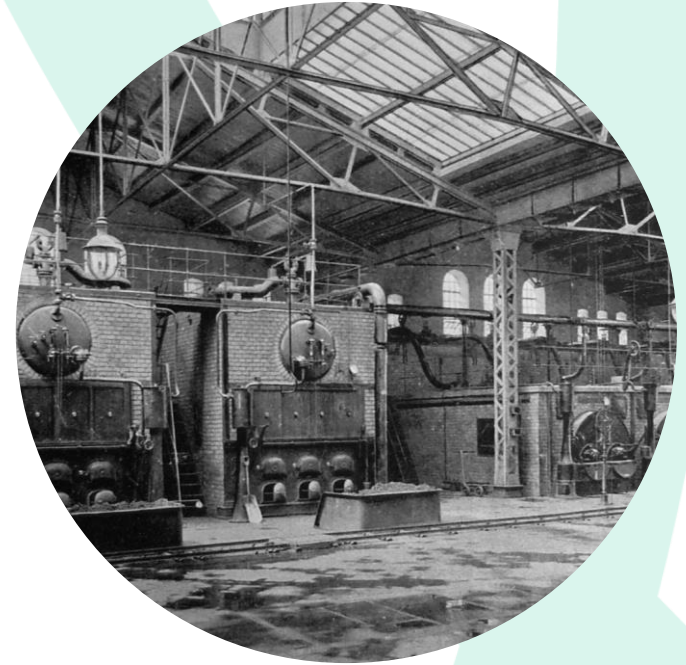


Dealing with Legacy Code



What is Legacy Code?



You're Probably Dealing with Legacy Code if...



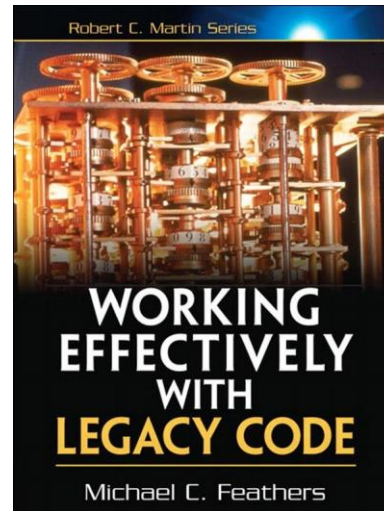
- your code is **hard to test**
- you **don't have Unit Tests** in place
- you need **too much time** to understand it
- you **fear changing** the code
- **a lot of bugs** are reported

Another Definition of Legacy Code...



„To me, legacy code is simply code without tests.“

Michael Feathers, Working Effectively with Legacy Code



Refactor or Rewrite?













Exercise – Refactor or Rewrite?



- Gather in groups
- Discuss the question “Refactor or Rewrite”
- When is it better to refactor an existing solution?
- Are there situations where rewriting from scratch is more appropriate?
- What are the implications (positive and negative) of rewriting or refactoring?
- Collect your answers (5 minutes)
- Discuss them with the audience (5 minutes)

How to decide – Refactoring or Rewrite



Motivations for Refactoring or Rewriting	Refactor	Rewrite
There is risk of loosing the market		
Building knowledge about current system has high priority		
Using state-of-the-art technology has high priority		
Current implementation is problematic and not in production yet		
Current implementation has architectural limitations		

How to decide – Adaption vs Revolution



<https://www.targetprocess.com/blog/refactoring-vs-rewrite/>

How to decide – Be careful



Refactoring and restructuring the existing codebase is almost always the better option.

