Tute 8: Design Patterns

Review:

What are some commonly used design patterns?

Design Patterns are a set of commonly used techniques for solving commonly encountered problems, some of which can be trivial, and some of which can be non-trivial. Design Patterns can be divided into four main groups:

- Creational
- Structural
- Behavioural
- Concurrency

Commonly used Design Patterns include:

- Factory
- Prototype
- Resource Acquisition
- Singleton
- Adapter
- Observer
- MVC
- Iterator
- Visitor
- Strategy
- Lock
- Thread pool

Many Design Patterns are similar and often have similar ideas. Likewise many are easy to use in conjunction with other patterns (ie, it is easy to combine a Strategy with a Factory, and easy to combine those with a Prototype).

Why are creational patterns popular?

Creational patterns are popular, because at some point you need to create objects, and it is not always known exactly what objects need to be created at compile time. This means that calling the constructor of a class (code that can only be written at compile time) needs to be delegated in some way. This can be as simple as an if statement:

```
Object *createObject(const std::string &type)
{
```

```
        if (type == “A”)
                return new A();
        else if (type == “B”)
                return new B();
        …
}
```

Although this is not strictly a design pattern, a very basic factory could be implemented in this way. Another implementation could use a map of creational functions or prototype objects.

Programming Concepts:

What is a factory pattern?

A factory pattern is an object that can generate the type of object you need where you only need to provide some of the details and the factory will figure the rest out so you don’t have to.

What is the singleton design pattern? How do we implement this in modern C++?

See the files provided here:
https://github.com/muaddib1971/cpt323/tree/master/chat_examples/week8

What is the MVC design pattern? Can this only be useful in gui programming?

See the discussion of MVC had in week 7 of the course. The important aspect of mvc is that there are three major components: the model (the representation of the “state” of the application), the view (what the end user “sees”) and the controller (the event handler). All three components should be designed to reduce coupling so that with less work we can replace each of these components with a different implementation and it will be transparent to the other components. For example, moving from a database to a text file based storage should only impact upon the files in the model.

How is a prototype used in conjunction with a factory?

Prototypes can be used in conjunction with factories by storing empty prototype objects, which are then used to create concrete objects at runtime, eg:

```
class Prototype
{
public:
        virtual Prototype *copy() = 0;
};
```

```
class A : public Prototype()
{
public:
	virtual Prototype *copy() { return new A(); }
};

class B : public Prototype()
{
public:
	virtual Prototype *copy() { return new B(); }
};

class Factory
{
private:
	static std::map<std::string, Prototype*> prototypes;
public:
	static void initTypes()
	{
		prototypes[“A”] = new A();
		prototypes[“B”] = new B();
	}

	static Prototype *createObject(const std::string &type)
	{
		return prototypes[“A”]->copy();
	}
};
```

How can this be used with a strategy pattern?

The easiest way to implement the Strategy Pattern is to create an abstract strategy class, and have the different strategies inherit from it. Then the controller has a method to choose which strategy to use, as well as a method to use it:

```
class Strategy
{
public:
	virtual void doSomething() = 0;
};

class StratOne : public Strategy
{
```

```
public:
	virtual void doSomething() { std::cout << “StratOne\n”; }
};

class StratTwo : public Strategy
{
public:
	virtual void doSomething() { std::cout << “StratTwo\n”; }
};

class SomeClass
{
private:
	Prototype *strat;
public:
	void selectStrategy(const std::string &type)
	{
		strat = Factory::createObject(type);
	}

	void doSomething()
	{
		start->doSomething();
	}
};
```

Compilation: None this week.

Errors: None this week.

Debugging: Comments

What is a comment/what is it used for?

What makes a good comment is subject to dispute. The two main types of comment are: one that describes what a piece of code is doing, or one that describes why code has been written that way. Either is acceptable, although providing comments to explain what the code is doing should be reserved for code that is complex and difficult to read, or for documentation purposes

(ie, using comments alongside documentation generators), otherwise if the code is simple and its function can be easily determined by reading it, then commenting it is superfluous and should be avoided. Comments describing why code is behaving the way it is are particularly useful for working with code you haven't used for a long time, or for working with other people's code (especially if the choices you've made for doing something a particular way are non-obvious). All functions in your program should have a header comment. Put the effort into writing good header comments and summarising complex blocks of code rather than commenting every line.

How do we make code more readable?

It is always a good idea to make code as readable as possible. Ideally code should be written so that its functionality can be determined by reading it without the help of comments (this is not always possible, but it is a good target to aim for). To make code as readable as possible, a single style and naming convention should be chosen and always used (K&R is the usual style used for C++/C programs, but not always). Ideally code should also be written to keep as few indentation levels as possible (as a good rule of thumb, if you have more than 3 or 4 indentation levels, you should consider writing another function and offloading some work to that). Code should also be written so that functions have a specific purpose and don't go over multiple pages (if you have a function which is very long, or is doing multiple tasks, consider splitting it up into different functions).

How do we make code more modular?

Modularity means programming so that your code is divided into separate, potentially interchangeable components (this includes both classes and methods). A good rule of thumb when dealing with modularity is to ensure that code (objects and functions) only know as much as they need to about other sections of code, and no more. One way of ensuring this is through encapsulation (using private variables) however a class with a lot of accessors and mutators is not as modular as one without, so the number of these should also be kept as low as possible. This also means that information that is needed only by one function should only be available in that one function (this is not always possible, but it is a good target to aim for). This can be a big benefit when making future changes to code or debugging, since a single feature won't require you to make changes in many different places.

Exercises: None this week