**Composition - Coffee Machine Example**

Composition is one of the [fundamental concepts in object-oriented programming](https://stackify.com/oops-concepts-in-java/). 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.

**Main benefits of composition**

Given its broad use in the real world, it’s no surprise that composition is also commonly used in carefully designed software components. When you use this concept, you can:

1. reuse existing code
2. [design clean APIs](https://theamiableapi.com/2012/01/16/java-api-design-checklist/)
3. change the implementation of a class used in a composition without adapting any external clients

**Reuse existing code**

The main reason to use composition is that it allows you to reuse code without modeling an *is-a* association as you do by using [inheritance](https://stackify.com/oop-concept-inheritance/). That allows stronger encapsulation and makes your code easier to maintain as Joshua Bloch explains in the 3rd edition of his book [Effective Java](https://www.thoughts-on-java.org/review-effective-java-3rd-edition/).

The concept of composition is often used in the real world, and it should be the same in software development. A car is not an engine; it has one. And a coffee machine has a grinder and a brewing unit, but it is none of them. The car and the coffee machine integrate an engine, grinder and brewing unit via their external APIs to compose a higher level of abstraction and provide more significant value to their users.

You can do the same in software development when you design a class to keep a reference to an object and to use it in one or more of its methods.

**Design a clean API**

This also enables you to design clean and easy-to-use APIs. When you compose a class, you can decide if the referenced classes become part of the API or if you want to hide them.

As I explained in my article about [encapsulation](https://stackify.com/oop-concept-for-beginners-what-is-encapsulation/), Java supports different access modifiers. It’s a common best practice to use the *private* modifier for all attributes, including the ones that reference other objects, so that it can only be accessed within the same object. If you want to allow external access to an attribute, you need to implement a getter or setter method for it.

But that’s not the only thing you can do to create a clean API. If you use no access modifiers for a class, it becomes package-private. This class can’t be accessed outside of its own package and is not part of the API. External clients of your software component are not aware of this class. They can only use it via a public class that uses the package-private class in a composition.

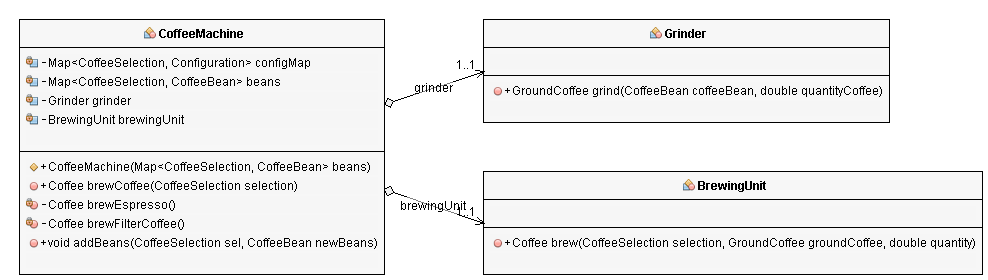
Let’s take a look at an example.

**API design in the CoffeeMachine example**

I use composition in the CoffeeMachine project that you might already know from the previous posts of this series. You can clone it at <https://github.com/thjanssen/Stackify-OopAbstraction>.

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.

That design works pretty well in the real world. Let’s try the same approach for our example application.



***Grinder* and *BrewingUnit* as internal classes**

The *Grinder* and *BrewingUnit* classes are package-private and can’t be accessed from the outside. You will not even see them, as long as you’re not adding a class to the package of my application.

class Grinder {

public GroundCoffee grind(CoffeeBean coffeeBean, double quantityCoffee) {

return new GroundCoffee();

}

}

class BrewingUnit {

public Coffee brew(CoffeeSelection selection, GroundCoffee groundCoffee, double quantity) {

return new Coffee(selection, quantity);

}

}

**The *CoffeeMachine* class defines the public API**

The public methods of the *CoffeeMachine* class define the main part of the API of my small sample application. These are:

