Topics in C/C++

Design Pattern

Lecturer: Duc Dung Nguyen, PhD.
Contact: nddung@hcmut.edu.vn

Faculty of Computer Science and Engineering
Hochiminh city University of Technology

Contents

SOLID

SOLID

- Five principles of Object-Oriented class design.
  A set of rules and best practices to follow while designing a class structure.
- First introduced by Robert J. Martin (a.k.a Uncle Bob, the author of “Clean Code and Clean Architecture”) in 2000. But the SOLID acronym was introduced later by Michael Feathers.
  “To create understandable, readable, and testable code that many developers can collaboratively work on.”

SOLID

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

SOLID

- The **S**ingle Responsibility Principle
  - Only one potential change (database logic, logging logic, and so on.) in the softwares specification should be able to affect the specification of the class.
  - Team work is important
  - Make version control easier

- The **O**pen-Closed Principle
  - Classes should be open for extension and closed to modification

SOLID

- The **O**pen-Closed Principle
  - Classes should be open for extension and closed to modification
- The **L**iskov Substitution Principle
  - Subclasses should be substitutable for their base classes
  - Given that class B is a subclass of class A, we should be able to pass an object of class B to any method that expects an object of class A and the method should not give any weird output in that case.

- The **I**nterface Segregation Principle
  - Segregation means keeping things separated, and the Interface Segregation Principle is about separating the interfaces
  - Many client-specific interfaces are better than one general-purpose interface.
  - Clients should not be forced to implement a function they do no need.

SOLID

- The **I**nterface Segregation Principle
  - Segregation means keeping things separated, and the Interface Segregation Principle is about separating the interfaces
  - Many client-specific interfaces are better than one general-purpose interface.
  - Clients should not be forced to implement a function they do no need.
- The **D**ependency Inversion Principle
  - Classes should depend upon interfaces or abstract classes instead of concrete classes and functions

Design Patterns

Design Patterns

- Creational Patterns
- Structural Patterns
- Behavioral Patterns

Creational Patterns - Builder

- Allows a client object to construct a complex object by specifying only its type and content.
- The construction process can be used to create different representations.
- Problem: We want to construct a complex object, however, we do not want to have a complex constructor member or one that would need many arguments. So we define an instance for creating an object but letting subclasses decide which class to instantiate and refer to the newly created object through a common interface.
- Solution: Define an intermediate object whose member functions define the desired object part by part before the object is available to the client. Builder Pattern lets us defer the construction of the object until all the options for creation have been specified.

Creational Patterns - Factory

- A utility class that creates an instance of a class from a family of derived classes. The problem here is that once we add a new concrete product call we should modify the Factory class. It is not very flexible and it violates the open-close principle.
- Problem: creates objects without exposing the instantiation logic to the client. refers to the newly created object through a common interface
- Solution: parameterized Factories

Creational Patterns - Factory Method

- It defines an interface for creating an object, but leaves the choice of its type to the subclasses, creation being deferred at run-time. It refers to the newly created object through a common interface.
- *Factory Method is similar to Abstract Factory but without the emphasis on families.*
- Problem: A framework needs to standardize the architectural model for a range of applications, but allow for individual applications to define their own domain objects and provide for their instantiation.
- **[TR]: Abstract Factory**

Creational Patterns - Prototype

- A prototype pattern is used when the type of objects to create is determined by a prototypical instance, which is cloned to produce new objects.
- Problem: This pattern is used, for example, when the inherent cost of creating a new object in the standard way (e.g., using the new keyword) is prohibitively expensive for a given application.
- Solution: Declare an abstract base class that specifies a pure virtual clone() method. Any class that needs a "polymorphic constructor" capability derives itself from the abstract base class and implements the clone() operation.

Creational Patterns - Singleton

- It ensures that a class has only one instance and provides a global point of access to that instance.
- Problem: This is useful when exactly one object is needed to coordinate actions across the system.
- Solution: The implementation involves a static member in the “Singleton” class, a private constructor and a static public method that returns a reference to the static member.

Structural Patterns - Adapter

- Convert the interface of a class into another interface that clients expect.
- Problem: Adapter lets classes work together that couldnt otherwise because of incompatible interfaces.
- Solution: A Class Adapter Based on (Multiple) Inheritance

Structural Patterns - Bridge

- To separate out the interface (decouple abstraction) from its implementation so that the two can vary independently
- Problem: Sometimes an abstraction should have different implementations, inheritance binds an implementation to the abstraction and thus it would be difficult to modify, extend, and reuse abstraction and implementation independently.

Structural Patterns - Composite

- To compose objects into tree structures to represent part-whole hierarchies.
- Composite lets clients treat individual objects and compositions of objects uniformly. e.g. Treat the file (leaf) and folder (composite) uniformly as they have many properties in common (size, name, etc)
- Problem: There are times when a program needs to manipulate a tree data structure and it is necessary to treat both Branches as well as Leaf Nodes uniformly. Consider for example a program that manipulates a file system where folders contain other folders and files while files are the simple objects or leaves.

Structural Patterns - Decorator

- To add additional responsibilities dynamically to an object.
- Decorators provide a flexible alternative to subclassing for extending functionality.
- This is also called “Wrapper”. A decorator is different from an adapter in that a decorator changes the objects responsibilities, while an adapter changes an object interface.
- Problem: There are times when a program needs to manipulate a tree data structure and it is necessary to treat both Branches as well as Leaf Nodes uniformly. Consider for example a program that manipulates a file system where folders contain other folders and files while files are the simple objects or leaves.

Structural Patterns - Facade

- Hide the complexities of the system by providing an interface to the client from where the client can access the system on a unified interface.
- Facade defines a higher-level interface that makes the subsystem easier to use.
- Problem: A segment of the client community needs a simplified interface to the overall functionality of a complex subsystem.
- Solution: Use a facade class and the client only needs to access this class to perform the operations of the subsystems.

[TR] Flyweight, Proxy

Behavioral Patterns - Chain of Responsibility

- To avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request.
- Chains the receiving objects and passes the requests along the chain until an object handles it.
- Problem: There is a potentially variable number of handler or processing element or node objects, and a stream of requests that must be handled. Need to efficiently process the requests without hard-wiring handler relationships and precedence, or request-to-handler mappings.

Behavioral Patterns - Command

- Decouple sender and receiver by encapsulating a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undo-able operations.
- It can also be thought as an object-oriented equivalent of call back method.
- Problem: Need to issue requests to objects without knowing anything about the operation being requested or the receiver of the request.

Behavioral Patterns - Iterator

- Permit the traversal of a container (like a pointer moving across an array). However, to get to the next element of a container, you need not know anything about how the container is constructed.
- By simply using the member functions provided by the iterator, you can move, in the intended order of the container, from the first element to the last element.
- Problem: Need to “abstract” the traversal of wildly different data structures so that algorithms can be defined that is capable of interfacing with each transparently.

Behavioral Patterns - Others

- Interpreter
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor