OOP Design Principles

Arctech Info Private Limited

Importance of Good design

Features of good software design

GOOD DESIGN OF INDIVIDUAL CLASSES IS CRUCIAL TO GOOD OVERALL SYSTEM DESIGN.

A WELL-DESIGNED CLASS IS MORE RE-USABLE IN DIFFERENT CONTEXTS, AND MORE MODIFIABLE FOR FUTURE VERSIONS OF SOFTWARE.

TODAY, LET US LOOK AT SOME GENERAL CLASS DESIGN GUIDELINES, AS WELL AS SOME TIPS FOR SPECIFIC LANGUAGES, LIKE C#

General goals for building a good class

- Having a good usable interface
- Implementation objectives, like efficient algorithms and convenient/simple coding
- Separation of implementation from interface!!
- Improves modifiability and maintainability for future versions
- Decreases coupling between classes, i.e., the amount of dependency between classes (will changing one class require changes to another?)
- Can re-work a class inside, without changing its interface, for outside interactions
- Consider how much the automobile has advanced technologically since its invention. Yet the basic interface remains the same -- steering wheel, gas pedal, brake pedal, etc.

Encapsulation

- Hiding implementation and exposing interface
  - Private data and methods vs the public methods
  - Avoid returning reference or pointers to private data members
- Ensure the object is always in a valid state
  - e.g., a zero denominator would be invalid for a Fraction, a negative radius makes no sense for a Circle
  - All constructors should initialize the object to a valid state
  - Make sure the class has at least one programmer-defined constructor (default constructor in C++ does nothing)
  - All public methods should leave the object in a valid state before returning

The Law of Demeter

- This refers to a general "principle of least knowledge" guideline for developing software
- Fundamental notion is that an object should assume as little as possible about other objects or components
- A method that follows the Law of Demeter does not operate on
  - global objects
  - data that are a part of another object

Design Patterns

Design patterns are typical solutions to commonly occurring problems in software design.

They are like pre-made blueprints that you can customize to solve a recurring design problem in your code.

You can’t just find a pattern and copy it into your program, the way you can with off-the-shelf functions or libraries.

The pattern is not a specific piece of code, but a general concept for solving a particular problem.

You can follow the pattern details and implement a solution that suits the realities of your own program.

Patterns are often confused with algorithms, because both concepts describe typical solutions to some known problems.

an algorithm always defines a clear set of actions that can achieve some goal

a pattern is a more high-level description of a solution.

The code of the same pattern applied to two different programs may be different.

An analogy to an algorithm is a cooking recipe:

both have clear steps to achieve a goal.

On the other hand, a pattern is more like a blueprint:

you can see what the result and its features are, but the exact order of implementation is up to you.

Factory Pattern example

```
class Weapon  //abstract
{...}

class Gun : public Weapon
{...}

class Bazooka : public Weapon
{...}

class Game
{
    void Start()
    {
       Weapon *weapon;
       char ch;
       count << "Select weapon: ";
       cin >> ch;

       switch(ch)
       {
           case 'g':
              weapon = new Gun();
              break;

           case 'b':
              weapon = new Bazooka();
              break;
       }
   }
}
```

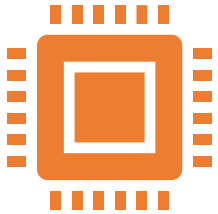
- Factory Pattern

```
class Game
{
   void Start()
   {
       Weapon *weapon;
       char ch;
       count << "Select weapon: ";
       cin >> ch;

       weapon = WeaponsFactory::Create(ch);
   }
}

class WeaponsFactory
{
    public static Weapon* Create(char ch)
    {
       switch(ch)
       {
           case 'g':
              return new Gun();

           case 'b':
              return new Bazooka();
       }
    }
}
```

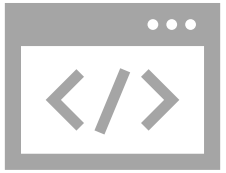
SOLID

- SOLID is a popular set of design principles that are used in object-oriented software development.
- SOLID is an acronym that stands for five key design principles:
  - single responsibility principle,
  - open-closed principle,
  - Liskov substitution principle,
  - interface segregation principle, and
  - dependency inversion principle.
- All five are commonly used by software engineers and provide some important benefits for developers.

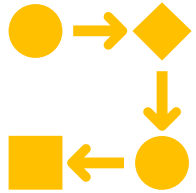
Goal of SOLID



The broad goal of the SOLID principles is to reduce dependencies so that engineers change one area of software without impacting others.



Additionally, they're intended to make designs easier to understand, maintain, and extend.



Ultimately, using these design principles makes it easier for software engineers to avoid issues and to build adaptive, effective, and agile software.

History of SOLID

- The SOLID principles were developed by Robert C. Martin in a 2000 essay, “Design Principles and Design Patterns,”
  - the acronym S O L I D was coined later by Michael Feathers.
- In his essay, Martin acknowledged that successful software will change and develop.
- As it changes, it becomes increasingly complex.
- Without good design principles, Martin warns that software becomes rigid, fragile, immobile, and viscous.
- The SOLID principles were developed to combat these problematic design patterns.

Single Responsibility Principle

- Robert Martin summarizes this principle well by mandating that,
  - “a class should have one, and only one, reason to change.”
- Following this principle means that each class does only one thing and one thing only.
- In other words, every class or module only has responsibility for only one small part of the entire software’s functionality.
- More simply, each class should solve only one problem.

Open-Closed Principle

- IT is expected that existing, well-tested classes will need to be modified when something needs to be added.
- Yet, changing classes can lead to problems or bugs.
- Instead of changing the class, you simply want to extend it.
- With that goal in mind, Martin summarizes this principle, “You should be able to extend a class’s behavior without modifying it.”
- See the Weapon example, where
  - The abstract base class Weapon ensures then a new weapon can be created in separate classes (e.g., Rocket.h & Rocket.cpp) which inherit from Weapon
  - The Factory pattern ensures the conditional construction of weapons is handled in a separate Factory class
- Using both these design patterns has helped us avoid modifying the Game class

Some drawbacks of good design principles

- The principles come with many benefits
- But following the principles generally leads to writing longer and more complex code.
- Thus, it can extend the design process and make development a little more difficult.
- However, this extra time and effort is well worth it because it makes software so much easier to maintain, test, and extend.
- Following these principles is not a cure-all and won't avoid design issues.
- That said, the principles have become popular because when followed correctly, they lead to better code for readability, maintainability, design patterns, and testability.
- In the current environment, all developers should know and utilize these principles.

Conclusion

- Implementing design principles during development will lead to systems that are more maintainable, scalable, testable, and reusable.
- In the current environment, these principles are used globally by engineers.
- As a result, to create good code and to use design principles that are competitive while meeting industry standards, it’s essential to utilize these principles.
- Implementing these principles can feel overwhelming at first
- Regularly working with them will help you become comfortable with it, (few months to years for becoming an expert).