Only rewrite a system if all of the following are true:

- You've done extensive research and you're nearly certain it will be less expensive and less time-consuming to rewrite.
- You have a ton of time to devote to building the new system.
- You're a better designer than the system's original architect. If you are the system's original architect, are you sure you're significantly better now?
- You've got a plan to get feedback from users early and often as you rewrite.
- You're rich enough and have enough time to maintain both systems while you build the new one



How to decide – Rewrite can be strategically dangerous

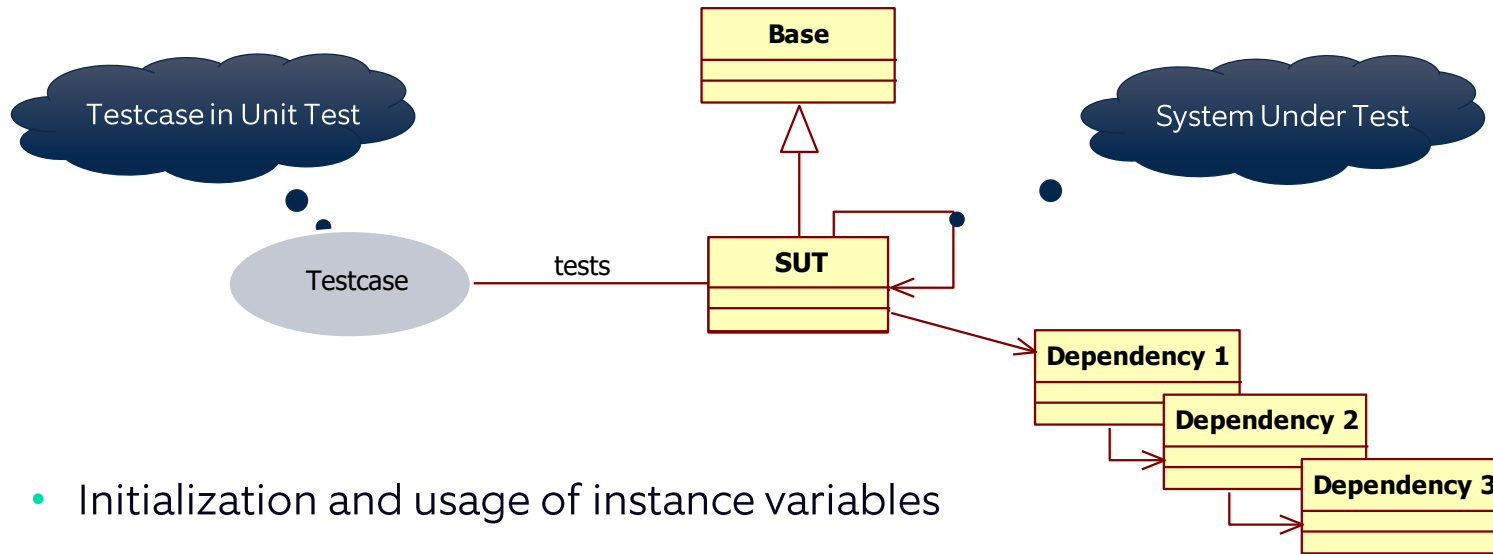


Examples:

- Netscape released version 6 during the year 2000, three years after they released version 4 and yes, there was no version 5. Why? Because the company decided to develop the software from scratch! It took three years for them to do it.
By that time they lost the market.
- Borland bought Arago to make dBase for Windows but started the development from scratch. At the time they were able to release their software, MS Access was already dominating the market.

How to Get Legacy Code Under Test?

The Impact of Dependencies



- Initialization and usage of instance variables
- Calling methods of base classes
- Calling methods of same class
- Instantiation of a local object with new
- Navigation via the object model
- Accessing static methods/members
- Extension methods

