# Design Principles

**Keywords to use:**

- Maintainability
- Code organization
- Ease of extendibility
- Ease of modification
- Less prone to bugs
- Less technical debt
- Flexibility
- Make overall coding process easier
- Better code readability

| Principle | Description | Application / Reasoning |
|---|---|---|
| Disciplined Exception Handling Principle | There are two legitimate response to exceptions:<br>1. **Retrying**. This is asking the user to re-input the values and execute the routine again.<br>2. **Failure**. This is when it executes the exception and reports failure to the caller. | **Design by contract** enforces this principle because clients may violate preconditions. When client violates the precondition, an exception is thrown and this principle is used as a method of handling the thrown exception, to retry or certify as failure. |
| Non-redundancy Principle | The body of a routine should not test the routine's precondition. The supplier/callee should not test the precondition. Client should check if precondition is met or not and handle the exceptions (because it's their responsibility as they violated the contract).<br><br>This is because a supplier is unable to know all possible inputs given by the clients. But it is possible for a client to catch and deal with the exception caused by a precondition violation. Precondition violation indicates a bug caused by client. | **Design by contract** enforces this to ensure that both client and supplier fulfills and do not violate the contract, but only act according to their responsibilities written in the contract. |
| Fail Fast | A system should fail immediately when something went wrong. This allows the developer to easily detect the issue and fix it immediately to avoid technical debt in the future and to avoid the problem hard to be discovered. | **Design by contract** enforces this by making use of assertions and exceptions to validate the preconditions and postconditions, it allows the system to fail fast. |

| | | |
|---|---|---|
| Liskov Substitution Principle | If class S is a subtype of class T, then S must conform to T such that an object of type T can be replaced with objects of type S and preserving the correctness of type T.<br><br>Example:<br>`T object = new S();`<br><br>When class S conforms to class T, then an object of class T is replaceable with an object of class S. This can be said because class S has fulfilled the contract that is established by class T. | **Design by contract** enforces this by making sure that the subclasses fulfills the contracts of their parents. This means that if the parent does "job X", the subclasses must also be able to do "job X". If fulfilled, the subclasses can be used to replace the parent.<br><br>Example:<br>`Parent job = new Child();` |
| Sound Subcontracting | If a class is subcontracted by a supplier, the client must not know this. In other words, when a class that is supposed to perform "task A" delegates this "task A" to its subclasses, the client must not know that this had happened because the contract is established between the client and the parent (not the subclasses).<br><br>A subclass can only **weaken** the preconditions of its parent.<br>A subclass can only **strengthen** the postcondition of its parent.<br>**must honor the contract of their parents** | A supplier can only delegate the task to its subclasses if and only if:<br><br>• the subclass preconditions are the same or weaker<br>⇒ it can accept less of its parent's precondition<br>⇒ if parent accept values > 0, then subclass must accept values > 0 or values > 10 (weaker condition)<br>⇒ this is to ensure the correctness of the parent's precondition after subcontracting to the subclass<br>• the subclass postconditions are the same or stronger<br>⇒ it can guarantee more of its parent's postcondition<br>⇒ if parent return values > 0, then subclass must return values > 0 or values > 10 (stronger condition)<br>⇒ this is to ensure the correctness of the parent's precondition after subcontracting to the subclass |
| Command-Query Separation Principle | Every method should either be a command or query.<br>• A command performs actions without changing the objects states<br>• A query returns a value with no side effects | **Design by contract** applies this by using query in precondition or postcondition to validify the state of the objects.<br><br>Example:<br>Using query like getters to access the attributes and to ensure that it won't change the values of the attributes (tighter control). |

| | | |
|---|---|---|
| Separation of Concerns | <u>Every program should be separated into sections or modules</u> with their own concern or responsibilities.<br><br><u>Each module should have a single and well-defined responsibilities</u> such that they overlap with other modules as little as possible. | This principle allows us to easily able to modify the code when new features are added.<br><br><u>Example</u>:<br>If the code for certain responsibilities are separated out, then adding a new feature will only need to modify the code that associates with it. If each responsibility is well defined, then |
| Dependency Inversion Principle | <u>High level</u> modules <u>should not depend on low level</u> modules, but rather <u>depend on abstractions</u>.<br><br>Abstractions should not depend on details, but details should depend on abstractions. | Concept of <u>**abstraction**</u> applies this.<br>A high-level module should depend / <u>associate with an interface (whenever possible)</u>, because it creates a flexibility for future modifications as <u>adding new feature can just implement the abstraction interface</u> without the need to modify the high-level module.<br><br>When a low-level module changes, the high-level module doesn't need to change because it is dependent on abstraction. This creates better reusability. |
| Single Responsibility Principle | Every <u>module or class should have responsibility over a single part of the functionality</u> provided by the software, and that responsibility should be entirely encapsulated by the class. | This is part of the bloaters in code smells. When <u>a class does not comply with this principle, that class may consists of many functionalities</u>.<br><br>This is considered as a bad practice because <u>when a new feature is added into that class, the refactoring process becomes complicated</u> because there are many dependencies going on such that changing one part will result in a change of many sections of the code, which may result in technical debt.<br><br>It promotes code organization, less dependencies, less prone to bugs, easier maintainability and debugging. |

