# 

# Code and Blog to showcase the understanding of the SOLID principles

REPRESENTATIVE

UJJAWAL SINGH

[ujjawal.singh@knoldus.com](mailto:ujjawal.singh@knoldus.com)

+917017762293

SOLID principles

SOLID principles are a set of **golden rules used by object-oriented developers** since the early 2000s. They set the standard of how to program in OOP languages and now beyond into agile development and more. SOLID programs scale better, cost less time to work with, and can more easily respond to change. Employers will always prefer a candidate with a strong grasp of SOLID principles.

The 5 principles of SOLID are:

* **S**ingle-responsibility principle
* **O**pen-closed principle
* **L**iskov substitution principle
* **I**nterface segregation principle
* **D**ependency inversion principle

Single-responsibility principle

The **single-responsibility principle** (SRP) states that each class, module, or function in your program should only do one job. In other words, each should have full responsibility for a single functionality of the program. The class should contain only variables and methods relevant to its functionality.

Benefits of Single Responsibility Principle

* When an application has multiple classes, each of them following this principle, then the application becomes more maintainable, easier to understand.
* The code quality of the application is better, thereby having fewer defects.
* Onboarding new members is easy, and they can start contributing much faster.
* Testing and writing test cases is much simpler

Example

Example to understand this concept better. Consider a food delivery application that takes food orders, calculates the bill, and delivers it to customers. We can have 1 separate class for each of the tasks to be performed, and then the main class can just invoke those classes to get these actions done one after the other.

We have a Customer class that has customer attributes like name, address. Order class has all order information like item name, quantity.

The BillCalculation class calculates the total bill and sets the bill amount in the order object. The DeliveryApp has 1 task of delivering the order to the customer. In the real world, these classes would be more complex and might require their functionality to be further broken down into multiple classes.

For example, the bill calculation logic might require some kind of lookup functionality to be implemented where we look for the price of each item included in the order against some kind of database, add them up, add taxes, delivery charges, etc and finally reach the total price. Depending on how complex the code starts to become, we might want to move the taxes, database queries etc, to other separate classes. Similarly, the delivery class might want to interface with another task management system that actually assigns the task of delivery to different delivery agents based on location, shift timings, whether that delivery person has actually shown up to work, etc. These individual steps could move to separate classes when they need specialized handling.

If the functionality of bill calculation, as well as order delivery, was added in the same class, then that class gets modified whenever the bill calculation logic or the delivery agent logic needs to change; which goes against the Single Responsibility Principle. As per the example, we have a separate class for handling each of these functions. Any single business requirement change should ideally have an impact on only one class, thus catering to the Single Responsibility Principle.

class GFG {

public static void main(String[] args)

{

Customer customer1 = new Customer();

customer1.setName("John");

customer1.setAddress("Pune");

Order order1 = new Order();

order1.setItemName("Pizza");

order1.setQuantity(2);

order1.setCustomer(customer1);

order1.prepareOrder();

Bill Calculation billCalculation

= new BillCalculation(order1);

bill Calculation.calculateBill();

DeliveryApp deliveryApp = new DeliveryApp(order1);

deliveryApp.delivery();

}

}

Open-closed principle

The **open–closed principle** states "*software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification*" that is, such an entity can allow its behaviour to be extended without modifying its source code.

The name *open–closed principle* has been used in two ways. Both ways use generalizations (for instance, inheritance or delegate functions) to resolve the apparent dilemma, but the goals, techniques, and results are different.