Understand and Know Your Dependencies



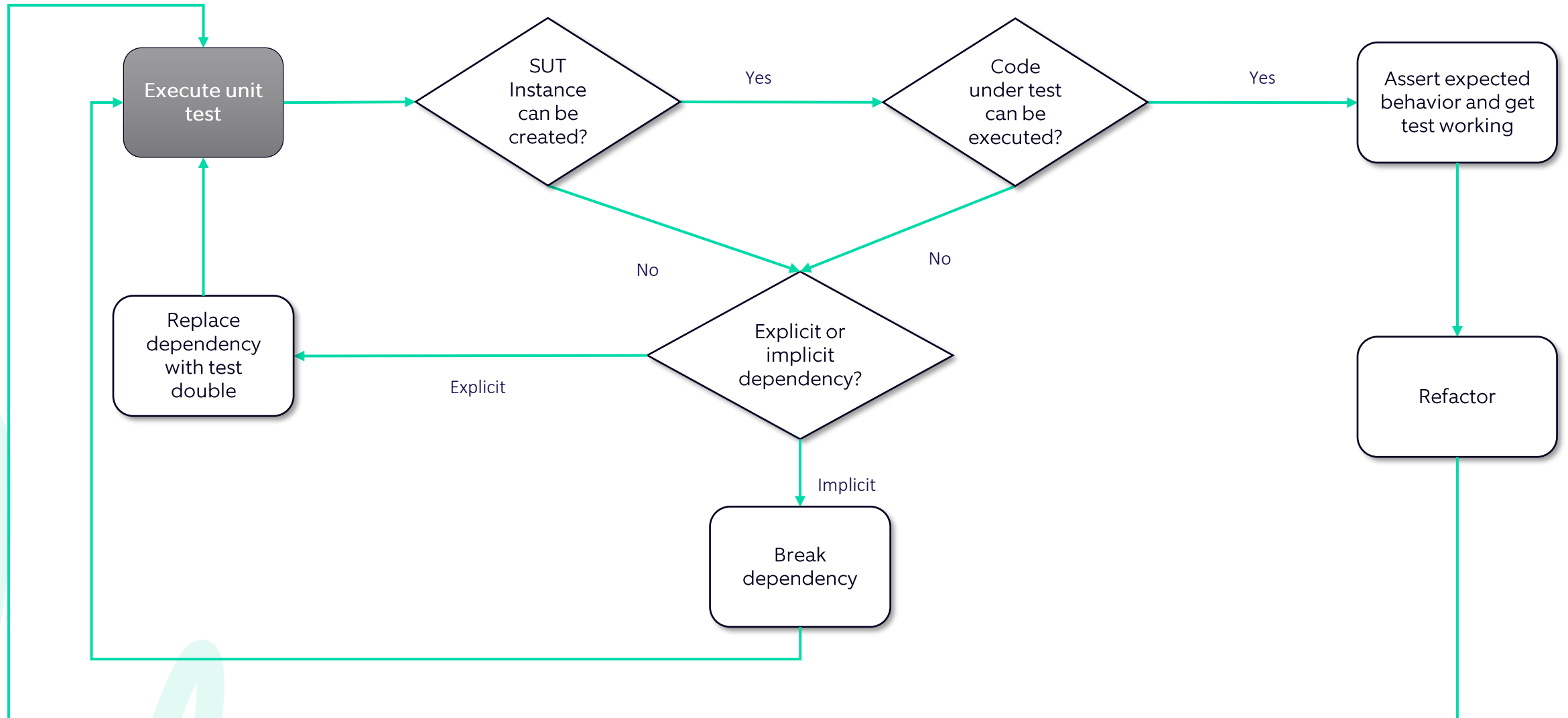
Hidden Dependency

- Not shown in the public interface of the class
- Instances stored in private members
- Instantiation happens in class constructor or methods
- Calls to static methods

Explicit Dependency

- Shown in the public interface of the class
- Inheritance
- Interface implementation
- Constructor and method parameters
- Aspects (Annotations, Attributes)

Getting Legacy Code Under Test



Getting Legacy Code Under Test



- Try to create an instance of the SUT in your test
 - If that works, try to exercise the SUT for your test case
 - If that works, implement the corresponding verifications and proceed with the normal refactoring cycle
 - If that fails, proceed to “use seam or break dependency” and try again
 - If that fails, proceed to “use seam or break dependency” and try again
- **Use seam or break dependency:** check if there is a seam that already allows you to modify the behavior without changing the code of the SUT (e.g. via dependency injection or some other technique such as subclass and override)

How To Break Dependencies?



