SOLID principle: Single Responsibility Principle (SRP)

... “a class should only have one reason to change“
This principle states that **a class should only have one responsibility**.
For instance, imagine an online store that issues its cards for its customers, and from the beginning, the Payment and Card teams are in mutual agreement to apply for interest and lock cards from customers who are in late payments for 14 days or more.
In the following code, we have the first design of the Payment Class, which supports both requirements.

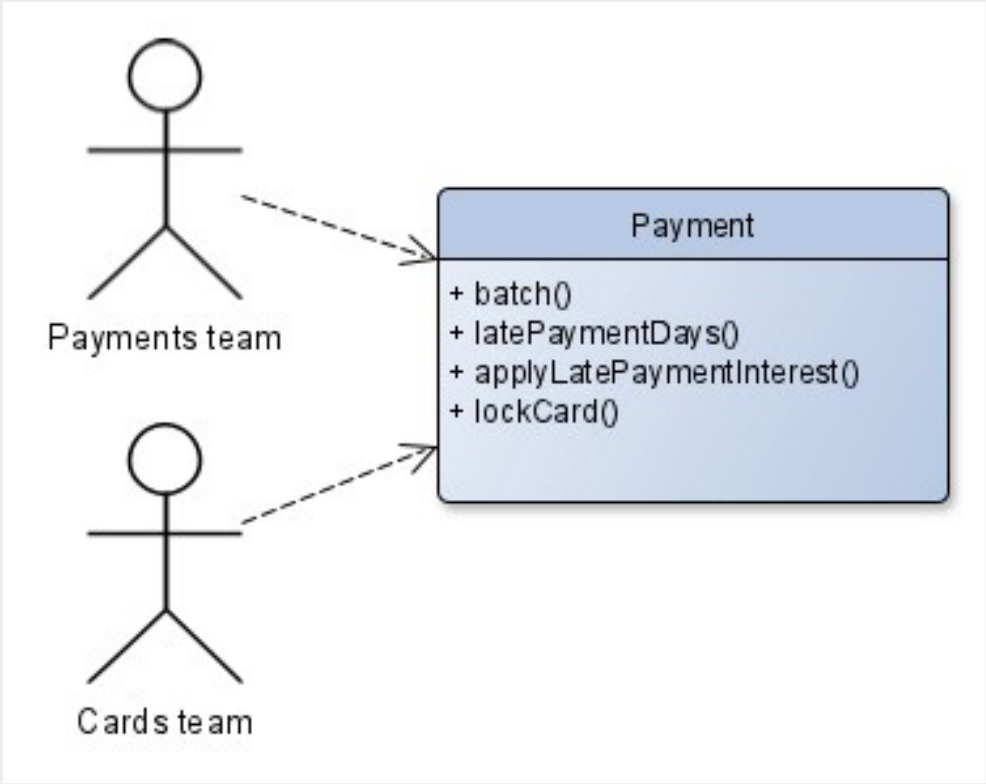
```
public class Payment {
  public static final int MAX_DAYS = 14;

  public void batch(List<Customer> customers) {
    for (Customer customer : customers) {
      int nDays = latePaymentDays(customer);
      if (nDays >= MAX_DAYS) {
        applyLatePaymentInterest(customer);
        lockCard(customer);
      }
    }
  }
}
```

The Problem: The Class has more than one responsibility

But suddenly, the Cards team wants to change the validation to 10 days. However, the Payments team manages other policies related to when interests by late payment are applied. As a result, the Payments team disagrees with the Cards team. Moreover, both teams are stuck on how to proceed.
This scenario is a clear example of how this Class design violates the Single Responsibility Principle. The Payment class has more than one reason to change and breaks the Payments team's business logic if they accept the Cards team's requirement.

The following figure shows the Payment Class with different responsibilities:



The solution: Create a Class with only one responsibility

What do we need to do?. In this scenario, we can apply the **S**ingle Responsibility Principle.
Firstly, we move the *lockCard()* responsibility to a new Card class. This technique is most known as refactoring.

