The Open-Closed Principle :

The Open-Closed Principle
Enabling loose coupling

The Open-Closed Principle **SOLID principles: The Open-Closed Principle (OCP)**
… *"a module, class, or function should be open for extension but closed for modification. "*

Bertrand Meyer coined the principle, suggesting that we should build software to be extendable without touching its current code implementation.
But there are situations we change one of our classes, and we realize that we need to adapt all its depending classes.

1.Enables loose coupling.
2.Helps to avoid future software maintenance costs.

When we cannot visualize future requirements

For instance, imagine designing and implementing a rate limit algorithm to control the number of requests allowed for every endpoint in a REST API.
The RateLimit class implements an interceptor — *HandlerInterceptor* — that allows an application to intercept HTTP requests before they reach the service. We can either let the request go through or block it and send back the status code 429.
The team wants to retrieve the number of requests by plan from a text file.

The following *getAPIPlans* method retrieves those parameters.

```
public class RateLimit implements HandlerInterceptor {
  private Map<String, Long> apiPlans;

  @Override
  public boolean preHandle(HttpServletRequest request,
    HttpServletResponse response, Object handler) throws Exception {
    //getClientId
    apiPlans = getAPIPlans();
    //build Buckets
    //evaluate request per clientId
    //accept(200) or refuse(429) request
  }

  private Map<String, Long> getAPIPlans() throws Exception {
    Map<String, Long> apiPlans = new ConcurrentHashMap<>();
    Resource resource = new ClassPathResource("apiPlans.txt");
    try {
      List<String> allLines =
Files.readAllLines(Paths.get(resource.getURI()));
      for (String line: allLines) {
            String[] attributes = line.split(":");
            String plan = attributes[0];
            long capacity = Long.valueOf(attributes[1]).longValue();
            apiPlans.put(plan, capacity);
      }
    } catch (IOException e) {
        throw new RuntimeException(e.getMessage());
    }
    return apiPlans;
  }
}
```

When suddenly an unexpected scenario arises.

The developer leaves the company, and a new one arrives — for example, You.
As developers, we usually receive tasks to do maintenance on projects that do not belong to us; specifically, we never created that code.
Then weeks later, your team decides that parameters must be retrieved from a database. Therefore you proceed to replace the *getAPIPlans* method; then, *you break the open-closed principle*.
That is the meaning of the principle; you can not touch the code that is already implemented and working for a long time. Suppose the code is too

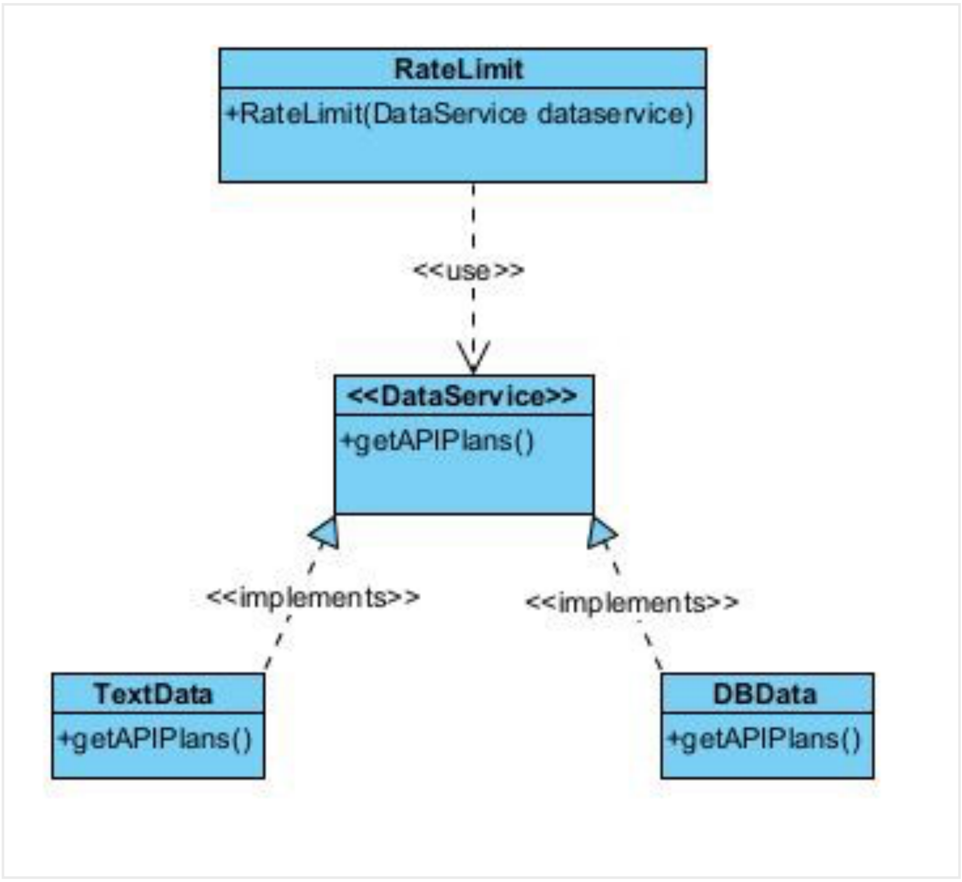
complex to understand, not well documented, and includes a lot of dependencies. In that case, we have a lot of probabilities of introducing a bug or breaking some functionalities that we cannot visualize. Unless it is a bug that we have to fix, we should never modify the existing code.
Even if the code is not well designed or does not follow well object-oriented principles, it could not be easy to extend a class to introduce new functionalities.
The team decides to implement the open-closed principle to support future changes for this scenario. But they need to refactor the code by adopting polymorphism and aggregation.