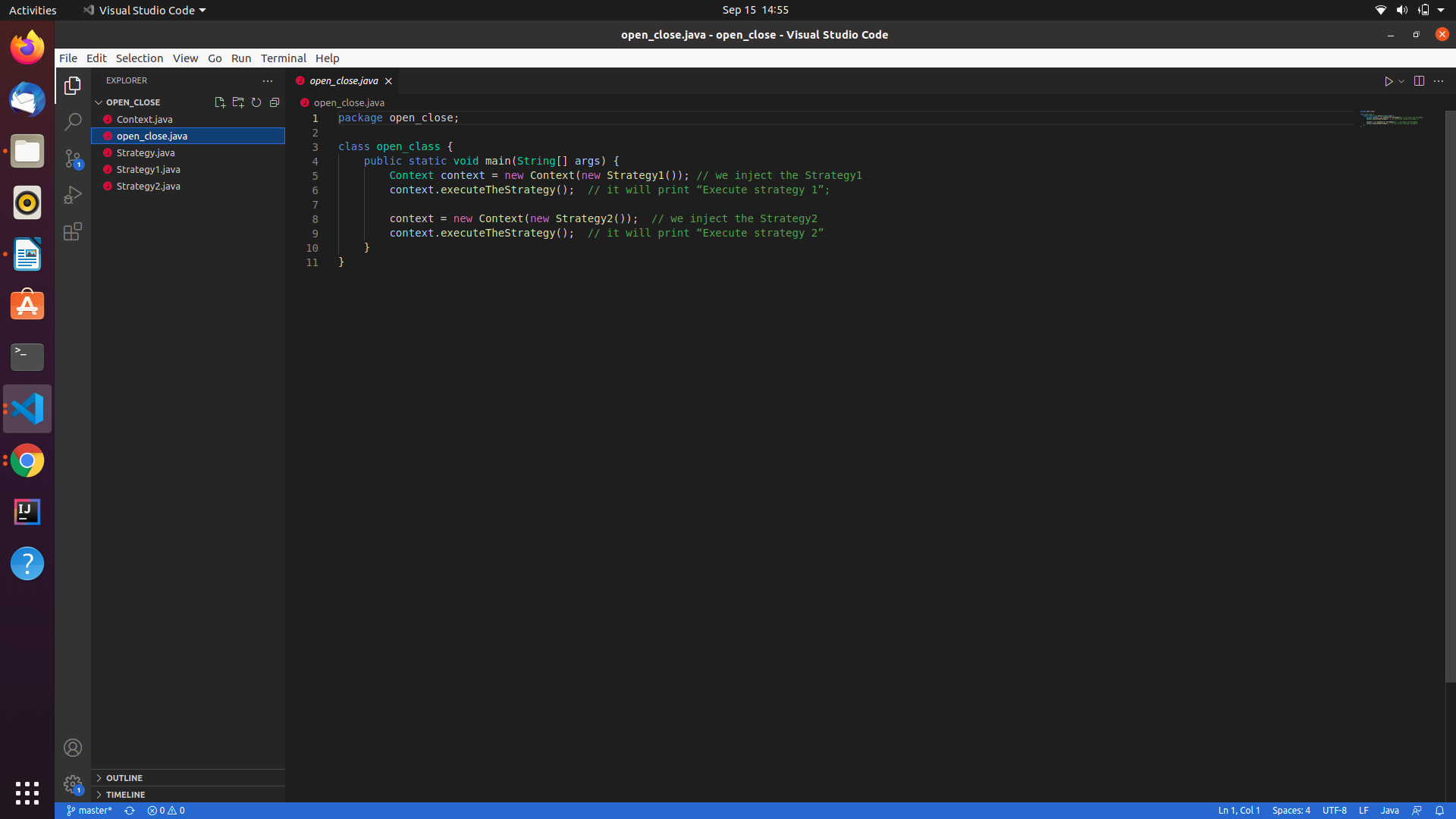
Example

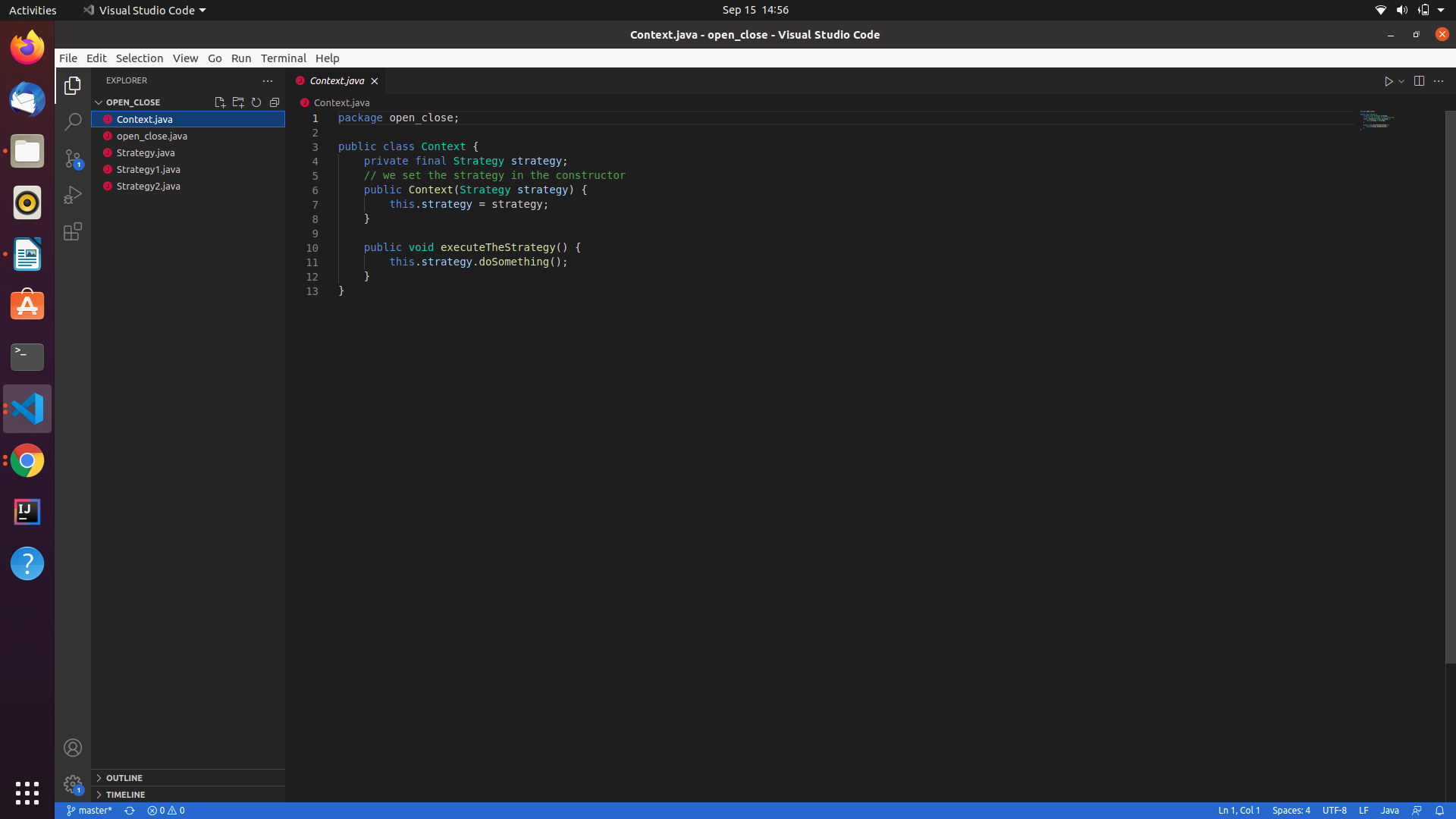
Let’s see a simple example. We will need a strategy interface, a class that uses the strategies, the context class, and some implementation of the strategy interface.

