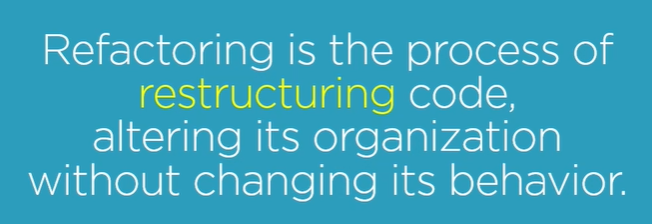
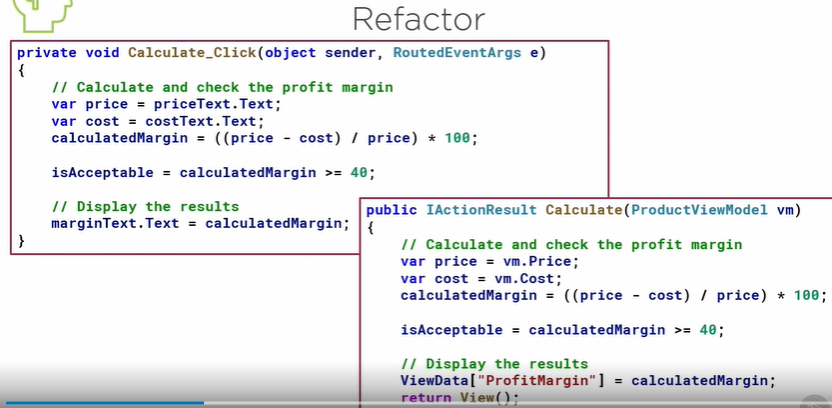
Code comprehension

