Design Patterns in C++: Creational

SOLID Design Principles

Zachary Bennett

Software Engineer

@z_bennett_ zachbennettcodes.com

SOLID Design Principles

A set of object-oriented design principles that aim to help developers write code that is easily extensible, maintainable and more reusable.

The SOLID Principles are a good base for implementing object-oriented design patterns.

SOLID Principles

Single-responsibility

Classes should have a single purpose

Open-closed

Open for extension, closed for modification

Liskov Substitution

Behavioral class substitution

Interface Segregation

Favor multiple, specific interfaces over a single interface

Dependency Inversion

Depend on abstractions not implementations

Why SOLID?

More extensible code

Easier to maintain code

Reusable code

Module Flow

What “not” to do

See examples of code that is ready for a redesign

What “to” do

Look at code that properly implements SOLID principles

Single-responsibility Principle

Single-responsibility Principle

Every class/module should be responsible for one portion of the overall system.

Benefits

Avoid ”spaghetti” code

Allows you to implement proper separation of concerns

Maintainability

```
class CoffeeMachine
{
    . . .

    void pourCoffee()
    {

      std::cout << ”Pouring coffee”;
    }

    void sendCoffeeMetrics()
    {

      std::cout << ”Sending metrics”;

      UrlRequest request;
      request.uri(“/metrics”);
      . . .
      request.perform();
    }

}
```

◀ Here’s a CoffeeMachine implementation showing what not to do

◀ This method belongs here and contains code that is specific to the responsibility of a CoffeeMachine

◀ This method contains implementation details that are not specific to a CoffeeMachine

```
class CoffeeMachine
{
    . . .

    void pourCoffee()
    {

      std::cout << ”Pouring coffee”;
    }

    void sendCoffeeMetrics()
    {

      std::cout << ”Sending metrics”;

      coffeeService.sendMetrics(metrics)

    }

}
```

◀ This class implements the Single-responsibility Principle

◀ This updated method is not concerned with the details of sending metrics

◀ This CoffeeMachine class does not have to change

Open-closed Principle

Open-closed Principle

It should be easy to extend a class’s behavior without changing the code of the class itself.

Benefits

Changing requirements does not necessarily mean changing code

Well-defined API’s

Reusable code

```
class CoffeeMachine {

    . . .


  private:

    // Settings are fixed
    vector<int> settings = { 1, 2, 3 }



    void roastBySetting(int setting) {
       // Ensure that setting is included
       // in valid settings
       switch(setting) {
         case 1:
         . . .
         case 2:
         . . .
       }
    }

}
```

◀ **Here’s a CoffeeMachine implementation showing what not to do**

◀ **Any time a new roast setting needs to be added, this code needs to update**

◀ **This class is not easily extensible, and it encourages direct code modification**

```
class CoffeeMachine {

  . . .



  // Settings are dynamic
  void roastBySetting(RoastSetting setting) {
       . . .

       roastingService.roast(&setting)
  }


}
```

- This class implements the Open-closed Principle

- When a new RoastSetting is added, the CoffeeMachine class code does not need to update

- If the RoastSetting type changed, the CoffeeMachine class would still not need to be changed

Liskov Substitution Principle

Liskov Substitution Principle

If type “A” is derived from type “B” then you should be able to substitute objects of type “B” for objects of type “A”.

It’s just behavioral subtyping.

Benefits

Flexibility

Well-defined abstractions

Reusable code

```
class Roaster {
  public:
    virtual void     roast();
}


class CoffeeRoaster : Roaster {
  public:
    void roast() {
      // Specific coffee implementation
    }
}

class EspressoRoaster : Roaster {
  public:
    void roast() {
      // Specific espresso implementation
    }
}

. . . Usage

void roast(Roaster roaster) {
  roaster.roast() // Doesn’t care about type
}
```

◂ **Base class**

◂ **Coffee-specific implementation**

◂ **Espresso-specific implementation**

◂ **Outside functions can unknowingly use either the coffee or espresso implementation**

Interface Segregation Principle

Interface Segregation Principle

Clients using your code should not be forced into depending upon methods or other abstractions that they don’t need.

Benefits

Possible reduction in compile time

Maintainability

Proper separation of concerns

```
class Machine {
  public:
    virtual void roast();
    virtual void pour();
    virtual void grind();
}
```

◀ **Base class**

```
class AllInOneCoffeeMachine : Machine {
  public:
    void roast() …
    void pour() …
    void grind() …
}
```

◀ **This class happens to need to implement all of the Machine methods**

```
class SimpleCoffeeMachine : Machine {
  public:
    void pour() …

    // Has to implement unneeded methods
}
```

◀ **This class doesn’t need to implement all the Machine methods so this violates the ISP.**

```
struct Roaster { virtual void roast(); }
struct Pourer  { virtual void pour();  }
struct Grinder { virtual void grind(); }

struct RobustMachine : Roaster, Pourer, Grinder {

}


class AllInOneCoffeeMachine : RobustMachine {
  public:
    void roast() …
    void pour() …
    void grind() …
}

class SimpleCoffeeMachine : Pourer {
  public:
    void pour() …
}
```

◂ **Individual interfaces**

◂ **This class can inherit from multiple interfaces to bring in all the functionality that it needs**

◂ **Individual interfaces allow clients to use only want they need**

Dependency Inversion Principle

Dependency Inversion Principle

High-level modules (classes which depend upon other, low-level classes of a program) should not depend on low-level modules directly. They should both depend upon an abstraction.

Benefits

Loose coupling of software

Huge benefit to code reusability

Proper separation of concerns

```
class CoffeeMachine {
  vector<int> status;
  …
}


class CoffeeTest {
  void start(CoffeeMachine &machine) {
    for (auto bit: machine.status) {
       // operate on status bits
    }
  }
}
```

- **Low-level “module”**

- **High-level “module”**

- **If the CoffeeMachine class implementation changes than the CoffeeTest class will need to change as well**

```
struct CoffeeStatusReader {
  virtual vector<int> readStatus();
}
```

◂ **Shared abstraction**

```
class CoffeeMachine : CoffeeStatusReader {
  vector<int> status;

  void readStatus() {
    for (auto bit: status) {

    }
  }
   …
}
```

◂ **Low-level “module”**

```
class CoffeeTest {
  void start(CoffeeStatusReader &reader) {
     reader.readStatus();
  }
}
```

◂ **The high-level module no longer depends upon the low-level module. The implementation of the low-level functionality can change without the high-level module needing to change as well.**

Summary

SOLID Principles

- Single-responsibility principle
- Open-closed principle
- Liskov substitution principle
- Interface segregation principle
- Dependency inversion principle

Foundation for design patterns