Обектно ориентирано програмиране

OO DESIGN PATTERNS

What are Software Design Principles?

Software design principles represent a set of guidelines that helps us to avoid having a bad design. The design principles are associated to Robert Martin who gathered them in "Agile Software Development: Principles, Patterns, and Practices". According to Robert Martin there are 3 important characteristics of a **bad design** that should be avoided:

- **Rigidity** - It is hard to change because every change affects too many other parts of the system.
- **Fragility** - When you make a change, unexpected parts of the system break.
- **Immobility** - It is hard to reuse in another application because it cannot be disentangled from the current application.

SOLID Design Principles

- **S**ingle Responsibility Principle
- **O**pen/Closed Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion

Single Responsibility Principle

A class should have only one reason to change.

In this context a responsibility is considered to be one reason to change. This principle states that if we have 2 reasons to change for a class, we have to split the functionality in two classes. Each class will handle only one responsibility and on future if we need to make one change we are going to make it in the class which handle it. When we need to make a change in a class having more responsibilities the change might affect the other functionality of the classes.

Single Responsibility Principle

The argument for the single responsibility principle is relatively simple: it makes your software easier to implement and prevents unexpected side-effects of future changes.

You can avoid these problems by asking a simple question before you make any changes: What is the responsibility of your class/component/microservice?

If your answer includes the word “and”, you’re most likely breaking the single responsibility principle. Then it’s better to take a step back and rethink your current approach. There is most likely a better way to implement it.

Open Close Principle

Software entities like classes, modules and functions should be ***open for extension*** *but* ***closed for modifications.***

OCP is a generic principle. You can consider it when writing your classes to make sure that when you need to extend their behavior you don’t have to change the class but to extend it. The same principle can be applied for modules, packages, libraries. If you have a library containing a set of classes there are many reasons for which you’ll prefer to extend it without changing the code that was already written (backward compatibility, regression testing, etc.). This is why we have to make sure our modules follow Open Closed Principle.

Open Close Principle (cont.)

“A class is closed, since it may be compiled, stored in a library, baselined, and used by client classes. But it is also open, since any new class may use it as parent, adding new features. When a descendant class is defined, there is no need to change the original or to disturb its clients.”

Open/Closed Principle uses interfaces instead of superclasses to allow different implementations which you can easily substitute without changing the code that uses them. The interfaces are closed for modifications, and you can provide new implementations to extend the functionality of your software.

Liskov Substitution Principle

Let Φ(x) be a property provable about objects x of type T. Then Φ(y) should be true for objects y of type S where S is a subtype of T.

Bad example

```
class Bird {
public:
   void fly() {}
};
class Duck : public Bird {};
```

`The duck can fly because it is a bird, but what about this:`

```
class Ostrich: public Bird {};
```

Ostrich is a bird, but it can't fly, Ostrich class is a subtype of class Bird, but it shouldn't be able to use the fly method, that means we are breaking the LSP principle.

Liskov Substitution Principle

Good example

```
class Bird {};
class FlyingBirds : public Bird {
public:
   void fly() {}
};
class Duck : public FlyingBirds {};
class Ostrich: public Bird {};
```

Interface Segregation Principle

Clients should not be forced to depend upon interfaces that they don't use.

This principle teaches us to take care how we write our interfaces. When we write our interfaces we should take care to add only methods that should be there. If we add methods that should not be there the classes implementing the interface will have to implement those methods as well.

