Principles of Object-Oriented Design

As a part of an overall strategy of agile and adaptive programming, a number of object-oriented design principles were proposed for the design and programming of computer software system that is easy to maintain and extend over time. These principles are guidelines intended for programmers to apply while working on software to remove "code smells" (potentially buggy code) by refactorizing the source code until it is both legible and extensible. In this page, we introduce the **SOLID** principles, that is, Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion. The following information was integrated from various sources on the Web.

Single Responsibility Principle (SRP)

The SRP requires that a class should have only a single responsibility.

Example: If a class `SalesOrder` keeps information about a sales order, and in addition has a method `saveOrder()` that saves the `SaleOrder` in a database and a method `exportXML()` that exports the `SalesOrder` in XML format, this design will violate the SRP because there will be different types of users of this class and different reasons for making changes to this class. A change made for one type of user, say change the type of database, may require the re-test, recompilation, and re-linking of the class for the other type of users.

A better design will be to have the `SalesOrder` class only keeps the information about a sales order, and have different classes to save order and to export order, respectively. Such a design will confirm to SRP.

Open-Closed Principle (OCP)

The OCP requires that each software entity should be open for extension, but closed for modification.

Example: Suppose an `OrderValidation` class has a method `validate(Order order)` that is programmed to validate an order based on a set of hard-coded rules. This design violates the OCP because if the rules change, the `OrderValidation` class has to be modified, tested, and compiled.

A better design will be to let the `OrderValidation` class contain a collection of `ValidationRule` objects each of which has a `validate(Order order)` method (perhaps defined in a `Validation` interface) to validate an `Order` using a specific rule, and the `validate(Order order)` method of `OrderValidation` class can simply iterate through those `ValidationRule` objects to validate the order. The new design will satisfy the OCP, because if the rules change, we can just create a new `ValidationRule` object and add it to an `OrderValidation` instance at run time (rather than to the class definition itself).

This is can also be achieved by using subclasses of a base class `AbstractValidationRule` that has an override-able function `validate(Order order)`. Subclasses can implement the method differently without changing the base class functionality.

Liskov Substitution Principle (LSP)

The LSP requires that objects in a program should be replaceable with instances of their subclasses without altering the correctness of that program.

The users must be able to use objects of subclasses via references to base classes without noticing any difference. When using an object through its base class interface, the object of a subclass must not expect the user to obey preconditions that are stronger than those required by the base class.

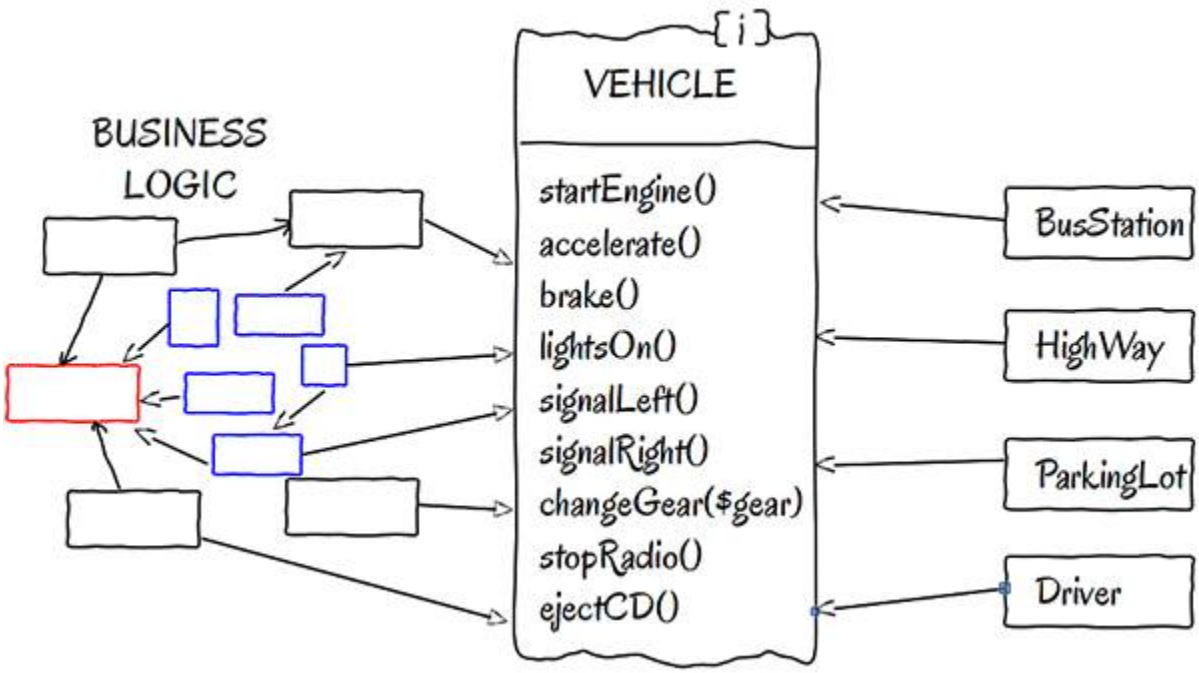
Example: Suppose a `Rectangle` class has two instance variables `height` and `width`, and a method `setSize(int a, int b)`, which set `height` to `a` and `width` to `b`. Suppose `Square` is a subclass of `Rectangle` and it overrides the inherited method by setting both `height` and `width` to `a`. This design will violate LSP. To see this, consider a client uses a reference variable of type `Rectangle` to call the `setSize()` method to assign different values of `a` and `b`, and then immediately verify if the sizes were set correctly or the area is correctly computed. The results will be different if the variable references to a `Rectangle` object than to a `Square` object.