* a constructor that expects a *Map* of coffee beans,
* the *addBeans* method which enables you to refill coffee beans or to add different ones
* the *brewCoffee* method that you can call with a *CoffeeSelection* to brew a cup of filter coffee or espresso

import java.util.HashMap;

import java.util.Map;

public class **CoffeeMachine** {

private Map<**CoffeeSelection**, **Configuration**> configMap;

private Map<**CoffeeSelection**, **CoffeeBean**> beans;

private **Grinder** grinder;

private **BrewingUnit** brewingUnit;

public **CoffeeMachine**(Map<CoffeeSelection, CoffeeBean> beans) {

this.beans = beans;

this.grinder = new Grinder();

this.brewingUnit = new BrewingUnit();

this.configMap = new HashMap<CoffeeSelection, Configuration>();

this.configMap.put(CoffeeSelection.ESPRESSO, new Configuration(8, 28));

this.configMap.put(CoffeeSelection.FILTER\_COFFEE, new Configuration(30, 480));

}

public Coffee **brewCoffee**(CoffeeSelection selection) throws CoffeeException {

switch (selection) {

case FILTER\_COFFEE:

return brewFilterCoffee();

case ESPRESSO:

return brewEspresso();

default:

throw new CoffeeException("CoffeeSelection [" + selection + "] not supported!");

}

}

private Coffee brewEspresso() {

Configuration config = configMap.get(CoffeeSelection.ESPRESSO);

// grind the coffee beans

GroundCoffee groundCoffee = this.grinder.grind(

this.beans.get(CoffeeSelection.ESPRESSO), config.getQuantityCoffee());

// brew an espresso

return this.brewingUnit.brew(

CoffeeSelection.ESPRESSO, groundCoffee, config.getQuantityWater());

}

private Coffee brewFilterCoffee() {

Configuration config = configMap.get(CoffeeSelection.FILTER\_COFFEE);

// grind the coffee beans

GroundCoffee groundCoffee = this.grinder.grind(

this.beans.get(CoffeeSelection.FILTER\_COFFEE), config.getQuantityCoffee());

// brew a filter coffee

return this.brewingUnit.brew(

CoffeeSelection.FILTER\_COFFEE, groundCoffee, config.getQuantityWater());

}

public void addBeans(CoffeeSelection sel, CoffeeBean newBeans)

throws CoffeeException {

CoffeeBean existingBeans = this.beans.get(sel);

if (existingBeans != null) {

if (existingBeans.getName().equals(newBeans.getName())) {

existingBeans.setQuantity(existingBeans.getQuantity() + newBeans.getQuantity());

} else {

throw new CoffeeException(

"Only one kind of beans supported for each CoffeeSelection.");

}

} else {

this.beans.put(sel, newBeans);

}

}

}

The *Grinder* and *BrewingUnit* objects can’t be accessed outside of their package. So, I need to instantiate them within the *CoffeeMachine* class. As you can see in the code snippet, I do that in the constructor method. This approach also allows you to control the usage of these objects used within the composition.

I use both of them in the *brewFilterCoffee* and the *brewEspresso* methods. That allows me to reuse the code provided by the two classes without exposing them to any client. And it also prevents any misuse of these objects. The *CoffeeMachine* class can validate the *CoffeeBean*s that are provided to the *grind* method of the *Grinder*and the kind of coffee and quantity of water that gets used to brew a selected coffee.

As you can see, composition allows you to design an API that’s easy and safe to use by encapsulating the details of your composition.

**Hide internal code changes**

Using composition and encapsulation not only enables you to create better APIs, but you can also use it to make your code easier to maintain and modify. As long as a class gets only used by your own code, you can easily change it and adapt any client code if necessary.

**Change internal classes without side effects**

The *Grinder* class in the *CoffeeMachine* example is not visible to any external clients. So, I could decide to change the signature of the *grind* method, or I could add additional methods to the Grinder class without worrying about any external side effects.