Dependency Breaking Techniques

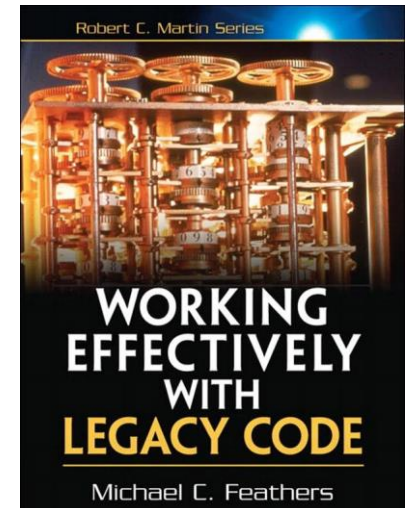


- Help you to break dependencies that prevent you from bringing a class (or method) under test
- Provide guidance on a how to approach certain problem situations where dependencies come in the way of adding tests or executing tests
- Help reducing or eliminating side-effects caused by dependencies
- Help introducing seams to allow for getting the code under test

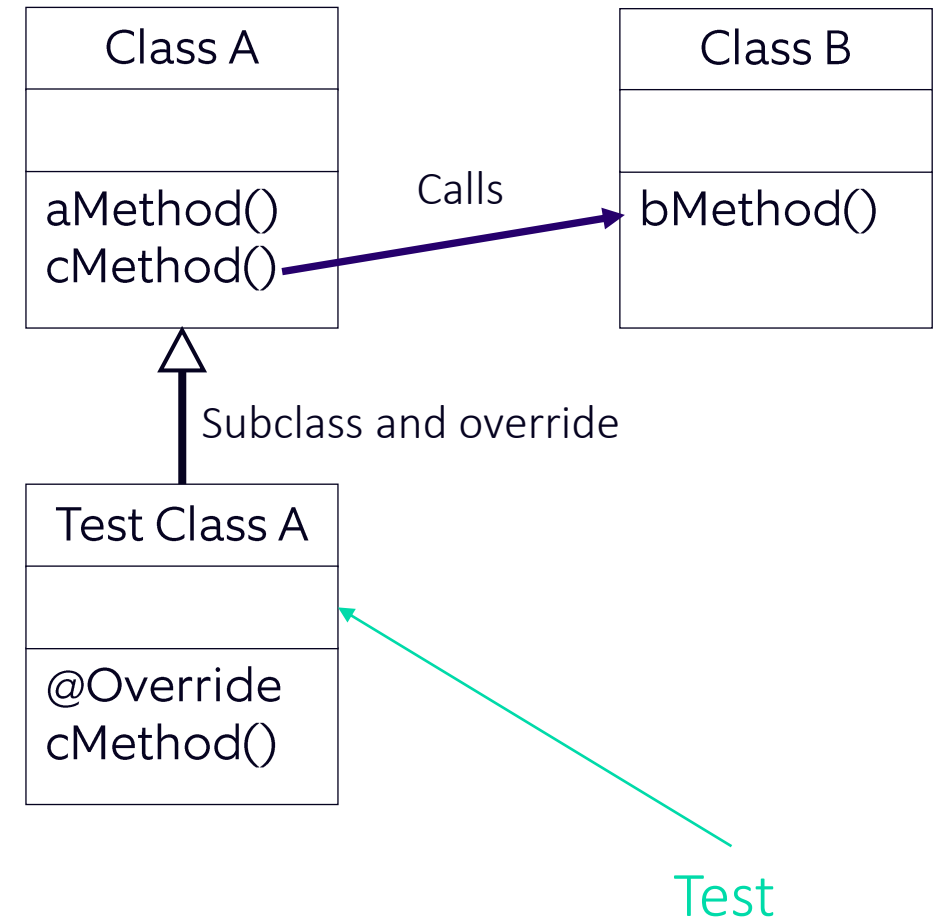
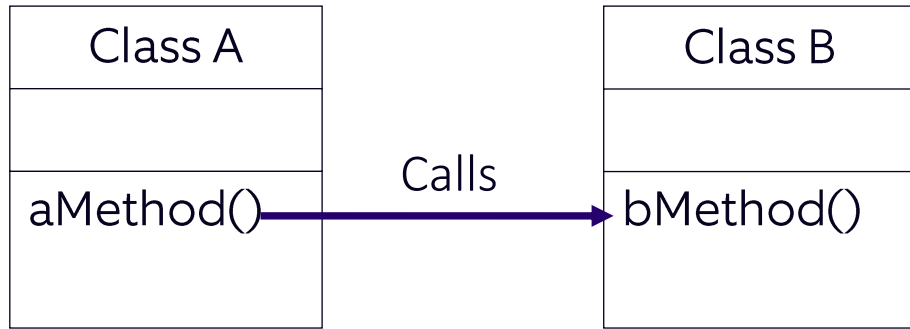
Dependency Breaking Techniques



