Design Patterns

- Software design patterns are abstractions that help structure system designs.
- Design patterns gathered some traction in programming due to the publication of "Design Patterns: Elements of Reusable Object-Oriented Software" book in October 1994 by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, often referred to as the "Gang of Four" (GoF), that identifies and describes 23 classic software design patterns.
- A design pattern is neither a static solution, nor is it an algorithm. It is a way to describe and address by name, a repeatable solution or approach to a common design problem, that is, a common way to solve a generic problem.
- Patterns can emerge on their own or by design. This is why design patterns are useful as an abstraction over the implementation, and help at then design stage.
- With the concept named, an easier way to facilitate communication over a design choice is given so that every person can share the design concept.

Abstract Factory

- A Factory is a utility class that creates an instance of a class from a family of derived classes.
- The Factory Design Pattern is useful in a situation that requires the creation of many different types of objects, all derived from a common base type.
- Typically a *Factory* pattern is used to facilitate creating instances of concrete Classes that derive from an *abstract* base class. Such *Factory* patterns are known as an *Abstract Factory*.
- The *Abstract Factory* Method defines a method for creating the objects, which subclasses can then override to specify the derived type that will be created. Thus, at run time, the Factory Method can be passed a description of a desired object (e.g., a string read from user input) and return a base class pointer to a new instance of that object.
- The pattern works best when a well-designed interface is used for the base class, so there is no need to cast the returned object.
- Take for example the following `Computer` abstract base class and extending `Laptop` and `Desktop` classes.

```
class Computer
{
public:
    virtual void run() = 0;
    virtual void stop() = 0;
    virtual void print() = 0;

    // without this, you do not call Laptop or Desktop destructor in
this example!
    virtual ~Computer() {};
};

class Laptop: public Computer
{
public:
    void run() override   { mHibernating = false; };
    void stop() override  { mHibernating = true; };
```

```
    void print() override { cout << "Laptop hibernating: " <<
mHibernating << endl; };

    // because we have virtual functions, we need virtual destructor
    virtual ~Laptop() {};
private:
    bool mHibernating; // Whether or not the machine is hibernating
};

class Desktop: public Computer
{
public:
    void run() override   { mOn = true; };
    void stop() override  { mOn = false; };
    void print() override { cout << "Desktop on: " << mOn << endl; };
    virtual ~Desktop() {};
private:
    bool mOn; // Whether or not the machine has been turned on
};
```

- A `ComputerFactory` static class might be added to return a `Computer*` instance given a description, e.g.,

```
class ComputerFactory
{
public:
    static Computer* makeNewCompter(char type)
    {
        switch (type)
        {
            case 'l':
                return new Laptop();
            case 'd':
                return new Desktop();
        }
        return NULL;
    }
};
```

- The advantage of this is that the ComputerFactory class is the only one that requires knowledge of the derived classes.
- If a change is made to any derived class of Computer, or a new Computer subtype added, the implementation file for ComputerFactory is the only file that need be recompiled.
- Everyone who uses the factory need only care about the interface for ComputerFactory, which should remain consistent throughout the life of the application.
- Most importantly, if there is a need to add a type, any code calling the factory need not change to support the additional type. The factory can handle new types entirely.