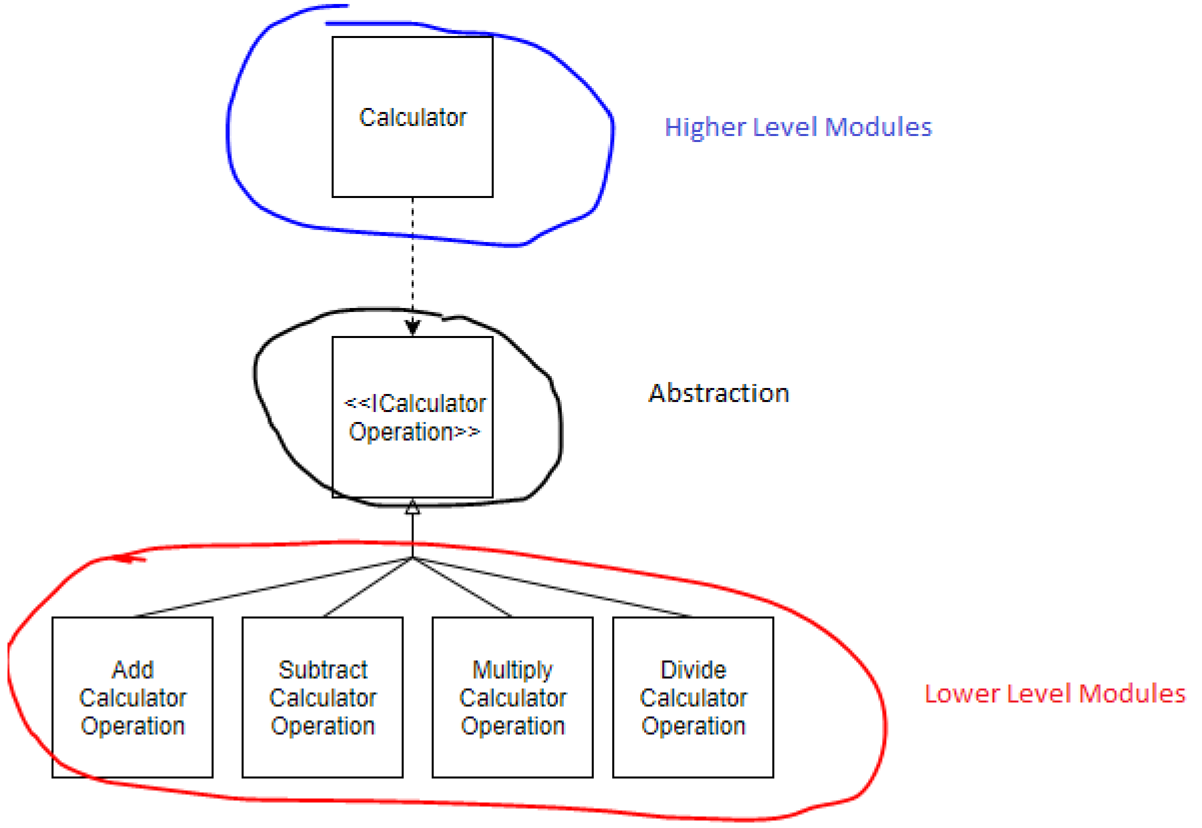
Dependency Inversion Principle

High-level modules should not depend on low-level modules. Both should depend on abstractions.

Abstractions should not depend on details. Details should depend on abstractions.

Dependency Inversion Principle states that we should decouple high level modules from low level modules, introducing an abstraction layer between the high level classes and low level classes. Further more it inverts the dependency: instead of writing our abstractions based on details, the we should write the details based on abstractions.

Dependency Inversion Principle example


Calculator
Higher Level Modules
<<ICalculator Operation>>
Abstraction
Add Calculator Operation
Subtract Calculator Operation
Multiply Calculator Operation
Divide Calculator Operation
Lower Level Modules

Design patterns

Creational Design Patterns

Behavioral Design Patterns

Structural Design Patterns

http://www.oodesign.com/

https://en.wikipedia.org/wiki/Design_Patterns

Creational Design Patterns

Creational patterns are ones that create objects for you, rather than having you instantiate objects directly. This gives your program more flexibility in deciding which objects need to be created for a given case.

Singleton - **Ensure that only one instance of a class is created** and **Provide a global access point to the object**.

Factory - **Creates objects without exposing the instantiation logic to the client** and **Refers to the newly created object through a common interface**.

Creational Design Patterns

Abstract Factory - Offers the interface for creating a family of related objects, without explicitly specifying their classes.

Builder - Defines an instance for creating an object but letting subclasses decide which class to instantiate and Allows a finer control over the construction process.

Prototype - Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Behavioral Design Patterns

Most of these design patterns are specifically concerned with communication between **objects**.

Chain of responsibility delegates commands to a chain of processing objects.

Command creates objects which encapsulate actions and parameters.

Interpreter implements a specialized language.

Iterator accesses the elements of an object sequentially without exposing its underlying representation.

Mediator allows loose coupling between classes by being the only class that has detailed knowledge of their methods.

Behavioral Design Patterns

Memento provides the ability to restore an object to its previous state (undo).

Observer is a publish/subscribe pattern which allows a number of observer objects to see an event.

State allows an object to alter its behavior when its internal state changes.

Strategy allows one of a family of algorithms to be selected on-the-fly at runtime.

Template method defines the skeleton of an algorithm as an abstract class, allowing its subclasses to provide concrete behavior.

