Open/Closed Principle

Have you ever driven a car? Even if you've never tried it, you have an idea of how to do it. Every car behaves similarly, so a driver knows what to do even in a new one. What will happen if we add more pedals, or swap gas and brake pedals?

Would it be safe to drive this car? Obviously not! It's impossible to modify the control principles of a car without any consequences.
Let's learn how to design programs for a long time to come as someone's already made it for cars.

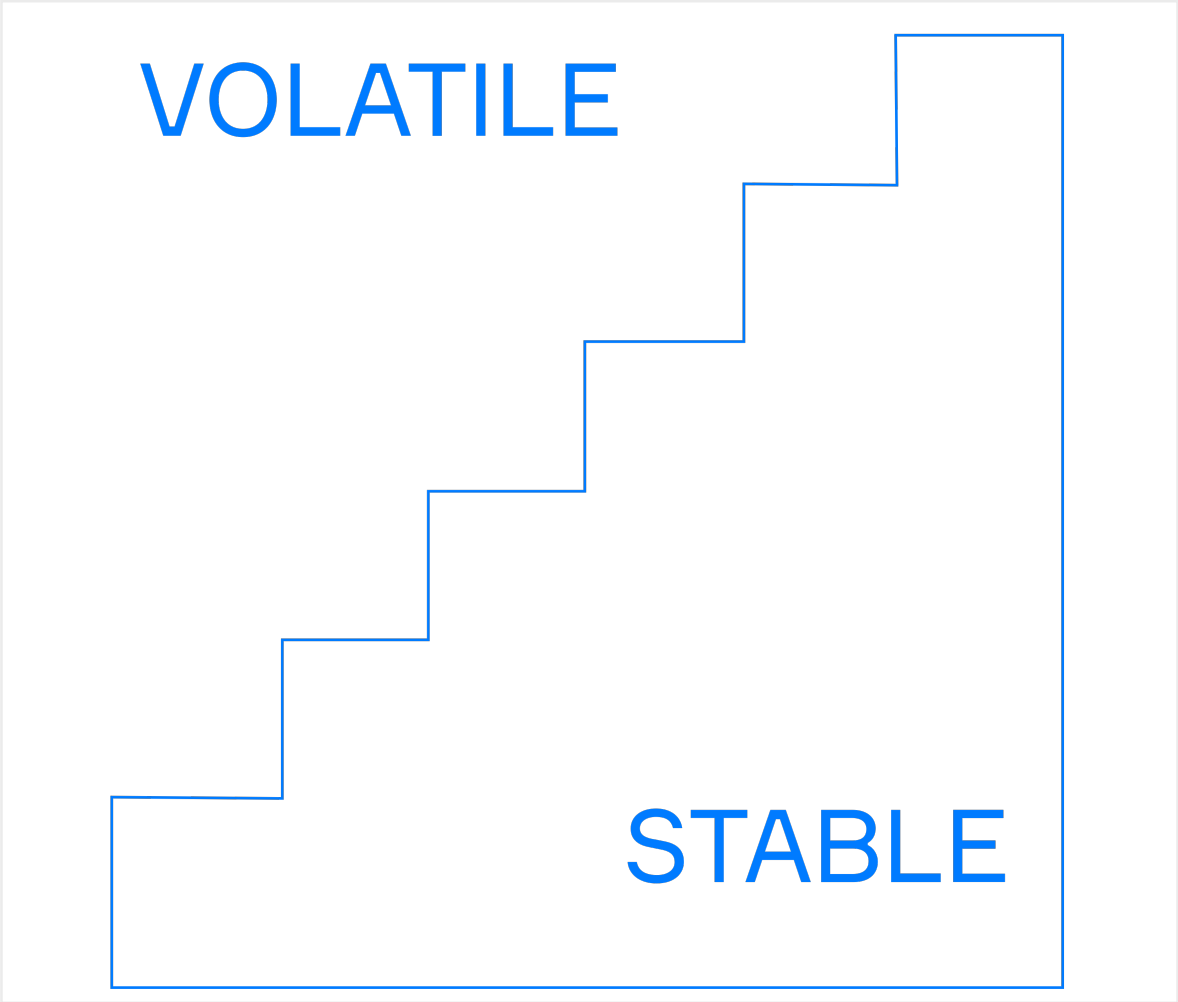
Old vs New problem

There are two contradicting demands we often face in software developing:

- We want to add new features to programs
- We don't want to break the compatibility with the older versions

It's not always a problem to add a new feature or to change the working principle of a component.
It depends on how many other components rely on it. We may have a function that we use a lot, so if we decide to change it, we should check all the places where it's called. A lot of work to do! Comparatively, it's easier to change rarely used functions.

Think of code components as of a staircase: we should have a solid foundation of those components that we use a lot, and we can easily change the upper parts because they are less dependent on the components below.



If your code is a library or a reusable module, do not forget that other developers will expect it to be a solid foundation for their code.

