Design Patterns

Madiba Hudson-Quansah

Contents

Chapter 1

Introduction

Definition 1.0.1: Design Pattern

A design pattern describes a recurring problem in the construction phase and, then describes the core of the solution to the problem in a way that promotes re-usability and modularity in any context.

Design Patterns capture the best practices of experienced object oriented software developers, and are general solutions to software development problems. Each pattern is comprised of four parts:

The pattern name - A handle to describe a design problem, its solution and consequences succinctly.

The problem - Describes when to apply the pattern.

- Explains the problem and its context.
- It might describe specific design problems such as how to represent algorithms as objects.
- It might describe class or object structures that are symptomatic of an inflexible design.
- May include a list of conditions that must be met before it makes sense to apply the pattern.

The solution - Describes the elements that make up the design, their relationships, responsibilities and collaborations.

- Does not describe a particular concrete design or implementation.
- Provides an abstract description of a design problem and how a general arrangement of elements solves it.

The consequences - Describes the results and trade-offs of applying the pattern.

- Space and time trade-offs.
- Language and implementation issues.
- Impact on system flexibility, extensibility or portability.

Chapter 2

Types of Design Patterns

Erich Gamma, Richard Helm, Ralph Johnson and John Vlisides in their Design Patterns book define 23 design patterns divided into three types:

Creational Patterns - Patterns that create objects for you, rather than having you instantiate objects directly.

Structural Patterns - Patterns that compose groups of objects into larger structures.

Behavioral Patterns - Patterns that define how objects interact.

2.1 Creational Patterns

Here are the creational patterns:

Factory Pattern - A decision making class that returns one of several possible subclasses of an abstract base class depending on the data provided to it.

Abstract Factory Pattern - Provides and interface to create and return one of several families of related objects.

Builder Pattern - A class that constructs a complex object step by step.

Prototype Pattern - A class that creates a new object by copying an existing object rather than creating new instances.

Singleton Pattern - A class of which only a single instance can exist.

2.1.1 Factory Pattern

Listing 2.1: Factory Pattern

```
#pragma once
#include <cstdint>
#include <memory>

using namespace std;

enum class ShapeType : uint8_t {
    RECTANGLE,
    TRIANGLE,
};

struct Shape {
    virtual ShapeType getShapeName() const = 0;
    virtual float getArea() const = 0;
    virtual ~Shape() = default;
```

```
};

struct Rectangle : public Shape {
    Rectangle(float l, float b) : Shape{}, l{l}, b{b} {}

    ShapeType getShapeName() const override { return ShapeType::RECTANGLE; }

    float getArea() const override { return l * b; }

    ~Rectangle() override = default;

    float l;
    float b;
};

struct Triangle : public Shape {
    Triangle(float h, float b) : Shape{}, h{h}, b{b} {}

    ShapeType getShapeName() const override { return ShapeType::TRIANGLE; }

    float getArea() const override { return 1 / 2.F * b * h; }

    ~Triangle() override = default;

    float h;
    float b;
};

/**
 * @breif Returns one of several possible subclasses of an abstract base class
 * depending on the data provided
 *
 */
struct ShapeFactory {
    unique_ptr<Shape> getShape(ShapeType type, float x, float y) {
        switch (type) {
            case ShapeType::RECTANGLE:
                return make_unique<Rectangle>(Rectangle{x, y});
            case ShapeType::TRIANGLE:
                return make_unique<Triangle>(Triangle{x, y});
        }
    }
};
```

This `ShapeFactory` class is a factory class that returns a subclass of the `Shape` class depending on the data provided to it. The `Shape` class is an abstract class that has a pure virtual function `getArea()`.

The Factory Pattern should be used when:

- A class can't anticipate the class of objects it must create.
- A class uses it's subclasses to specify the objects it creates.
- You want to abstract away the knowledge of the concrete classes from the client.

2.1.2 Abstract Factory Pattern

The Abstract Factory Pattern sits one level of abstraction higher than the Factory Pattern. The Abstract Factory Pattern is a factory object that returns one of several factories, i.e. A factory of factories. An example of this could be for a GUI library that has a factory for elements of a certain themes, e.g. a Windows theme or a Mac theme.