Polymorphism

Polymorphism is part of the core concepts of Object-Oriented Programming and means many forms, allowing *an object to behave differently in some instances*. For our scenario, polymorphism will enable the *getAPIPlans* method to achieve its goals in different ways: retrieve the parameters from a text file or a database.

Aggregation

Aggregation defines a HAS-A relationship between two classes. Their objects have their life cycle, but one of them is the owner of the HAS-A relationship.
The following diagram shows the goal of our design.



Use of interfaces instead of implementations

}

Enabling the Open-Closed Principle

Firstly, and thinking abstractly, you should create an interface and define a contract that will include all required functionalities.

```
public interface DataService {

  public Map<String, Long> getAPIPlans() throws Exception;

}
```

DataService interface

Secondly, we move our *getAPIPlans* method to a new class that implements the previous interface.

```
public class TextData implements DataService {
  @Override
  public Map<String, Long> getAPIPlans() throws Exception {
    Map<String, Long> apiPlans = new ConcurrentHashMap<>();
         //code omitted

    return apiPlans;
  }
}
```

TextData class

Thanks to abstractions, we can create a new class to implement *getAPIPlans* with different behavior, in this case, to retrieve parameters from a database.

```
public class DBData implements DataService {
  private DataSource datasource;

  public DBData(DataSource datasource) {
    this.datasource = datasource;
  }

  @Override
  public Map<String, Long> getAPIPlans() throws Exception {
    Map<String, Long> apiPlans = new ConcurrentHashMap<>();
    for (Plan plan : datasource.getAPIPlans()) {
      //code omitted

    }
    return apiPlans;
  }
}
```

DBData class

Introducing a new abstraction layer with different implementations avoids tight coupling between classes.
Finally, we refactor our RateLimit class *aggregating* an instance of the DataService type in its constructor method.

The RateLimit class refactored to adapt the Open-closed principle

```
public class RateLimit implements HandlerInterceptor {
  private Map<String, Long> apiPlans;
  private DataService dataService;

  public RateLimit(DataService dataService) {
    this.dataService = dataService;
  }

  @Override
  public boolean preHandle(HttpServletRequest request,
    HttpServletResponse response, Object handler) throws Exception {
    //getClientId
    apiPlans = dataService.getAPIPlans();
    //build Buckets
    //evaluate request per clientId
    //accept(200) or refuse(429) request
  }
}
```

If later we decided to retrieve the parameters from a NoSQL database, we would no longer have to touch the code, create a new class that implements *getAPIPlans*, and instantiate this new class in RateLimit.
Even if, instead of implementing the *HandlerInterceptor* interface, we implement a *Filter* to design our Rate Limit algorithm, we can reuse the DataService interface as one of its dependencies.

Calling to *getAPIPlans* is now fixed (closed for modification). If we want it to behave differently, we implement it in a new class (open for extension) that will follow the contracts defined in our interface.
Our new DBData dependency is instantiated in our RateLimit class thanks to the magic of the Dependency Injection principle, which I will explain in a near-future article, so follow me!.