- Subclass and override
 - Extract and override call
 - Replace global reference with getter
 - Adapt parameter
 - Extract interface
- Parameterize method
 - Introduce instance delegator
 - Pull up feature
 - ... and many more



Extract and override call

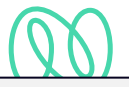


Example Extract and override

```
public class CarRentalContract
{
    private readonly string carPlate;
    private readonly int startKilometer;

    public CarRentalContract(string carPlate)
    {
        this.carPlate = carPlate;
        this.startKilometer = CarRepository.GetKilometer(carPlate);
    }

    public double BillKilometer()
    {
        int endKilometer = CarRepository.GetKilometer(carPlate);
        return (endKilometer - startKilometer) * 0.25d + 2.8d;
    }
}
```



```
public class CarRentalContract
{
    private readonly string carPlate;
    private readonly int startKilometer;

    public CarRentalContract(string carPlate)
    {
        this.carPlate = carPlate;
        this.startKilometer = GetKilometer(carPlate);
    }

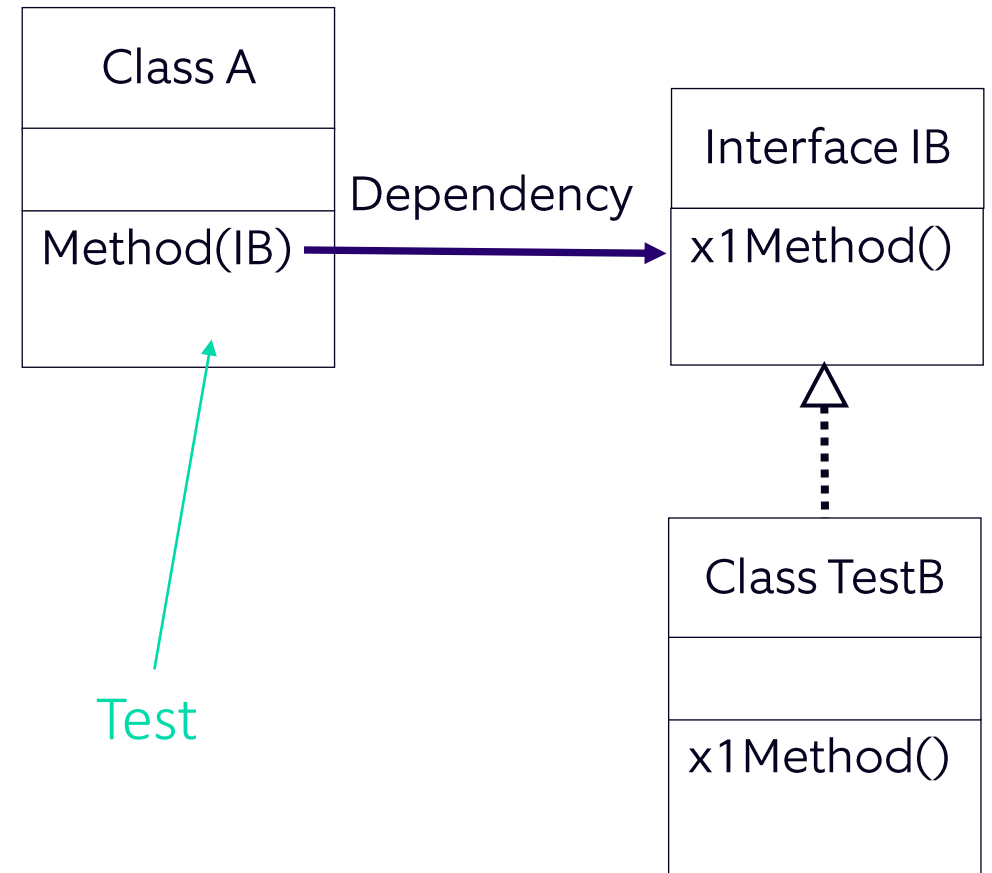
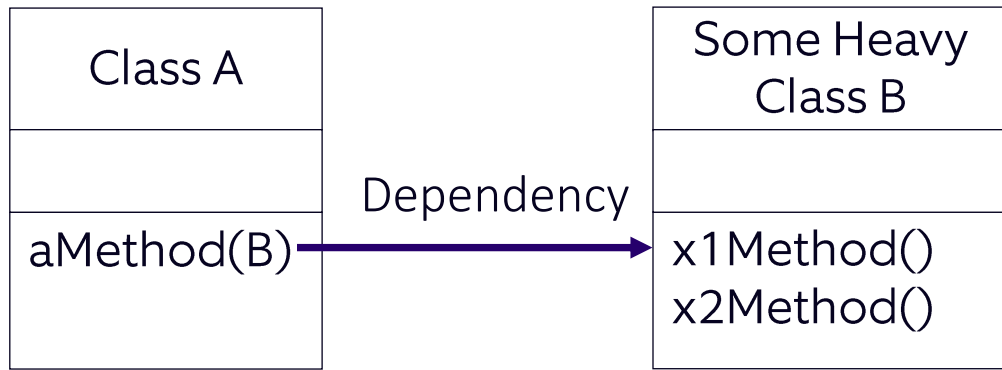
    public double BillKilometer()
    {
        int endKilometer = GetKilometer(carPlate);
        return (endKilometer - startKilometer) * 0.25d + 2.8d;
    }

    protected int GetKilometer(string carPlate) {
        return CarRepository.GetKilometer(carPlate);
    }
}
```

```
public class CarRentalContractForTest extends CarRentalContract
{
    public CarRentalContractForTest(string carPlate)
    {
        super(carPlate);
    }

    protected virtual int GetKilometer(string carPlate)
    {
        return 0;
    }
}
```


Adapt parameter



Test

Wrap a parameter behind an interface