* **Easy to Read**
* **Have a clear intent**
* **Simple**
* **Thoughtful**

Refactoring code

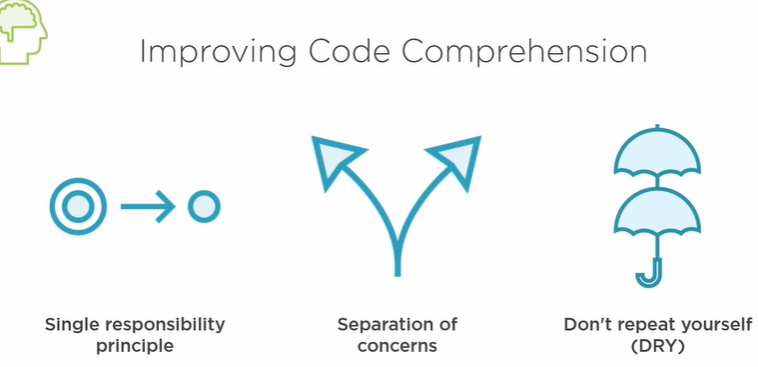
****

## Now refractor this code

****

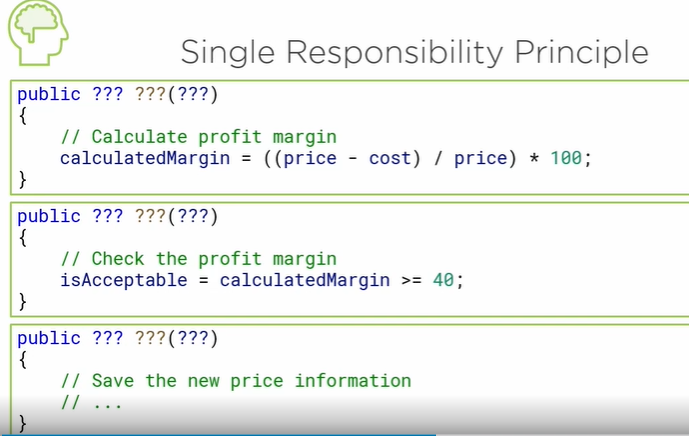
**Improving Code comprehension**

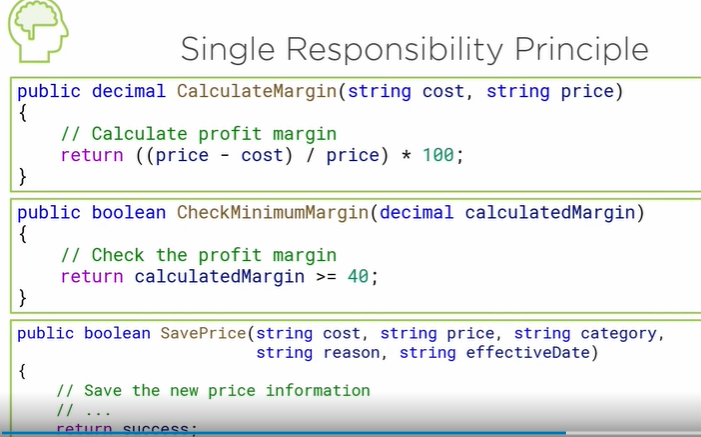
Code compression are practices that we can apply when refractoring our code

****

# Single responsibility Principle

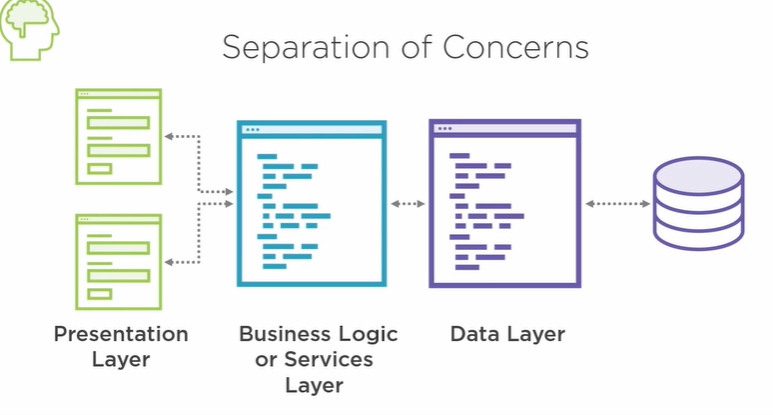
Create three deferent method for each task

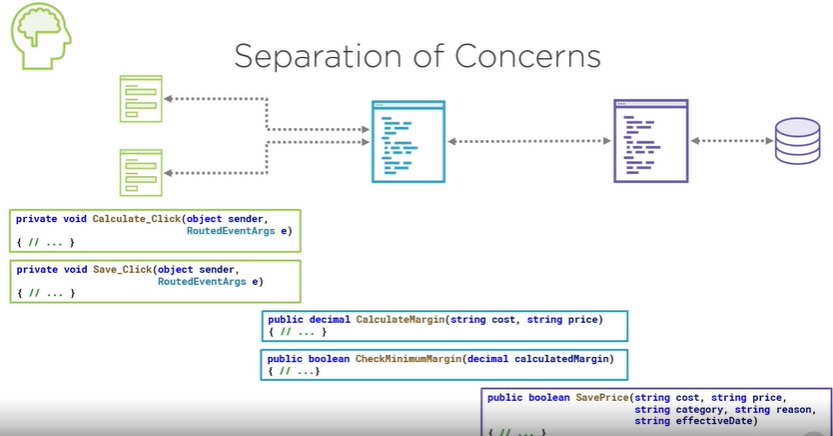
****

****

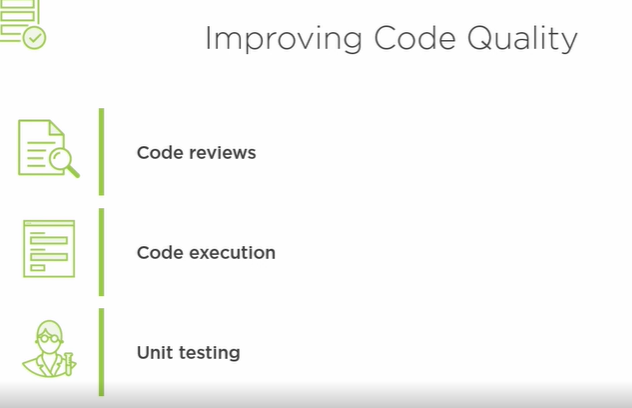
## Separation of concerns

**Separation of concerns involves separating an application into distinct parts where each part addresses a specific concern or aspect of the application.**