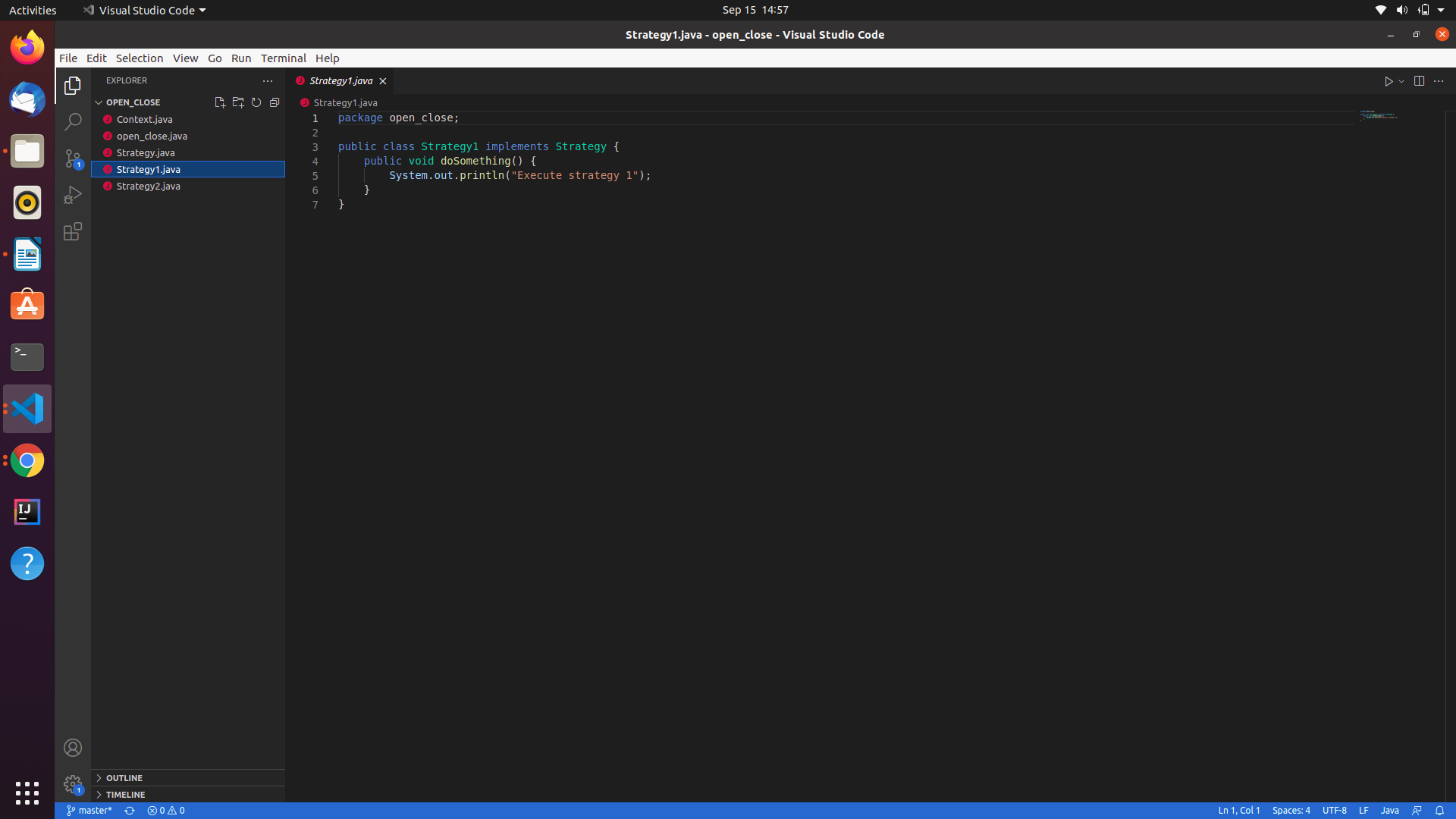
The context class should know just one thing about the strategies — what methods to call. This is what all the strategies have in common, so we will have a strategy interface with just the common methods (in our case, just one method).