Visitor separates an algorithm from an object structure by moving the hierarchy of methods into one object.

Structural Design Patterns

These concern class and object composition. They use inheritance to compose interfaces and define ways to compose objects to obtain new functionality.

Adapter allows classes with incompatible interfaces to work together by wrapping its own interface around that of an already existing class.

Bridge decouples an abstraction from its implementation so that the two can vary independently.

Composite composes zero-or-more similar objects so that they can be manipulated as one object.

Structural Design Patterns

Decorator dynamically adds/overrides behaviour in an existing method of an object.

Facade provides a simplified interface to a large body of code.

Flyweight reduces the cost of creating and manipulating a large number of similar objects.

Proxy provides a placeholder for another object to control access, reduce cost, and reduce complexity.

Singleton

The **Singleton pattern** ensures that a class has only one instance and provides a global point of access to that instance. It is named after the singleton set, which is defined to be a set containing one element. This is useful when exactly one object is needed to coordinate actions across the system.

Check list

- Define a private static attribute in the "single instance" class.
- Define a public static accessor function in the class.
- Do "lazy initialization" (creation on first use) in the accessor function.
- Define all constructors to be protected or private.
- Clients may only use the accessor function to manipulate the Singleton.

Singleton

```
class StringSingleton
{
public:
    // Some accessor functions for the class, itself
    std::string GetString() const
    {
        return mString;
    }
    void SetString(const std::string &newStr)
    {
        mString = newStr;
    }

    // The magic function, which allows access to the class from anywhere
    // To get the value of the instance of the class, call:
    //     StringSingleton::Instance().GetString();
    static StringSingleton &Instance()
    {
        // This line only runs once, thus creating the only instance in existence
        static StringSingleton *instance = new StringSingleton;
        // dereferencing the variable here, saves the caller from having to use
        // the arrow operator, and removes temptation to try and delete the
        // returned instance.
        return *instance; // always returns the same instance
    }
```

Singleton

```
private:
    // We need to make some given functions private to finish the definition of the singleton
    StringSingleton() {} // default constructor available only to members or friends of this class

                         // Note that the next two functions are not given bodies, thus any attempt
                         // to call them implicitly will return as compiler errors. This prevents
                         // accidental copying of the only instance of the class.
    StringSingleton(const StringSingleton &old); // disallow copy constructor
    const StringSingleton &operator=(const StringSingleton &old); //disallow assignment operator

          // Note that although this should be allowed,
          // some compilers may not implement private destructors
          // This prevents others from deleting our one single instance, which was otherwise
created on the heap
    ~StringSingleton() {}
private: // private data for an instance of this class
    std::string mString;
};
```

Factory

Definition: A utility class that creates an instance of several families of classes. It can also return a factory for a certain group.

- 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. The 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.

Problem

- We want to decide at run time what object is to be created based on some configuration or application parameter. When we write the code, we do not know what class should be instantiated.

Solution

- Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Factory

```cpp
class Computer
{
public:
    virtual void Run() = 0;
    virtual void Stop() = 0;

    virtual ~Computer() {}; /* without this, you do not call Laptop or Desktop destructor in this example! */
};
class Laptop : public Computer
{
public:
    virtual void Run() { mHibernating = false; };
    virtual void Stop() { mHibernating = true; };
    virtual ~Laptop() {}; /* because we have virtual functions, we need virtual destructor */
private:
    bool mHibernating; // Whether or not the machine is hibernating
};
class Desktop : public Computer
{
public:
    virtual void Run() { mOn = true; };
    virtual void Stop() { mOn = false; };
    virtual ~Desktop() {};
private:
    bool mOn; // Whether or not the machine has been turned on
};
```