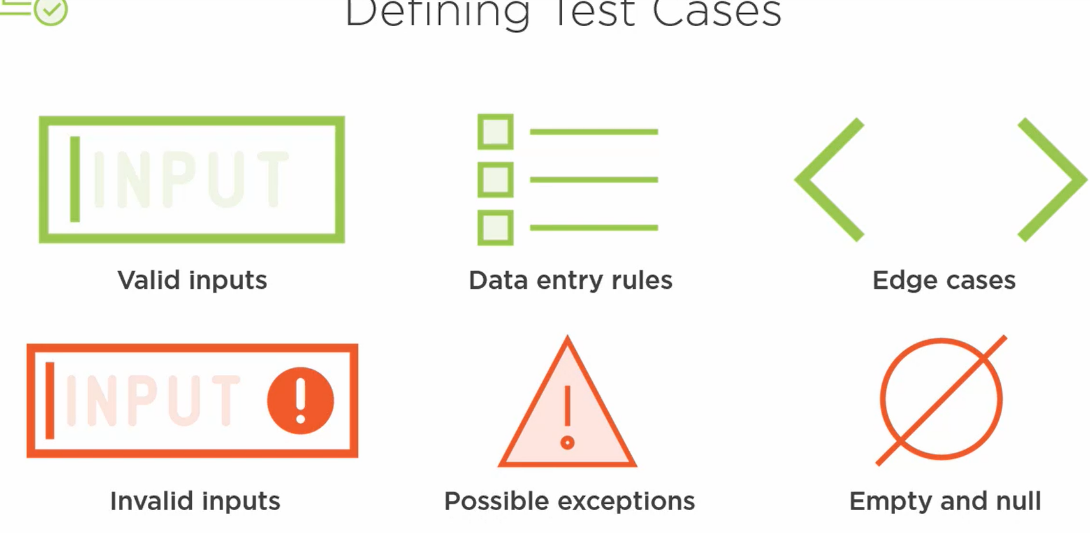
****

****

# Code Quality

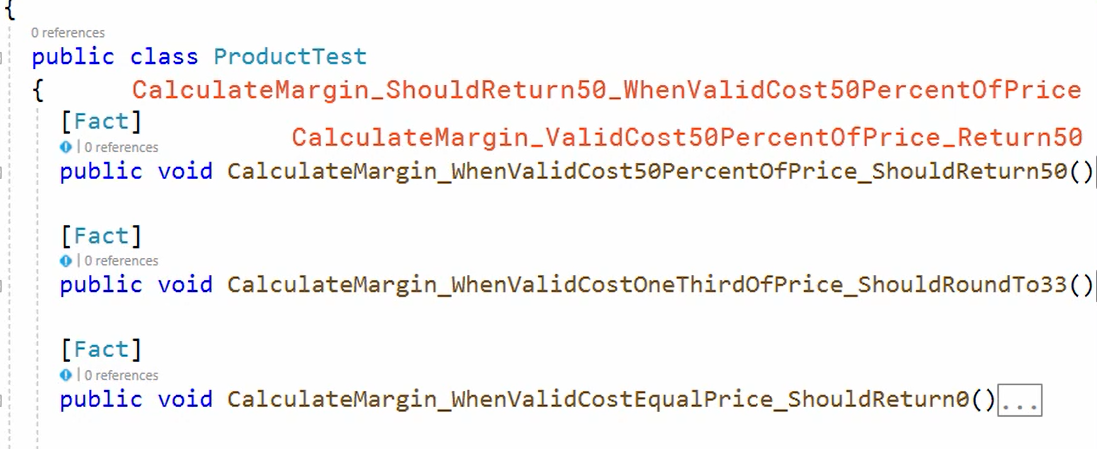
****

**Defining Test Cases**

****

## Demo: Unit testing

Naming method



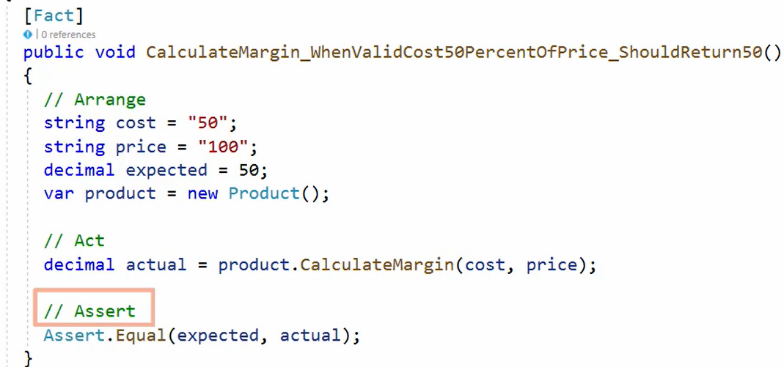
AAA format

Arrange, Act, Assert.

