changing, the view is notified to reflect this change. Note that we could have easily have multiple views being present at the same time showing data in different ways to interpret the data.

In theory, this works great and it works well when interacting doing simple tasks such as interacting via a command line: we can setup the controller to read from std::cin and the view to be storing and printing some visual representation to std::cout which results in low coupling.

However, imagine that the input is received via the user pressing buttons on a window. We can imagine in this situation the controller and the view are heavily coupled: the controller will need to be updated to know which buttons exist in the window.

Example: "Maze"

4 NCurses

NCurses is a C library which allows the programmer to "print" dynamically to the terminal. While we won't be showing you how to interact with ncurses, the final example of the maze used ncurses and you must use it (or a similar tool) in your final project. It allowed for input to be read and printed without needing to use cin or cout.

There are many webpages you can find to teach you how to program using ncurses. I used `http://www.cs.ukzn.ac.za/~hughm/os/notes/ncurses.html#init` and `http://tldp.org/HOWTO/NCURSES-Programming-HOWTO/index.html`. While both are a bit dated in appearance, they are good.