Example Adapt parameter



```
public class CarRentalContract
{
    private readonly int startKilometer;
    private readonly Car car;

    public CarRentalContract(Car car) {
        this.car = car;
        this.startKilometer = car.GetKilometer();
    }
    ...
}
```

```
public class Car
{
    private int kilometer;
    private int make;
    private int model;
    private int engineRPM;
    private readonly string carPlate;
    private boolean engineStarted;
    private FuelType fuelType;

    public enum FuelType { DIESEL, PETROL }

    public int GetKilometer() {
        return kilometer;
    }

    public void SetKilometer(int kilometer) {
        this.kilometer = kilometer;
    }
    ...
}
```

We only need
getKilometer()
from Car

```
public class CarRentalContract
{
    private readonly int startKilometer;
    private readonly CarKilometerReading car;

    public CarRentalContract(CarKilometerReading car) {
        this.car = car;
        this.startKilometer = car.GetKilometer();
    }
    ...
}
```

```
public interface ICarKilometerReading
{
    int GetKilometer(string carPlate);
}
```

```
public class Car implements ICarKilometerReading
{
    ...
}
```

10a

Hands-on: Refactoring Legacy Code



Hands-On – Parameterize Constructor



We have some legacy code. We need to make changes. To make changes we need to introduce tests first. We might have to change some code to enable testing. We need to introduce so-called Seams (see Michael Feathers' Working Effectively with Legacy Code). Changing code without test is risky, so we want to:

- Only change as little code as possible.
- Rely on automated Refactoring tools as much as possible.
- You must not change the public API of the class.