| | | |
|---|---|---|
| Technical Debt | A piece of functionality is implemented quickly, but it will cause problems in the future. If the functionality is implemented slowly with a clean design, this implementation is said to be free of potential problems. | If the functionality is implemented quickly, is considered as a bad design because it does not promote code organization and it induces problems when new features are added in and the amount of refactoring will be huge as the problems are accumulated.<br><br>The best approach is always to refactor when adding new features.<br>This makes the system to be "flexible" for future amendments. |
| Avoid using instanceOf | Using instanceOf is considered as a bad practice because it creates a tight coupling between the callee and the class. In addition, it also makes it harder to extend and maintain because if a new subclass is added, we would have to search through the entire system and add a new instanceOf of that class, which takes a longer time. This will also result in a lot of duplicated "instanceOf" which makes it ugly (readability suffers).<br><br>*hard to maintain + readability suffers* | The general solution to instanceOf is to use polymorphism. Setting up an abstract method in the superclass and allow the subclasses to override this method and execute their differences. This prevents repeated "instanceOf" code because the developer can just call the abstract method from the superclass without down-casting it. |
| Avoid Excessive Use of Literals | Embedded constants are bad because if the value needs to change in the future, developer have to hunt for every place and change them accordingly. | |
| Don't Repeat Yourself | Duplicated code is bad because the same code needs to be maintained in multiple places. If a bug is discovered in that code, every piece of duplicate must be reviewed and fixed, which is time consuming. | |
| Reduce Dependencies as Much as Possible | When classes is dependent on another class's method, a change in the logic of the class's method will subsequently need to refactor all dependent classes, which makes refactoring to be expensive. | |
| Declare Things in the Tightest Possible Scope | Tighter scope reduces the risk where a class can depend on it and thus reduces the amount of possible failures in the future. Because classes have less dependencies on one another, so when a class is modified, it will not affect other classes hence reduce the amount of refactoring. | |

| | | |
|---|---|---|
| Group Elements that Must Depend on each other Inside an Encapsulated Boundary | If dependencies can't be removed, and there are no other options to be not dependent on, then we could use the concept of packages to encapsulate them within a "boundary". The classes that must depend on each other can be put into a package and setting the relevant dependent fields/methods as protected. This gives only the classes in the same package or the subclasses on having access over the details. There is an "encapsulated boundary" over the "package" here because classes "outside" unable to view the details "inside" the package (as fields are set as protected). | |
| Minimize Dependencies that Cross Encapsulated Boundaries | This can be done by using access modifiers to remove dependencies across encapsulated boundary, like setting an attribute as private. Doing so makes the classes "outside" unable to see the internal fields/methods "inside", creating a tighter control on dependencies. Without doing so, the fields are visible from other classes hence giving chance for them to break the system. | |
| Avoid Variables with Hidden Meanings | Avoid variables that carry more than one meaning. For instance a variable "customerId" that stores integer of "-1" to signify invalid customer while customer that are more than 0 signify valid customer. Such variable is said to carrying two meanings (overloading the variable) | |

# Abstraction and Encapsulation

## Abstraction

Abstraction is the quality of dealing with ideas rather than concrete events. It hides the implementation details and only provide the functionality to the user by providing a more abstract picture. It focuses on what the object does rather than how the object does. This is achieved by providing a generalized view on the classes through interfaces and abstract classes.

## Benefits of abstraction

- makes future development easier (can just extend, as the implementation details are hidden)
- reduce technical debt (classes are now easier to organize)
- reduces coupling effect

## Encapsulation

Encapsulation is a mechanism of wrapping the data (variables) and code (methods) together as a single unit. In encapsulation, information is hidden from the user through the use of access modifiers (like private) to restrict the user's accessibility and only able to access what is needed through setters, getters and public methods.