Let’s do that and add a *CoffeeSelection* as another parameter to the *grind* method. The *Grinder* can then select different coarseness settings for a filter coffee and an espresso.

class Grinder {

public GroundCoffee grind(CoffeeBean coffeeBean, double quantityCoffee, CoffeeSelection selection) {

return new GroundCoffee(selection);

}

}

In this example, it’s just a simple change to the *Grinder* class. But that’s only the case because that class is package-private and used in a composition that doesn’t leak any information. I don’t need to worry about backward compatibility or how my changes might affect any code that uses the *Grinder* class.

**Handle the changes internally**

I just need to adjust the *CoffeeMachine* class because I know it’s the only class that uses the *Grinder*. The required change is simple. I just change the calls of the *grind*method in the *brewFilterCoffee* and *brewEspresso* methods and provide a *CoffeeSelection* as the third parameter.

import java.util.HashMap;

import java.util.Map;

public class CoffeeMachine {

private Map<CoffeeSelection, Configuration> configMap;

private Map<CoffeeSelection, CoffeeBean> beans;

private Grinder grinder;

private BrewingUnit brewingUnit;

public CoffeeMachine(Map<CoffeeSelection, CoffeeBean> beans) {

this.beans = beans;

this.grinder = new Grinder();

this.brewingUnit = new BrewingUnit();

this.configMap = new HashMap<CoffeeSelection, Configuration>();

this.configMap.put(CoffeeSelection.ESPRESSO, new Configuration(8, 28));

this.configMap.put(CoffeeSelection.FILTER\_COFFEE,

new Configuration(30, 480));

}

public Coffee brewCoffee(CoffeeSelection selection) throws CoffeeException {

switch (selection) {

case FILTER\_COFFEE:

return brewFilterCoffee();

case ESPRESSO:

return brewEspresso();

default:

throw new CoffeeException(

"CoffeeSelection [" + selection + "] not supported!");

}

}

private Coffee brewEspresso() {

Configuration config = configMap.get(CoffeeSelection.ESPRESSO);

// grind the coffee beans

GroundCoffee groundCoffee = this.grinder.grind(

this.beans.get(CoffeeSelection.ESPRESSO), config.getQuantityCoffee(), CoffeeSelection.ESPRESSO);

// brew an espresso

return this.brewingUnit.brew(

CoffeeSelection.ESPRESSO, groundCoffee, config.getQuantityWater());

}

private Coffee brewFilterCoffee() {

Configuration config = configMap.get(CoffeeSelection.FILTER\_COFFEE);

// grind the coffee beans

GroundCoffee groundCoffee = this.grinder.grind(

this.beans.get(CoffeeSelection.FILTER\_COFFEE),

config.getQuantityCoffee(), CoffeeSelection.FILTER\_COFFEE);

// brew a filter coffee

return this.brewingUnit.brew(

CoffeeSelection.FILTER\_COFFEE, groundCoffee, config.getQuantityWater());

}

public void addBeans(CoffeeSelection sel, CoffeeBean newBeans)

throws CoffeeException {

CoffeeBean existingBeans = this.beans.get(sel);

if (existingBeans != null) {

if (existingBeans.getName().equals(newBeans.getName())) {

existingBeans.setQuantity(existingBeans.getQuantity() + newBeans.getQuantity());

} else {

throw new CoffeeException(

"Only one kind of beans supported for each CoffeeSelection.");

}

} else {

this.beans.put(sel, newBeans);

}

}

}

As you can see, the change of the *Grinder* class doesn’t have any effect on the API. The composition and encapsulation provided by the *CoffeeMachine* class hide the changes. That makes them a lot easier to implement and improves the maintainability of the example project.

Summary

Composition is one of the key concepts of object-oriented programming languages like [Java](https://stackify.com/content/java/). It enables you to reuse code by modeling a *has-a* association between objects.

If you combine the concept of composition with the encapsulation concept, you can exclude the reused classes from your API. That enables you to implement software components that are easy to use and maintain.