CS 246 Fall 2016

November 23, 2016

1 Summary

- pImpl Idiom
- Bridge Pattern
- Strategy Pattern
- Model-View-Controller

2 Tip of the Week:

-

3 pImpl Idiom

- When programming, we want to minimize compilation dependency as much as possible. Consider this code:

```
class Xwindow{
  Display *d;
Window w;
  int s;
  GC gc;
  unsigned long colours[11];

   public:
XWindow(int width, int height);
    ...
};
```

- Consider:
  - The user doesn't really care about any of the fields. They are all important to interacting with the window but complicate the class and show more of the system to the user than we might want them to have.
  - What if the interface for the window changes tomorrow? We will have to recompile all our code which depends on the Xwindow class despite not changing the Xwindow class itself.
- Instead, we will create a pointer to the window implementation and access the fields of the window through that pointer. Our code will look like:

```
// in xwindow.h
class WindowImpl;

class Xwindow{
  WindowImpl* imp;
   public:
```

```
XWindow(int width, int height);
  ...
};

// in window.h
struct WindowImpl{
  Display *d;
Window w;
  int s;
  GC gc;
  unsigned long colours[11];

   public:
WindowImpl(int width, int height);
};
```

- Now if we need to change the implementation of Window, we will add them in the windowImpl class. Only the files which include window.h (which should only be xwindow.cc and window.cc) will need to be recompile. This can reduce compile time a lot.
- The XWindow class will now work as a wrapper class to our implementation. It will be the class the client interacts with.
- Note: the pImpl idiom should not always be used. It should be used in situations as seen here where changing the fields should not change the interface for the class.

4 Bridge Pattern

- Now that we have discussed the pImpl idiom, what if we wanted to use a different pImpl which can be choosen at runtime?
- We can use the pImpl as an abstract class and have multiple implementations which inherit from it. These implementations will all have the same fields to be accessed (as they inherit from the same pImpl) but the values of the fields will differ which gives a different implementation.
- For example, perhaps we want to have multiple sizes of a graphics display (e.g. small, large, fullscreen), we can have different implementations which we can switch at will.

5 Strategy Pattern

- Now suppose we wanted to make a simple guessing game where a number between 1 and 100 is randomly picked. We want to make it so a human can guess, a dumb AI can guess and a smart AI can guess.
- What we'll do is make an abstract class which all of our guessers will inherit from. This class will have a pure virtual method, guess(), which returns the guess from the guessers. See code.
- This is the strategy pattern. The idea is that we will put the strategy used in these classes that can be replaced at runtime.

You may ask what is the difference between the bridge pattern and the strategy pattern. If you were to draw out their UMLs, they would look (almost) identical. The only real difference is the intent of each pattern. The bridge pattern is used when we want to separate the implementation (things like fields and accessor methods). The strategy pattern is used when we want to be able to have multiple ways of doing the same operation. Note that each pattern encapsulates the information we want to be able to change in our subclasses. This is common theme amoung design patterns.

6 Model-View-Controller

- Model-View-Controller (MVC) is a design pattern used when implementing user interfaces. It is made up of three parts:
- Model: the representation of the logic of the system, e.g. rules, data.
- View: the representation of the game seen by the user, e.g. text display, GUI.
- Controller: how the user interacts with the system, e.g. input through stdin, pushing buttons, etc.
- The model and view will often have a Subject-Observer relationship, where the model notifies the view when it get information from the controller. Sometimes they will not communicate.
- How the controller will be used will often be tied to the view. This may come in the form of buttons which are part of the view if it's graphical or stdin if the view is textual. That being said, it's possible to have a graphical display receiving input from stdin. The controller will pass the information it receives from the user to the model which will update the view.
- To allow the view and controller to be picked at run-time, often these will be implemented using the strategy pattern.

7 Resource Acquisition is Initialization (RAII)

- RAII is vital to writing exception-safe code in C++
- RAII relies on the guarantee that when an exception is thrown, destructors for stack-allocated objects will be called
- Resources are acquired during initialization (e.g. in a constructor), so that they cannot be used before they are available, and are released when the owning object is destroyed
- Making use of RAII also more easily facilitates implementing the various levels of exception safety
- There are three guarantees that you can provide with respect to exception safety:
  - **basic** guarantee: if an exception is thrown, data will be in a valid state but may not make sense
    - e.g., if we change variables in an assignment operator before allocating heap memory
  - **strong** guarantee: if an exception is thrown, the data will appear as if nothing happened
    - The copy-and-swap idiom provides the strong guarantee
  - **nothrow** guarantee: an exception is never thrown
    - Swapping two pointers using std::swap is guaranteed not to throw an exception
- Pointers and dynamic memory pose a problem when trying to implement exception safety in particular.
- The pointer is deallocated but the memory (possibly an enormous object) is not.
- The original C++ solution was the `auto_ptr` is a templated type that behaves exactly like a pointer, except that when it is destroyed it calls `delete` on its pointed to memory. It is now **deprecated** and you should **not** use it.
- In C++11, `auto_ptr` was deprecated and replaced by `shared_ptr`, `unique_ptr`, and `weak_ptr`[1].
  - `shared_ptr` allows many pointers to the same block of memory and only deletes that memory when no other `shared_ptr`s point to it
  - `unique_ptr` is similar to `auto_ptr` but supports more functionality (e.g. operator[])
  - `weak_ptr` is like `shared_ptr` but doesn't count towards the "shared count;" it is used to prevent cyclic ownership in `shared_ptr`s

[1] Not discussed in class and not required knowledge.