Agile Software Development (TCS 855)

Unit-III Agile Software Design and PROGRAMMING

Software Design Principle (SDP): SOLID

Prof.(Dr.) Santosh Kumar

Department of Computer Science and Engineering

Graphic Era Deemed to be University, Dehradun

SOLID

- SOLID – the "O" in "SOLID" represents the open–closed principle.
- In software engineering, **SOLID** is a mnemonics acronyms for five design principles intended to make software designs more understandable, flexible, and maintainable.
- The SOLID concepts are

  i. **S: The Single responsibility principle:** "There should never be more than one reason for a class to change. In other words, every class should have only one responsibility.

  i. **O: The Open-Closed principle:** "Software entities ... should be open for extension, but closed for modification

  i. **L: The Liskov substitution principle:** "Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.

  i. **I: The Interface segregation principle:** "Many client-specific interfaces are better than one general-purpose interface."

  i. **D: The Dependency inversion principle:** "Depend upon abstractions, [not] concretions.

the SOLID principles apply to any object-oriented design, they can also form a core philosophy for methodologies such as agile development or adaptive software development

Single-responsibility principle (SRP)

- The **single-responsibility principle** (**SRP**) is a computer-programming principle that states that every module, class or function a computer program should have responsibility over a single part of that program's functionality, and it should encapsulate that part.
- All of that module, class or function's services should be narrowly aligned with that responsibility

Open–closed principle

- In Object Oriented programming, the **open–closed principle** states "*software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification*“, that is, such an entity can allow its behaviour to be extended without modifying its source code.
- The name *open–closed principle* has been used in two ways. Both ways use generalizations (for instance, inheriotance or delegate functions) to resolve the apparent dilemma, but the goals, techniques, and results are different.
- Open–closed principle is one of the five **SOLID** principles of object-oriented design.

Meyer's open–closed principle

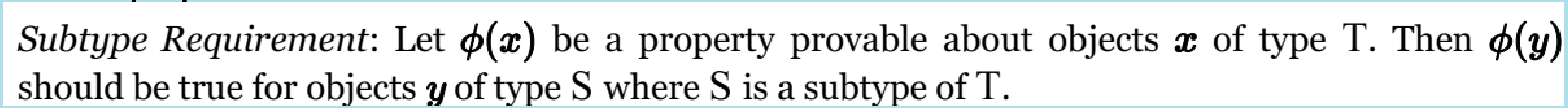
- Bertrand Meyer is generally credited for having originated the term *open–closed principle.*
  - A module will be said to be open if it is still available for extension. For example, it should be possible to add fields to the data structures it contains, or new elements to the set of functions it performs.
  - A module will be said to be closed if [it] is available for use by other modules. This assumes that the module has been given a well-defined, stable description (the interface in the sense of information hiding).
  - At the time Meyer was writing, adding fields or functions to a library inevitably required changes to any programs depending on that library.
  - Meyer's proposed solution to this dilemma relied on the notion of object-oriented inheritance (specifically implementation inheritance):
    - 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.

Polymorphic open–closed principle

- During the 1990s, the open–closed principle became popularly redefined to refer to the use of abstracted interfaces, where the implementations can be changed and multiple implementations could be created and polymorphically substituted for each other.
- n contrast to Meyer's usage, this definition advocates inheritance from abstract base classes. Interface specifications can be reused through inheritance but implementation need not be. The existing interface is closed to modifications and new implementations must, at a minimum, implement that interface.

Liskov substitution principle

- **Substitutability** is a principle in object-oriented programming stating that, in a computer program, if S is a subtype of T, then objects of type T may be *replaced* with objects of type S (i.e., an object of type T may be *substituted* with any object of a subtype S) without altering any of the desirable properties of the program (correctness, task performed, etc.).
- More formally, the **Liskov substitution principle** (**LSP**) is a particular definition of a subtyping relation, called (**strong**) **behavioral subtyping**, that was initially introduced by Barbara Liskov in a 1988 conference keynote address titled *Data abstraction and hierarchy*.
- It is a semantic rather than merely syntactic relation, because it intends to guarantee semantic interoperability of types in a hierarchy, object types in particular.
- Barbara Liskov and Jeannette Wing described the principle succinctly in a 1994 paper as follows:



Subtype Requirement: Let $\phi(\boldsymbol{x})$ be a property provable about objects $\boldsymbol{x}$ of type T. Then $\phi(\boldsymbol{y})$ should be true for objects $\boldsymbol{y}$ of type S where S is a subtype of T.