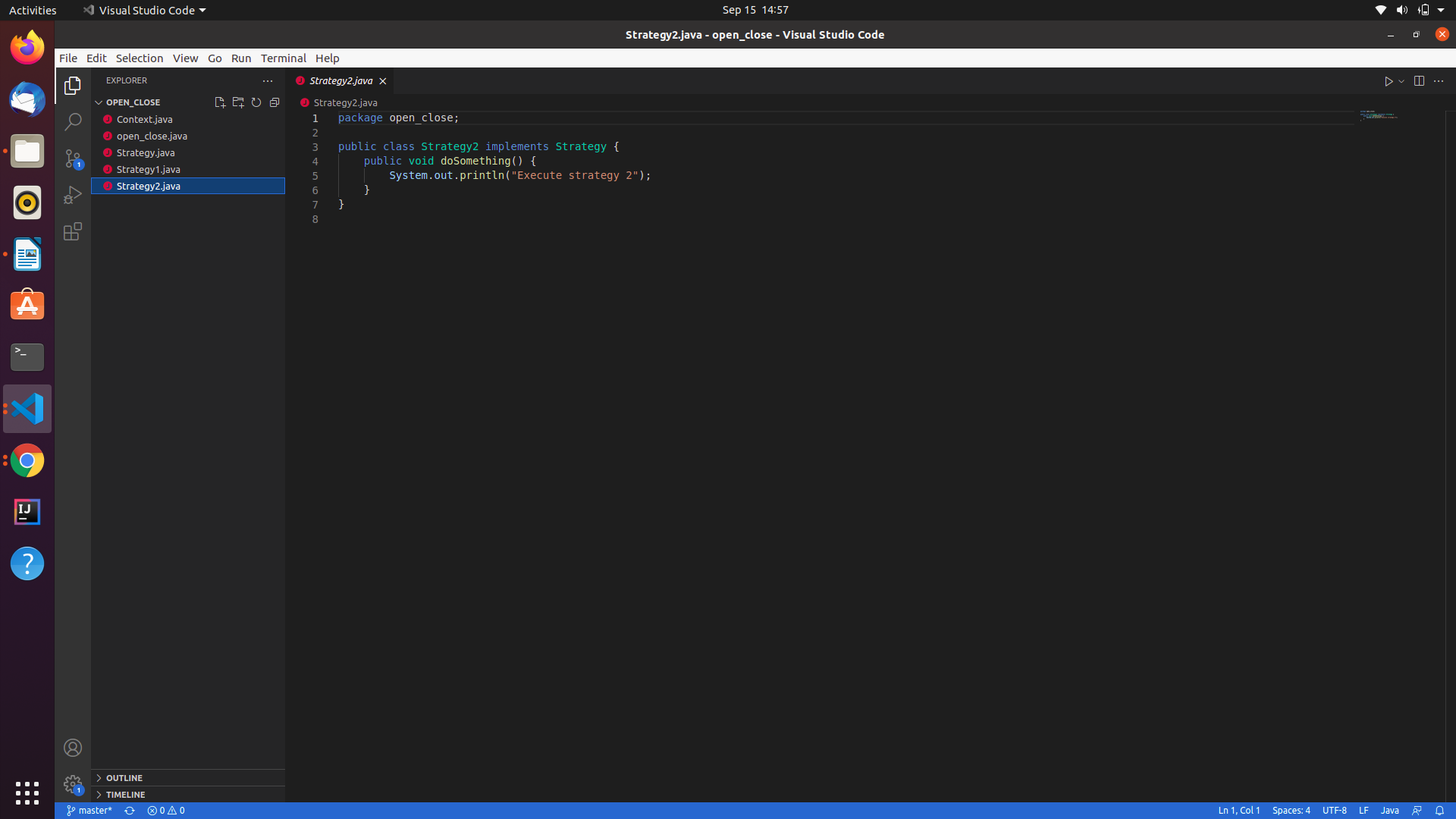
You could implement however many strategies you want and no matter how they work and what you want them to do, you don’t need to modify the context class. The context class knows just that it must call doSomething method and it is enough.

This is a trivial example, but the possibilities are unlimited. Just think of the advantages you get here, and it is not hard at all to follow this simple pattern. Just use interfaces and let a concrete class to know just what it needs to know about something by tying it to an interface instead to a concrete class. This way, you can extend the behavior just by implementing different strategies and without changing the context class's functionality.









Liskov substitution principle

The **Liskov substitution principle** (LSP) is a specific definition of a subtyping relation created by Barbara Liskov and Jeannette Wing. The principle says that any class must be directly replaceable by any of its subclasses without error.

In other words, each subclass must maintain all behavior from the base class along with any new behaviors unique to the subclass. The child class must be able to process all the same requests and complete all the same tasks as its parent class.

Example