## Benefits of encapsulation

- reduces coupling effect as there exists some restrictions between classes
- makes code flexible and easy to change with new requirements

| | **Abstraction** | **Encapsulation** |
|---|---|---|
| **Definition** | Hides implementation details and only provide the functionality to the user by providing a more generalized abstract view. | Wraps the data and code together as a single unit |
| **Functionality / Focus** | Highlights only essential features to make complex programs simpler (like only know that there an abstract method but it does not know the details, aka what the method does)<br><br>Hides implementation details on design level (aka how classes interact) | Binding data and codes into a single unit to prevent external access and access only what is needed (like using setters and getters to create tighter control)<br><br>Hides details on implementation level (aka using private modifier when implementing something) |
| **Implementation** | Using abstract classes and interfaces | Using access modifiers to encapsulate data |
| **Concept** | Focuses on what the object does rather than how the object does | Hides the internal mechanics of how it does something |

## Interface and Abstract Class

| | Advantages | Disadvantages |
|---|---|---|
| **Interfaces** | • able to implement multiple inheritance<br>• make objects become loosely coupled<br>• achieve full abstraction<br>• break up complex design and clear dependencies between objects<br>• can group common behaviors by unrelated objects (aka implemented classes) | • unable to have constructor<br>• attributes are by default public, static and final, so interfaces can’t store instance fields but only constant class fields |
| **Abstract class** | • able to store both abstract methods and concrete methods<br>• able to have attributes<br>• provide a commonality as a superclass for others to inherit<br>• can group common behaviors by related objects (aka subclasses) | • unable to extend multiple classes<br>• cannot be instantiated |

## JavaDoc

- able to cleanly separate interface from implementation
- use code by looking at the interface without having to read the source code, save time
- avoid users from depending on the implementation details that maintainers wish to change in the future
- the developer needs to provide a high-quality documentation of the interface to the users
- documentation is usually done through by placing a block comments “/*” above the method
- keys of documentation:
  - comprehensive (complete without missing parts)
  - consistent to the actual behavior
  - clearly and consistently presented (documentation without vagueness)
- information must be sufficiently documented so that other programmers can extend or use this library as their development
- also known as a contract because the users only need to look at the interface and comply according to the precondition and postcondition specified
- JavaDoc comments can be attached to:
  - Package
  - Class
  - Interface
  - Public class members, including attributes, methods and constructors
- Private methods and attributes are usually not part of JavaDoc because they are part of the implementation, not the interface. The comments written there are for maintainers to read not the interface users

## Up-casting and Down-casting

- introduce possible bugs that won’t caught during compile time (only during runtime)
- create errors when a class is not part of any inheritance that can’t up or down-cast
- use interfaces so that signature methods are known without down-casting (polymorphism)
- or use abstract methods in abstract class to avoid down-casting (polymorphism)

## Assertions and Exceptions

**Assertions**

- used to assert something to be true at a certain point of the code
- used to verify the internal logic of the code
- like verifying the postcondition

**Exceptions**

- used when something goes wrong

| | **Checked Exception** | **Unchecked Exception** |
|---|---|---|
| **Verification** | During compile time | During runtime |
| **Properties** | 1. the signature method declares the exceptions that it throws<br>• alerts the clients of the method that it is possible for this method to fail and have the chance to catch the exception and “do something”.<br>2. using try/catch to catch the checked exceptions<br>• like if a user inputs wrong file, then catch FileNotFoundException and prompt the user to re-input. | • exceptions that are not specified in the signature method<br>• used in situations where it is not reasonable for clients to recover from it or handle the exception<br>• this is usually the result of programming error rather than user error or external environment |
| **Used in** | Situations where client can reasonably expect to recover from an exception<br>(like enter wrong filename) | Situations where client cannot do anything to recover from the exception<br>(like attribute is null – NullError<br>or division by zero – Arithmetic) |

## Privacy Leak

- when someone outside gets a copy of an object meant to be securely inside
- this means that a client can modify the referenced value of a private attribute of the class
- this can be improved by modifying the getter to return a new copy of the reference instead of a direct reference to the private attribute
- if the attribute is mutable (i.e., lists), then set the getters to return a new copy of the reference instead of a direct reference to the attribute, this is called “defensively copy”
- privacy leak is a sign where data is not encapsulated properly to the extend where they are able to cross the encapsulated boundary for others to view (i.e., visible and modifiable from other classes)