Open/Closed Principle

Imagine we need to make some changes after everything's already finished. It could be rather difficult, especially when an important part of the code needs to be modified, as it would affect other parts.

Let's create a HOUSE class and try to plug electrical devices in it. This class contains WIRING and DEVICE objects. The wiring is the basis for all other electrical elements. If we change the way the wiring works, it can lead to changes for all other elements, but we hardly want to replace our favorite devices! We should *close* the wiring from the changes.

We also need to add new devices that may change very often. That's the reason why every year we buy new stuff for our houses.
Then we need to create an interface, so we could connect the devices to the wiring.

Programming to an interface

The simpler the method, the simpler the maintenance. We should do programming to an interface, not to an implementation.

Interfaces in programming serve as contracts to exchange information between separate components. You can think of an interface as a stable collection of methods that an object should satisfy.

To make the wiring work with any device, we will use POWER_SOCKET objects. The next step would be to implement the PLUG_IN method, so the device can work with the wiring. If an object has this method, we can plug it into the network.

```
Interface pluggable {
        void PLUG_IN(POWER_SOCKET);
}
```

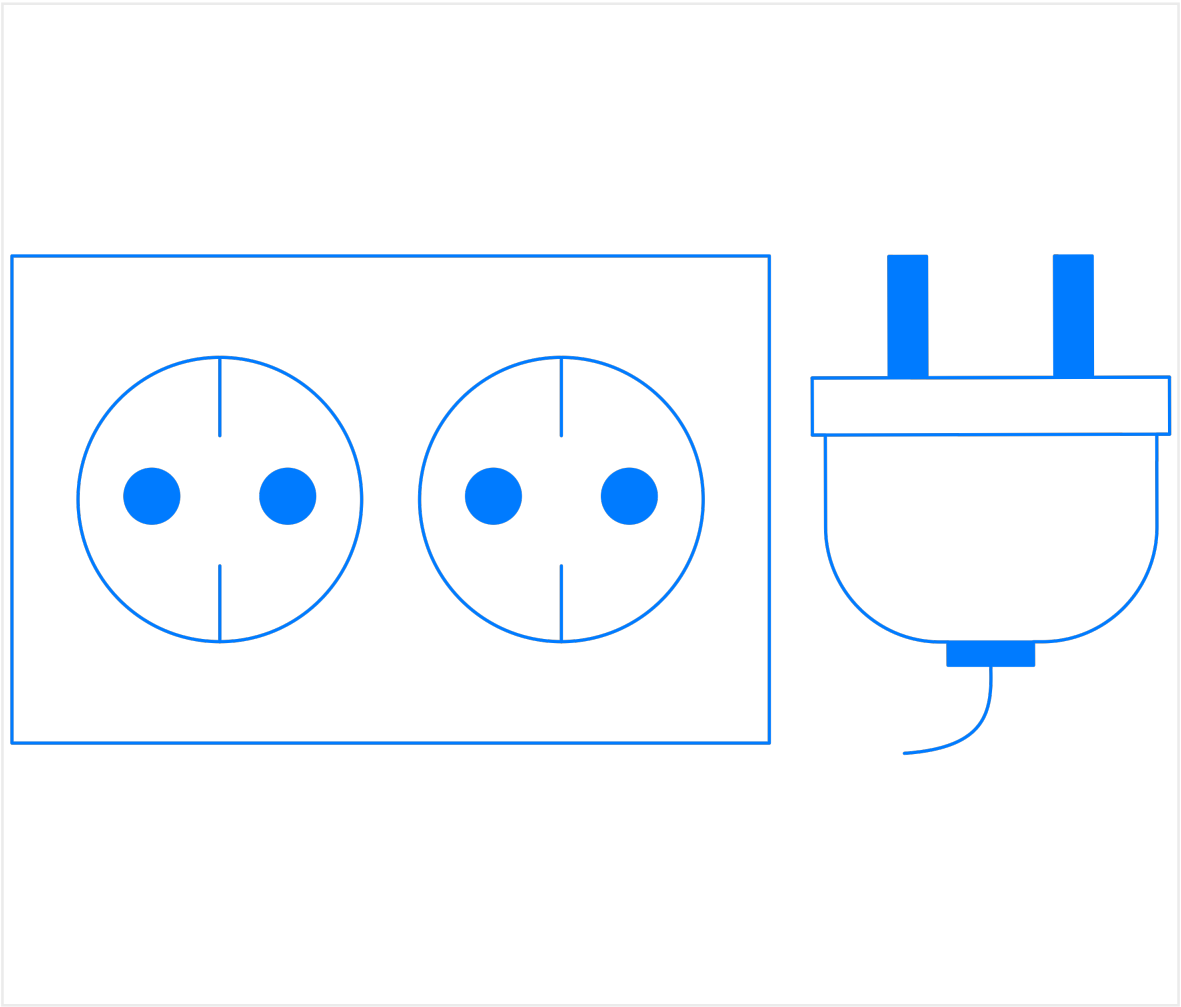
We create an interface with one method, and we just need the devices to work with power sockets. It's a fair contract because there are many devices and only one source of power in the house.