Follow the assignment instructions in repository “10-Legacy_Code”, directory “.../parameterise_constructor”

Hands-On – Subclass and Override



We have some legacy code. We need to make changes. To make changes we need to introduce tests first. We might have to change some code to enable testing. We need to introduce so-called Seams (see Michael Feathers' Working Effectively with Legacy Code). Changing code without test is risky, so we want to:

- Only change as little code as possible.
- Rely on automated Refactoring tools as much as possible.
- You must not change the public API of the class.

Follow the assignment instructions in repository “10-Legacy_Code”, directory “.../subclass_and_override”

Hands-On – Extract and Override Call



We have some legacy code. We need to make changes. To make changes we need to introduce tests first. We might have to change some code to enable testing. We need to introduce so-called Seams (see Michael Feathers' Working Effectively with Legacy Code). Changing code without test is risky, so we want to:

- Only change as little code as possible.
- Rely on automated Refactoring tools as much as possible.
- You must not change the public API of the class.

Follow the assignment instructions in repository “10-Legacy_Code”, directory “.../extract_and_override_call”

Hands-On – Replace Global Reference

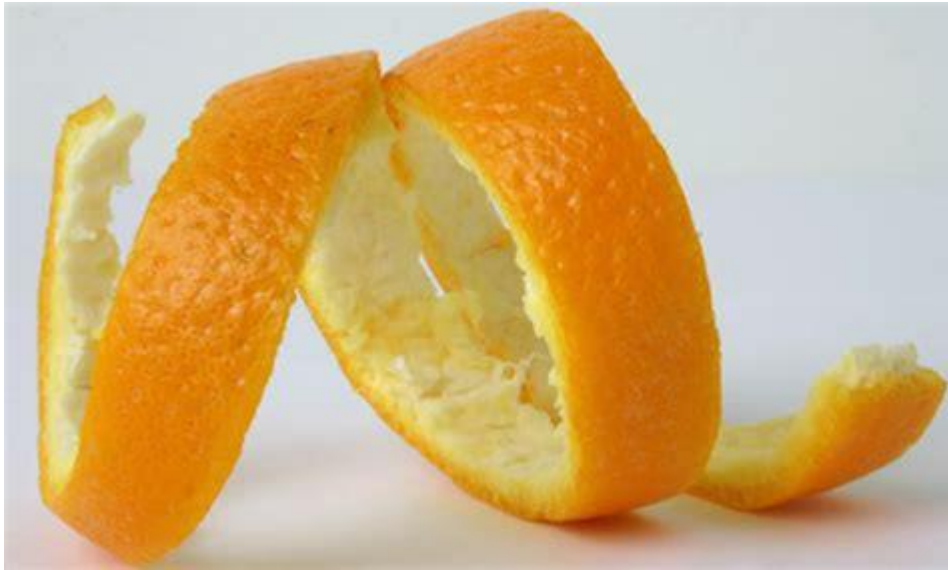


We have some legacy code. We need to make changes. To make changes we need to introduce tests first. We might have to change some code to enable testing. We need to introduce so-called Seams (see Michael Feathers' Working Effectively with Legacy Code). Changing code without test is risky, so we want to:

- Only change as little code as possible.
- Rely on automated Refactoring tools as much as possible.
- You must not change the public API of the class.

