SOLID is one of the most popular sets of design principles in object-oriented software development. It’s a mnemonic acronym for the following five design principles:

1. Single Responsibility Principle

A class should have one, and only one, reason to change.

Benefits of the Single Responsibility Principle

Frequency and Effects of Changes

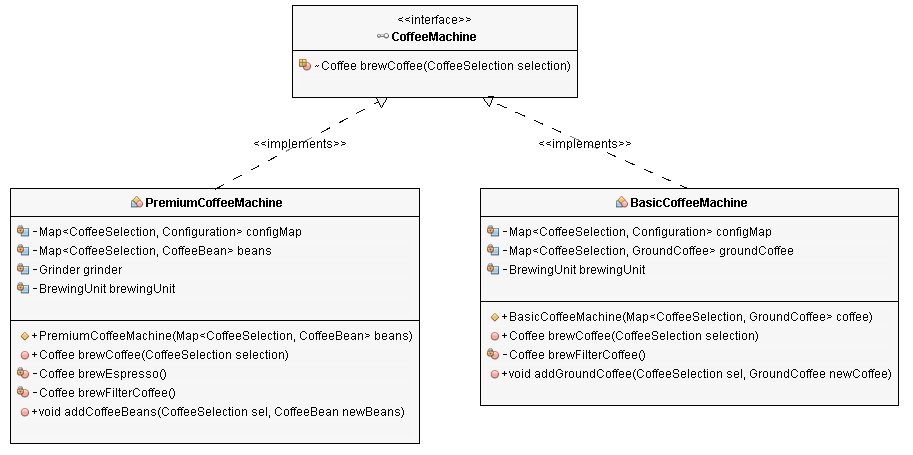
Easier to Understand

2. Open/Closed Principle

“Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification. 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.”

It promotes the use of interfaces to enable you to adapt the functionality of your application without changing the existing code which enables loose coupling.

The interfaces are closed for modifications, and you can provide new implementations to extend the functionality of your software.

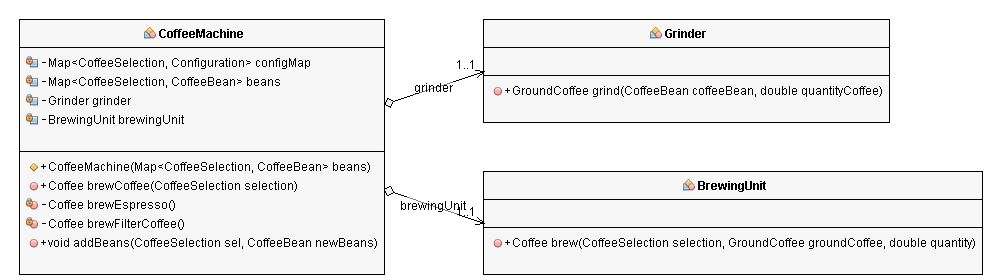


Composition (has-a):

It describes a class that references one or more objects of other classes in instance variables. This allows you to model a has-a association between objects.

You can find such relationships quite regularly in the real world. A car, for example, has an engine and modern coffee machines often have an integrated grinder and a brewing unit.

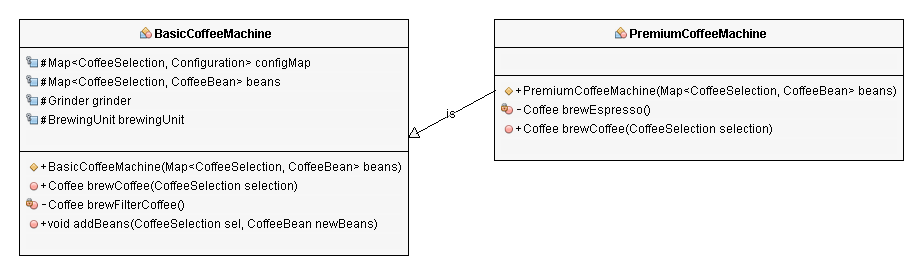
The CoffeeMachine class models a modern coffee machine with an integrated grinder and a brewing unit. In the real world, these two elements are parts of the coffee machine and can’t be separated. You also don’t interact with them directly. You always use them via the interface of the coffee machine. This interface only you gives you access to the operations that are required to brew a coffee and hides every other detail.



Inheritance (is-a):

It is a mechanism where you can to derive a class from another class for a hierarchy of classes that share a set of attributes and methods.

You can use it to declare different kinds of exceptions, add custom logic to existing frameworks, and even map your domain model to a database.



But what happens when you replace your *BasicCoffeeMachine*? You might get a better one with an integrated grinder, which can brew more than just filter coffee. Unfortunately, the *CoffeeApp* doesn’t support this kind of coffee machine.

3. Liskov Substitution Principle

 It extends the [Open/Closed principle](https://stackify.com/solid-design-open-closed-principle/) and enables you to replace objects of a parent class with objects of a subclass without breaking the application. This requires all subclasses to behave in the same way as the parent class. To achieve that, your subclasses need to follow these rules:

* Don’t implement any stricter validation rules on input parameters than implemented by the parent class.
* Apply at the least the same rules to all output parameters as applied by the parent class.

4.Interface Segregation Principle

“Clients should not be forced to depend upon interfaces that they do not use.”

By following this principle, you prevent bloated interfaces that define methods for multiple responsibilities. As explained in the [Single Responsibility Principle](https://stackify.com/solid-design-principles/), you should avoid classes and interfaces with multiple responsibilities because they change often and make your software hard to maintain.

5. Dependency Inversion

As you have seen in the example project, you only need to consequently apply the Open/Closed and the Liskov Substitution principles to your code base. After you have done that, your classes also comply with the Dependency Inversion Principle. This enables you to change higher-level and lower-level components without affecting any other classes, as long as you don’t change any interface abstractions.