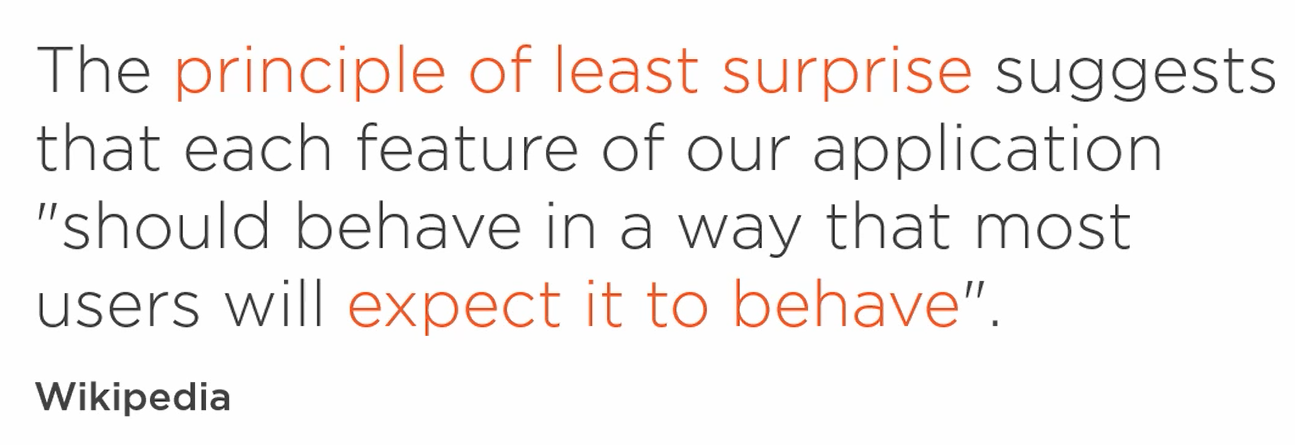
In the Arrange block, we set up the test and define the expected result. In this case, we define the sample data and create an instance of the product class.

In the Act block, we execute the method under test. Here we call the CalculateMargin method, passing in the arranged values.

Lastly, in the Assert block, we assert that the arranged expected result of the operation matches the actual result of the test. If not, the assert causes the test to fail. We follow the same format in all of the tests.