From now on, we can broaden the number of devices that work with the wiring in the house. There's no need to change the wiring, as the new devices will *extend* the whole network.

Reuse and inherit

When several objects working with the piece of your code are similar to each other, you can use interfaces, but what would you do if you buy a device with some foreign electrical plug? It's incompatible with the power sockets in your house because it implements the PLUG_IN method to a different type of socket.

In this case, we can use two different strategies. One is *inheritance*, and the second is *reusing* already working elements. In each option we will use another element of an electrical network: ADAPTER.



Our FOREIGN_DEVICE has the PLUG_IN method, but it only works with foreign power sockets. We can inherit the FOREIGN_DEVICE and redefine PLUG_IN method to work through an ADAPTER. We do not change the electrical network and even the device itself, but now it can work with both local and foreign types of power sockets, isn't it great?

We virtually combine the work of an adapter and the device to the new entity. Now we can use this entity directly with the wiring.

The second strategy is to reuse the capabilities of ADAPTER. We can implement the PLUG_IN method, and it will work with wiring as a DEVICE. On the other hand, we can use it as a POWER_SOCKET for a FOREIGN_DEVICE. We wrap its two-way capabilities, not creating any new interfaces or classes in our code but still solving the problem. Our code stays almost the same, but the network *extends*.

We create a chain to make the device work. We don't change anything, we plug the device into an adapter and plug the adapter into a power socket.

1)

```
// Interface representing a pluggable device
interface Pluggable {
    void plugIn(PowerSocket socket);
}

// Class representing a foreign device
class ForeignDevice implements Pluggable {
    // Implementation of the plug-in method for foreign power sockets
    @Override
    public void plugIn(PowerSocket socket) {
        // Logic specific to plugging into a foreign power socket
        System.out.println("Plugging into foreign power socket...");
    }
}

// Class representing an adapter that adapts the foreign device to work with
local power sockets
class Adapter extends ForeignDevice {
    // Redefine the plug-in method to work with local power sockets
    @Override
    public void plugIn(PowerSocket socket) {
        // Logic to adapt the foreign device to work with local power sockets
        System.out.println("Adapting and plugging into local power socket...");
    }
}

// Power socket class
class PowerSocket {
    // Implementation details of power socket
}

// Main class to demonstrate the scenario
public class Main {
    public static void main(String[] args) {
        // Create an instance of the adapter
        Adapter adapter = new Adapter();

        // Plug the adapter into a local power socket
        PowerSocket localSocket = new PowerSocket();
        adapter.plugIn(localSocket);
    }
}
```

2)

```
// Interface representing a pluggable device
interface Pluggable {
    void plugIn(PowerSocket socket);
}

// Class representing an adapter that can act as both a power socket and a
pluggable device
class Adapter implements Pluggable {
    // Implementation of the plug-in method for plugging into a power socket
    @Override
    public void plugIn(PowerSocket socket) {
        // Logic to plug into a power socket
        System.out.println("Adapter plugged into a power socket...");
    }

    // Implementation of the plug-in method for plugging a device into the
adapter
    public void plugIn(Pluggable device) {
        // Logic to plug a device into the adapter
        System.out.println("Device plugged into the adapter...");
    }
}

// Class representing a foreign device
class ForeignDevice implements Pluggable {
    // Implementation of the plug-in method for the foreign device
    @Override
    public void plugIn(PowerSocket socket) {
        // Logic specific to plugging into a foreign power socket
        System.out.println("Foreign device plugged into a power socket...");
    }
}

// Power socket class
class PowerSocket {
    // Implementation details of power socket
}

// Main class to demonstrate the scenario
public class Main {
    public static void main(String[] args) {
        // Create an instance of the adapter
        Adapter adapter = new Adapter();

        // Plug a foreign device into the adapter
        ForeignDevice foreignDevice = new ForeignDevice();
        adapter.plugIn(foreignDevice);
```

```
        // Plug the adapter into a power socket
        PowerSocket powerSocket = new PowerSocket();
        adapter.plugIn(powerSocket);
    }
}
```

Why does the open/closed principle matter?

If you're working with a small piece of code, there may be no patterns and regularities in it yet. Continue working until they appear and only then try to generalize the repeated parts of the code.
In other cases just embrace the benefits we get with the open/closed principle:

- We don't break backward compatibility
- Keeping the code with the small number of interfaces makes it easier to maintain
- Reusing already working code is a reliable and fast solution
- We don't create any new methods that we'll have to maintain in the future.

You can think about other things around you and see that they mostly have a stable user interface. That's a solid reason to keep using them, and it's the same for code! That's why the open/closed principle is so important.