Liskov substitution principle Contd…

- Liskov's notion of a behavioural subtype defines a notion of substitutability for objects; that is, if *S* is a subtype of *T*, then objects of type *T* in a program may be replaced with objects of type *S* without altering any of the desirable properties of that program (e.g. correctness).
- Behavioural subtyping is a stronger notion than typical subtyping of functions defined in type theory, which relies only on the contravariance of parameter types and covariance of the return type.
- Behavioural subtyping is undecidable in general: if *q* is the property "method for *x* always terminates", then it is impossible for a program (e.g. a compiler) to verify that it holds true for some subtype *S* of *T*, even if *q* does hold for *T*. Nonetheless, the principle is useful in reasoning about the design of class hierarchies.

Liskov substitution principle Contd…

- Liskov substitution principle imposes some standard requirements on signatures that have been adopted in newer object-oriented programming languages (usually at the level of classes rather than types:
  1. Contravariance of method parameter types in the subtype.
  2. Covariance of method return types in the subtype.
  3. New exceptions cannot be thrown by the methods in the subtype, except if they are subtypes of exceptions thrown by the methods of the supertype
- Within the type system of a programming language, a typing rule or a type constructor is:
  - ✓ *contravariant* if it reverses this ordering: If `A ≤ B`, then `I<B> ≤ I<A>`;
  - ✓ *covariant* if it preserves the ordering of types (≤), which orders types from more specific to more generic: If `A ≤ B`, then `I<A> ≤ I<B>`;
  - ✓ *bivariant* if both of these apply (i.e., if `A ≤ B`, then `I<A> ≡ I<B>`);
  - ✓ *variant* if covariant, contravariant or bivariant;
  - ✓ *invariant* or *nonvariant* if not variant.

Interface segregation principle (ISP)

- In the field of software engineering, the **interface-segregation principle** (**ISP**) states that no client should be forced to depend on methods it does not use.
- ISP splits interfaces that are very large into smaller and more specific ones so that clients will only have to know about the methods that are of interest to them. Such shrunken interfaces are also called *role interface*s.
- ISP is intended to keep a system decoupled and thus easier to refactor, change, and redeploy. ISP is one of the five SOLID principles of object-oriented design, similar to the High Cohesion Principle of GRASP.

Importance in object-oriented design

- Within object-oriented design, **interfaces** provide layers of abstraction that simplify code and create a barrier preventing coupling to dependencies.
- According to many software experts who have signed the Manifesto for Software Craftsmanship, writing well-crafted and self-explanatory software is almost as important as writing working software. Using interfaces to further describe the intent of the software is often a good idea.
- A system may become so coupled at multiple levels that it is no longer possible to make a change in one place without necessitating many additional changes. Using an interface or an abstract class can prevent this side effect.

Typical violation

- A typical violation of the Interface Segregation Principle is given in Agile Software Development: Principles, Patterns, and Practices in ATM Transaction example and in an article also written by Robert C. Martin specifically about the ISP.
- This example discusses the User Interface for an ATM, which handles all requests such as a deposit request, or a withdrawal request, and how this interface needs to be segregated into individual and more specific interfaces. [Please look into this use case for better understanding]

Dependency inversion principle

- In object-oriented design, the **dependency inversion principle** is a specific form of loosely coupling software modules.
- When following this principle, the conventional dependency relationships established from high-level, policy-setting modules to low-level, dependency modules are reversed, thus rendering high-level modules independent of the low-level module implementation details. The principle states:

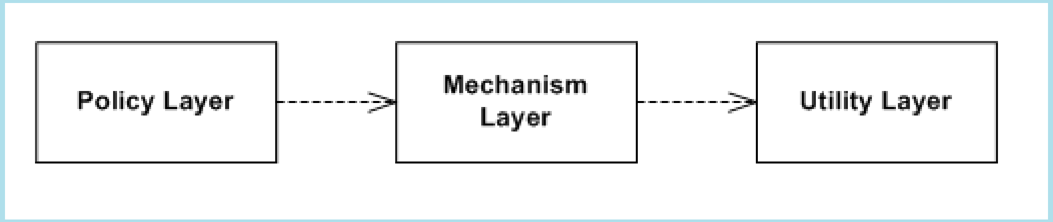
  A- High-level modules should not import anything from low-level modules. Both should depend on abstractions (e.g., interfaces).

  B- Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions.
- By dictating that *both* high-level and low-level objects must depend on the same abstraction, this design principle *inverts* the way some people may think about object-oriented programming.

- The idea behind points A and B of this principle is that when designing the interaction between a high-level module and a low-level one, the interaction should be thought of as an abstract interaction between them.
- This not only has implications on the design of the high-level module, but also on the low-level one: the low-level one should be designed with the interaction in mind and it may be necessary to change its usage interface.
- In many cases, thinking about the interaction in itself as an abstract concept allows the coupling of the components to be reduced without introducing additional coding patterns, allowing only a lighter and less implementation-dependent interaction schema.
- When the discovered abstract interaction schema(s) between two modules is/are generic and generalization makes sense, this design principle also leads to the following dependency inversion coding pattern.

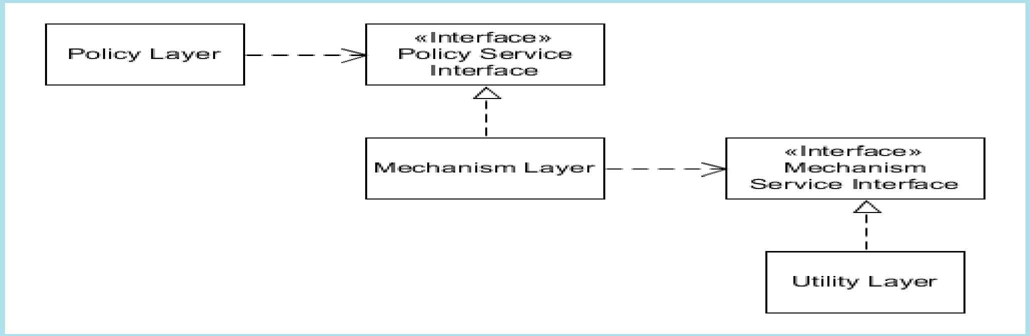
Traditional layers pattern

- In conventional application architecture, lower-level components (e.g., Utility Layer) are designed to be consumed by higher-level components (e.g., Policy Layer) which enable increasingly complex systems to be built.
- In this composition, higher-level components depend directly upon lower-level components to achieve some task.
- This dependency upon lower-level components limits the reuse opportunities of the higher-level components.



Dependency inversion pattern

- With the addition of an abstract layer, both high- and lower-level layers reduce the traditional dependencies from top to bottom.
- Nevertheless, the "inversion" concept does not mean that lower-level layers depend on higher-level layers directly.
- Both layers should depend on abstractions (interfaces) that expose the behavior needed by higher-level layers.


Policy Layer
«Interface» Policy Service Interface
Mechanism Layer
«Interface» Mechanism Service Interface
Utility Layer

- In a direct application of dependency inversion, the abstracts are owned by the upper/policy layers. This architecture groups the higher/policy components and the abstractions that define lower services together in the same package.
- The lower-level layers are created by inheritance/implementation of these abstract classes or interfaces.
- The inversion of the dependencies and ownership encourages the re-usability of the higher/policy layers. Upper layers could use other implementations of the lower services.
- When the lower-level layer components are closed or when the application requires the reuse of existing services, it is common that an Adapter mediates between the services and the abstractions.