Listing 2.2: Abstract Factory Pattern

```
#pragma once
#include <cstdint>
#include <format>
#include <memory>

using namespace std;

enum class ArmyType : uint8_t { NORSE, ROMAN, MONGOL };

struct Unit {
    Unit(const string& name, int hp) : name{name}, hp{hp} {}

    constexpr operator string() { return format("{} - {}", name, hp); }

    string name;
    int hp;
};

struct Army {
    virtual Unit createInfantry() = 0;
    virtual Unit createRanged() = 0;
    virtual Unit createMounted() = 0;
    virtual ~Army() = default;
};

struct NorseArmy : public Army {
    Unit createInfantry() override { return Unit{"Dregnir", 10}; }

    Unit createRanged() override { return Unit{"Bogmadur", 5}; }

    Unit createMounted() override { return Unit{"Hirdman", 15}; }

    ~NorseArmy() override = default;
};

struct RomanArmy : public Army {
    Unit createInfantry() override { return Unit{"Legionary", 10}; }

    Unit createRanged() override { return Unit{"Velite", 5}; }

    Unit createMounted() override { return Unit{"Equites", 15}; }

    ~RomanArmy() override = default;
};

struct MongolArmy : public Army {
    Unit createInfantry() override { return Unit{"Steppe Vanguard", 10}; }

    Unit createRanged() override { return Unit{"Mangudai", 5}; }
```

```
    Unit createMounted() override { return Unit{"Keshik", 15}; }

    ~MongolArmy() override = default;
};

class ArmyMaker {
   public:
    unique_ptr<Army> getArmy(ArmyType type = ArmyType::NORSE) {
        switch (type) {
            case ArmyType::NORSE:
                return make_unique<NorseArmy>();
                break;
            case ArmyType::ROMAN:
                return make_unique<RomanArmy>();
                break;
            case ArmyType::MONGOL:
                return make_unique<MongolArmy>();
                break;
        }
    }
};
```

This `ArmyMaker` class is an abstract factory class that returns a subclass of the `Army` class depending on the data provided to it. The `Army` class is an abstract class that has the pure virtual functions `createInfantry()`, `createRanged()` and `createMounted()` that return instances of the `Unit` class.

The Abstract Factory Pattern isolates the creation of objects from the client and because of this it is easier to change or interchange the product class families freely. The Abstract Factory Pattern should be used when:

- A system should be independent of how its products are created, composed and represented.
- A system should be configured with one of multiple families of products.
- A family of related product objects is designed to be used together and you need to enforce this constraint.

2.1.3 Builder Pattern

The Builder pattern separates the construction of a complex object from its representation so that the same construction process can create different representation.