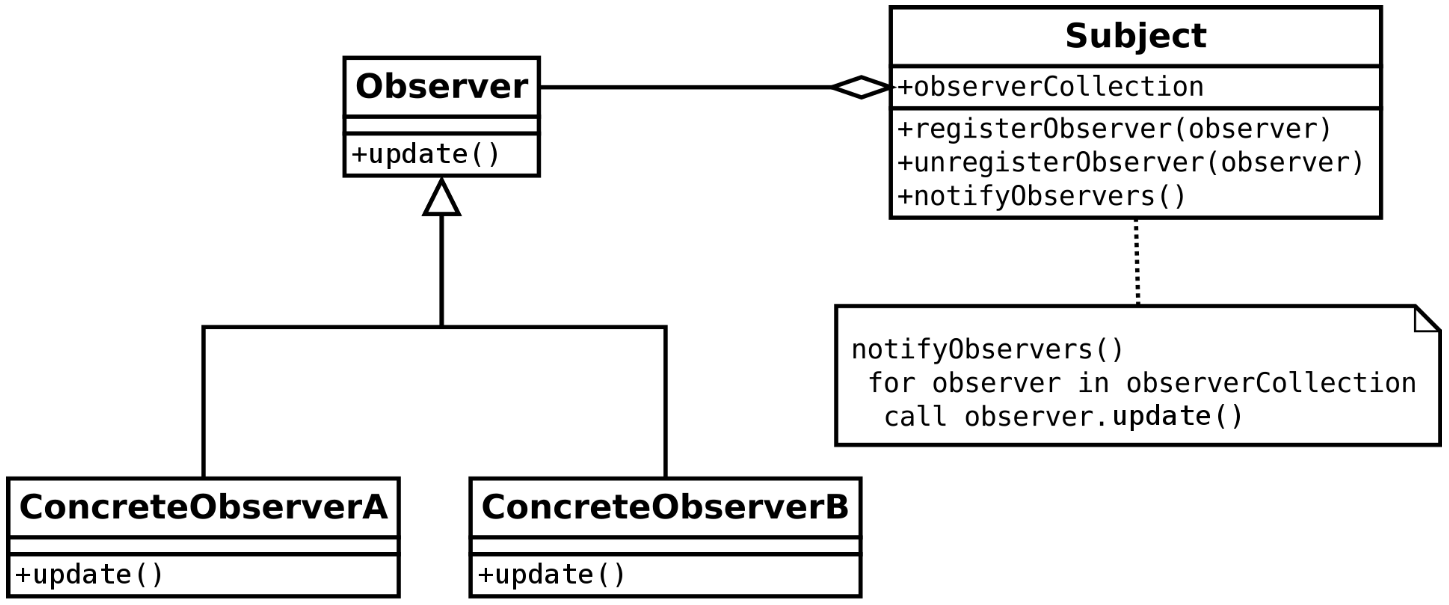
Factory

```
class ComputerFactory
{
public:
    static Computer *NewComputer(const std::string &description)
    {
        if (description == "laptop")
            return new Laptop;
        if (description == "desktop")
            return new Desktop;
        return NULL;
    }
};
```

Observer

Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Observer


Observer
+update()
Subject
+observerCollection
+registerObserver(observer)
+unregisterObserver(observer)
+notifyObservers()
notifyObservers()
for observer in observerCollection
call observer.update()
ConcreteObserverA
+update()
ConcreteObserverB
+update()

Observer

Пример: Централна банка и наблюдатели

Задача. Централна Банка и наблюдатели

Да се реализира клас **CentralBank** (централна банка). Централната банка (например БНБ) има име (низ – например БНБ) и определя курсовете на валутите спрямо местната валута. За целта класът **CentralBank** трябва да поддържа списък с курсовете на валути (име на валутата (**currency**) и курс (**rate**)). Да се реализират методи:

- **addCurrency** – добавя валута към списъка
- **deleteCurrency** – изтрива валута от списъка
- **setRate** – задава курс за дадена валута

Освен това има наблюдатели на курсовете в Централната банка (други банки, Change бюра, вестници, телевизии и др.). Централната банка поддържа списък с указатели към наблюдателите. Да се реализират следните методи:

- **register** – добавя (регистрира) наблюдател към списъка
- **unregister** – изтрива наблюдател от списъка
- **notify** – уведомява всички регистрирани наблюдатели за промяна на курс на валута, като извиква метод **update** на наблюдателя. Методът трябва да се извиква при всяка промяна на курс на валута.

Observer

Да се дефинира абстрактен клас **Observer** (наблюдател), който има чисто виртуален метод update за промяна на курс на дадена валута.

Да се дефинира производен клас **ConcreteObserver** (конкретен наблюдател), който има име (низ) и реализира метода **update**, който извежда на екрана:

Update <име на наблюдателя>: <валута> <курс>

Да се реализира главна програма, която
- Създава обект от клас **CentralBank**
- Добавя валути към обекта
- Създава няколко обекта от клас **ConcreteObserver**
- Регистрира наблюдателите в обекта на **CentralBank**
- Променя курс на валута
- Изтрива наблюдател
- Променя курс на валута