Card class
After that change and following Clean code principles, we can see how it looks the new Payment Class (refactored as well):

```
public class Card {
  public static final int MAX_DAYS = 10;

  public void batch(List customers) {
    for (Customer customer : customers) {
      int nDays = Payment.latePaymentDays(customer);
      if (nDays >= MAX_DAYS) {
        lockCard(customer);
      }
```

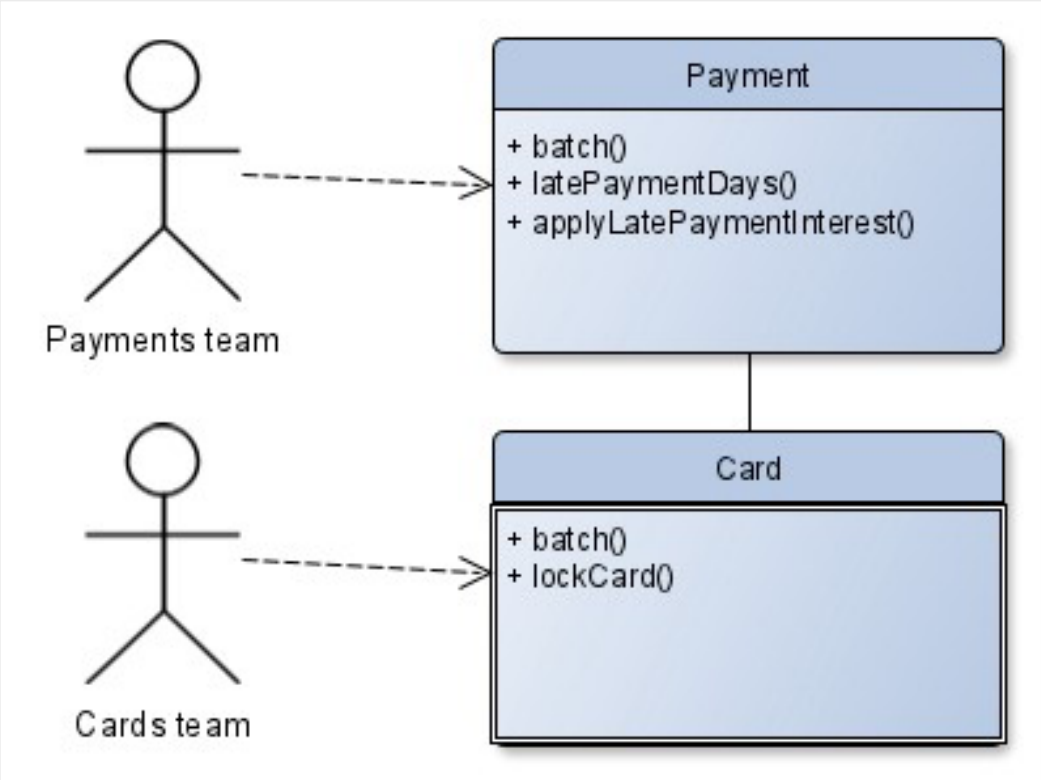
```
    }
  }
}
```

New payment class

```
public class Payment {
  public static final int MAX_DAYS = 14;

  public void batch(List<Customer> customers) {
    for (Customer customer : customers) {
      int nDays = latePaymentDays(customer);
      if (nDays >= MAX_DAYS) {
        applyLatePaymentInterest(customer);
      }
    }
  }
}
```

Finally, new changes to the MAX_DAYS variable will only depend on the requirements of every team separately. The following figure shows the Classes for different actors, without conflicts.



Classes for different actors
Therefore, the Payment Class is only responsible for supporting the Payments team, and the Card Class is solely responsible for supporting the Cards team.
Also, when new features arrive, we need to distinguish in which Class to include them. Moreover, this is related to the Cohesion concept, which helps us group similar functions inside a class that have the same purpose served by that Class.
In conclusion, once you identify classes with too many responsibilities, use this refactoring technique to create smaller classes with single responsibilities and focus only on one business actor.
Use this principle as a tool when translating business software requirements into technical specifications. Programmers must understand these design decisions before programming.

Now that you've learned the Single Responsibility Principle, it's time to learn the Open-closed principle to avoid future software maintenance costs.