Listing 2.3: Builder Pattern

```
#pragma once
#include <format>
#include <memory>

using namespace std;

/* Product */
struct ArmourSet {
    string breastplate;
    string helm;
    string legs;
    string boots;
    string gauntlets;

    string weapon;
    string shield;
```

```
17
18      constexpr operator string() const {
19          return format(
20              "ArmourSet: [\n\tbreastplate: {}\n\thelm: {}\n\tlegs: {}\n\tboots:
                    {}\n\tgauntlets: {}\n\tweapon: "
21              "{}\n\tshield: {}\n\t]",
22              breastplate, helm, legs, boots, gauntlets, weapon, shield);
23      }
24
25      void setBreastplate(const string& breastplate) { this->breastplate =
                breastplate; }
26      void setHelm(const string& helm) { this->helm = helm; }
27      void setLegs(const string& legs) { this->legs = legs; }
28      void setBoots(const string& boots) { this->boots = boots; }
29      void setGauntlets(const string& gauntlets) { this->gauntlets = gauntlets; }
30      void setWeapon(const string& weapon) { this->weapon = weapon; }
31      void setShield(const string& shield) { this->shield = shield; }
32  };
33
34  /* Abstract Builder */
35  class ArmourSetBuilder {
36     public:
37      ArmourSet getAmourSet() { return armourSet; }
38
39      void createNewAmourSet() { armourSet = ArmourSet{}; }
40
41      virtual void buildBreastPlate() = 0;
42      virtual void buildHelm() = 0;
43      virtual void buildLegs() = 0;
44      virtual void buildBoots() = 0;
45      virtual void buildGauntlets() = 0;
46      virtual void buildWeapon() = 0;
47      virtual void buildShield() = 0;
48
49      ~ArmourSetBuilder() = default;
50
51     protected:
52      ArmourSet armourSet;
53  };
54
55  class NorseArmourSetBuilder : public ArmourSetBuilder {
56     public:
57      void buildBreastPlate() override { armourSet.setBreastplate("Hofund's
                Hauberk"); }
58      void buildHelm() override { armourSet.setHelm("Heimdal's Hjalmar"); }
59      void buildLegs() override { armourSet.setLegs("Gorm's Greaves"); }
60      void buildBoots() override { armourSet.setBoots("Bjorn's Boots"); }
61      void buildGauntlets() override { armourSet.setGauntlets("Freyja's Fists"); }
62      void buildWeapon() override { armourSet.setWeapon("Sumarbrander"); }
63      void buildShield() override { armourSet.setShield("Yggdrasil's Yoke"); }
64      virtual ~NorseArmourSetBuilder() = default;
65  };
66
67  class RomanArmourSetBuilder : public ArmourSetBuilder {
68     public:
69      virtual ~RomanArmourSetBuilder() = default;
```

```
    void buildBreastPlate() override { armourSet.setBreastplate("Lorica
        Segmentata of Mars"); }
    void buildHelm() override { armourSet.setHelm("Galea of Gaius"); }
    void buildLegs() override { armourSet.setLegs("Greaves of Saturn"); }
    void buildBoots() override { armourSet.setBoots("Caligae of Caesar"); }
    void buildGauntlets() override { armourSet.setGauntlets("Manicae of Minerva"
        ); }
    void buildWeapon() override { armourSet.setWeapon("Gladius of Jupiter"); }
    void buildShield() override { armourSet.setShield("Scutum of Venus"); }
};

class MongolArmourSetBuilder : public ArmourSetBuilder {
   public:
    virtual ~MongolArmourSetBuilder() = default;
    void buildBreastPlate() override { armourSet.setBreastplate("Khatangu Degel"
        ); }
    void buildHelm() override { armourSet.setHelm("Mongol Helmet"); }
    void buildLegs() override { armourSet.setLegs("Mongol Leggings"); }
    void buildBoots() override { armourSet.setBoots("Mongol Boots"); }
    void buildGauntlets() override { armourSet.setGauntlets("Mongol Gauntlets");
         }
    void buildWeapon() override { armourSet.setWeapon("Mongol Bow"); }
    void buildShield() override { armourSet.setShield("Mongol Shield"); }
};

/* Director */
class Armourer {
   public:
    void setBuilder(unique_ptr<ArmourSetBuilder> builder) { this->builder = std
        ::move(builder); }

    ArmourSet getAmourSet() { return builder->getAmourSet(); }

    void constructArmourSet() {
        builder->createNewAmourSet();
        builder->buildBreastPlate();
        builder->buildHelm();
        builder->buildLegs();
        builder->buildBoots();
        builder->buildGauntlets();
        builder->buildWeapon();
        builder->buildShield();
    }

   private:
    unique_ptr<ArmourSetBuilder> builder;
};
```

The builder pattern should be used when:

- The algorithm for creating a complex object should be independent of the parts that make up the object and how they are assembled.
- The construction process must allow different representations for the object that is constructed.

2.1.3.1 Consequences of the Builder Pattern

- The Builder Pattern lets you vary a product's internal representation.

- Each specific builder is independent of the others and the rest of the program.
- You have more control over the each final product that is constructed as it is constructed step by step.
- The Builder pattern is somewhat like an Abstract Factory in that they both return classes made up of a number of methods and objects. The difference is that the Abstract Factory returns a family of related classes, the Builder constructs a complex object step by step depending on the data provided to it.