It turns out that in OO programming, a `Square` is not a `Rectangle` at all because it behaves differently from a `Rectangle`.

Interface Segregation Principle (ISP)

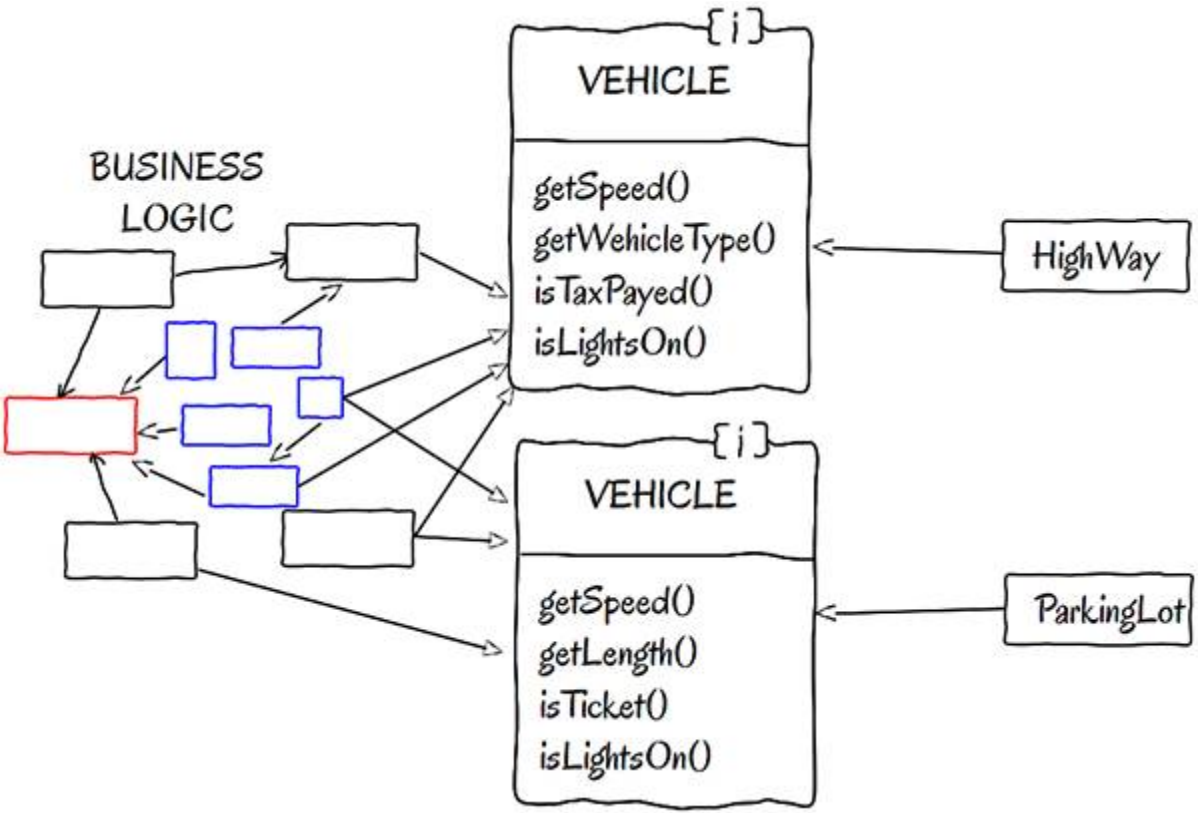
The ISP requires that clients should not be forced to depend on interfaces that they do not use.

Example: Suppose a Vehicle interface shown in the figure is designed for clients to use



This violates ISP because clients are forced to depend on methods they do not use: `HighWay` does not use `stopRadio()` or `ejectCD()`, and `ParkingLot` does not need `accelerate()` or `ejectCD()`.

A better design is to design smaller interfaces for different types of clients as shown in the following figure



Dependency Inversion Principle (DIP)

The DIP requires that high level modules should not depend on low level modules, both should depend on abstraction. Also, abstraction should not depend on details, details should depend on abstractions.

Example: Making a class `Button` associate to another class `Lamp` (because a `Lamp` has a `Button`) is a violation of DIP. A better design will be associate an AbstractButton with an AbstractButtonClient, and define Button as a subclass of the AbstractButton and a Lamp a subclass of the AbstractButtonClient.

Example: Making an `EBookReader` class to use `PDFBook` class is a violation of DIP because it requires to change the `EBookReader` class to read other types of e-books. A better design is to let `EBookReader` use an interface `EBook` and let `PDFBook` and other types of e-book classes implement `EBook`. Now adding or changing e-book classes will not require any change to `EBookReader` class.