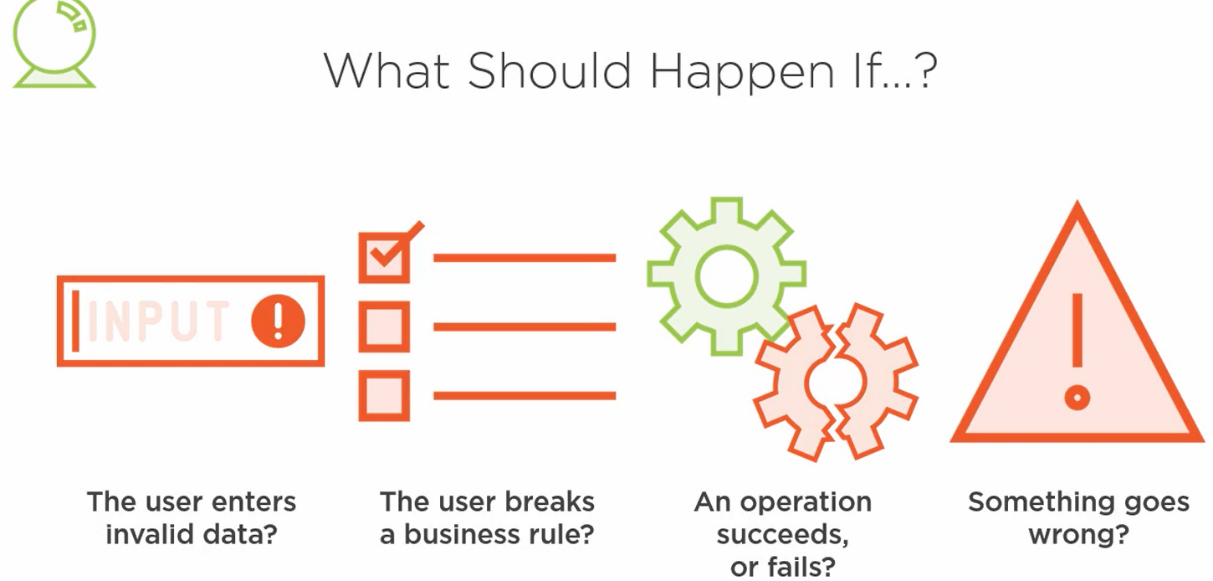
# Improving code predictability \*



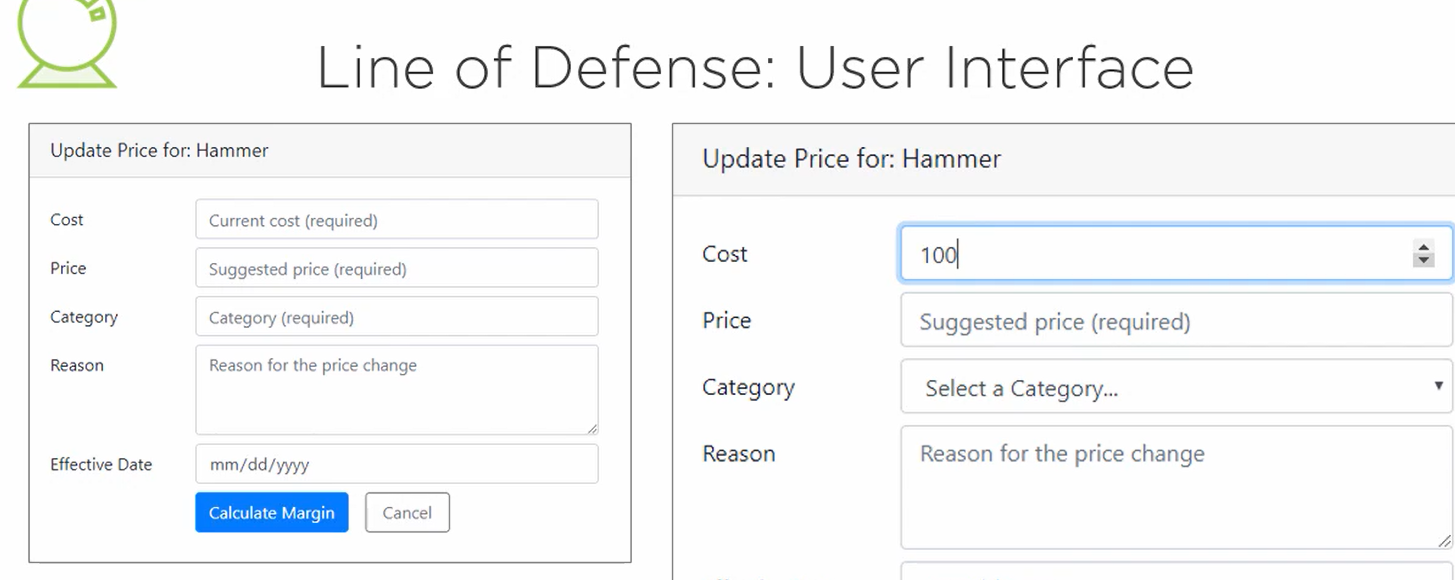
Ever been surprised by a feature that didn't behave as expected? You mistype a digit of your credit card and the app loses your order, or you type just the first character of an email address, and you immediately get a validation error message.

Building predictable code involves asking what should happen if?

What should happen if?



## Line of defense: User Interface



Use a drop down for Cost to minimize that the user will enter a bad format

Grey out button until everything is valid