2.1.4 Prototype Pattern

Specifies types of objects using a prototypical instance, and create new objects by copying this prototype. The Prototype Pattern should be used when:

Listing 2.4: Prototype Pattern

```
#pragma once

#include <format>
#include <vector>

using namespace std;

template <typename T>
struct Cloneable {
   private:
    Cloneable() = default;

   public:
    virtual T clone() = 0;
    virtual ~Cloneable() = default;
    friend T;
};

struct Swimmer {
    string name;
    int age{};
    string club;
    float time{};
    bool female{};

    constexpr explicit operator string() {
        return format(R"(Name: {}
Age: {}
Club: {}
Time: {}
Sex: {})",
                      name, age, club, time, female ? "Female" : "Male");
    }
};

class SwimData : public Cloneable<SwimData> {
   public:
    SwimData() = default;

    SwimData clone() override {
        SwimData newSwimData{swimmers};
        return newSwimData;
    }
```

```
44
45      void addSwimmer(const Swimmer& swimmer) { swimmers.emplace_back(swimmer); }
46
47      size_t getNumOfSwimmers() const { return swimmers.size(); }
48
49      Swimmer& getSwimmer(size_t i) { return swimmers.at(i); }
50
51    private:
52      SwimData(vector<Swimmer> swimmers) : swimmers{swimmers} {}
53      vector<Swimmer> swimmers;
54  };
```

- Creating a new instance is more expensive than copying an existing instance.
- You need classes that differ only in the type of processing they offer, for example parsing of strings representing different radixes.

2.1.4.1 Consequences of the Prototype Pattern

- You can add and remove classes at runtime as needed.
- You can revise the internal data representation of classes at runtime based on program conditions.
- You can specify new objects at runtime without creating a lot of classes and subclasses.
- Classes that have circular references cannot be cloned easily.
- Language implementation issues may arise for example in Java, if the classes already exist, you may not be able to change them to implement the Cloneable interface. The deep clone method can be particularly difficult to implement if the all the class objects contained in a class cannot be declared to implement the Serializable interface.

2.1.5 Singleton Pattern

Sometimes it is appropriate to only have one instance of a class for example:

- Window Managers
- Texture Managers for games
- Print Spoolers
- File Systems

Typically singleton objects are accessed by disparate objects throughout a system, and therefore require a global point of access. With a singleton pattern you can:

- Ensure that a class has only one instance.
- Provide a global point of access to the instance.
- Allow multiple instances in the future without affecting a singleton class's clients.

Listing 2.5: Singleton Pattern

```
1  #pragma once
2
3  #include <map>
4  #include <memory>
5  #include <mutex>
6  #include <optional>
7
8  using namespace std;
9
10 class TextureManagerSingleton;
```

```
using TextureManagerPtr = unique_ptr<TextureManagerSingleton>;
using TextureManager = TextureManagerSingleton;

class TextureManagerSingleton final {
   public:
    static TextureManagerPtr& Instance() {
        // To make ith thread safe
        unique_lock<mutex> lock{mtx};
        if (instance == nullptr) instance = TextureManagerPtr{new
           TextureManagerSingleton{}};
        return instance;
    }

    void addTexture(const string& name, const string& path) { textureMap.insert
       ({name, path}); }

    optional<string> getTexture(const string& name) {
        if (textureMap.contains(name)) return textureMap.at(name);
        return nullopt;
    }

   private:
    static mutex mtx;
    map<string, string> textureMap;
    static TextureManagerPtr instance;
    TextureManagerSingleton() = default;
};

mutex TextureManagerSingleton::mtx{};
TextureManagerPtr TextureManagerSingleton::instance = nullptr;
```

2.1.5.1 Consequences of the Singleton Pattern

- It can be difficult to subclass a singleton class, since it can only work if the base singleton class has not yet been initialized
- It is possible to have multiple instances of a singleton class if the singleton class is not implemented correctly.
- The number of instances of a singleton class can be controlled.