Follow the assignment instructions in repository “10-Legacy_Code”, directory “.../replace_global_reference”

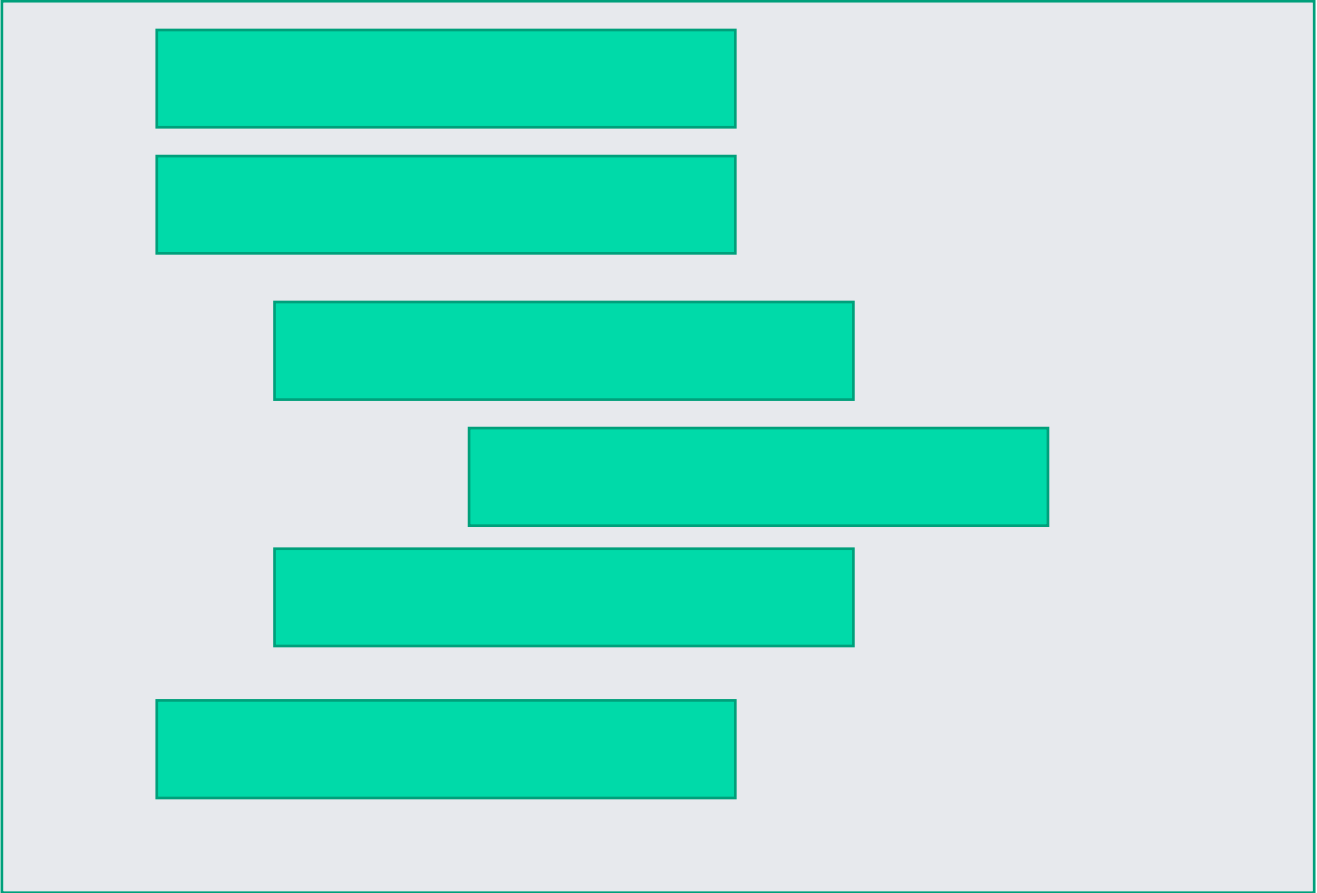
Peel and Slice



Peel



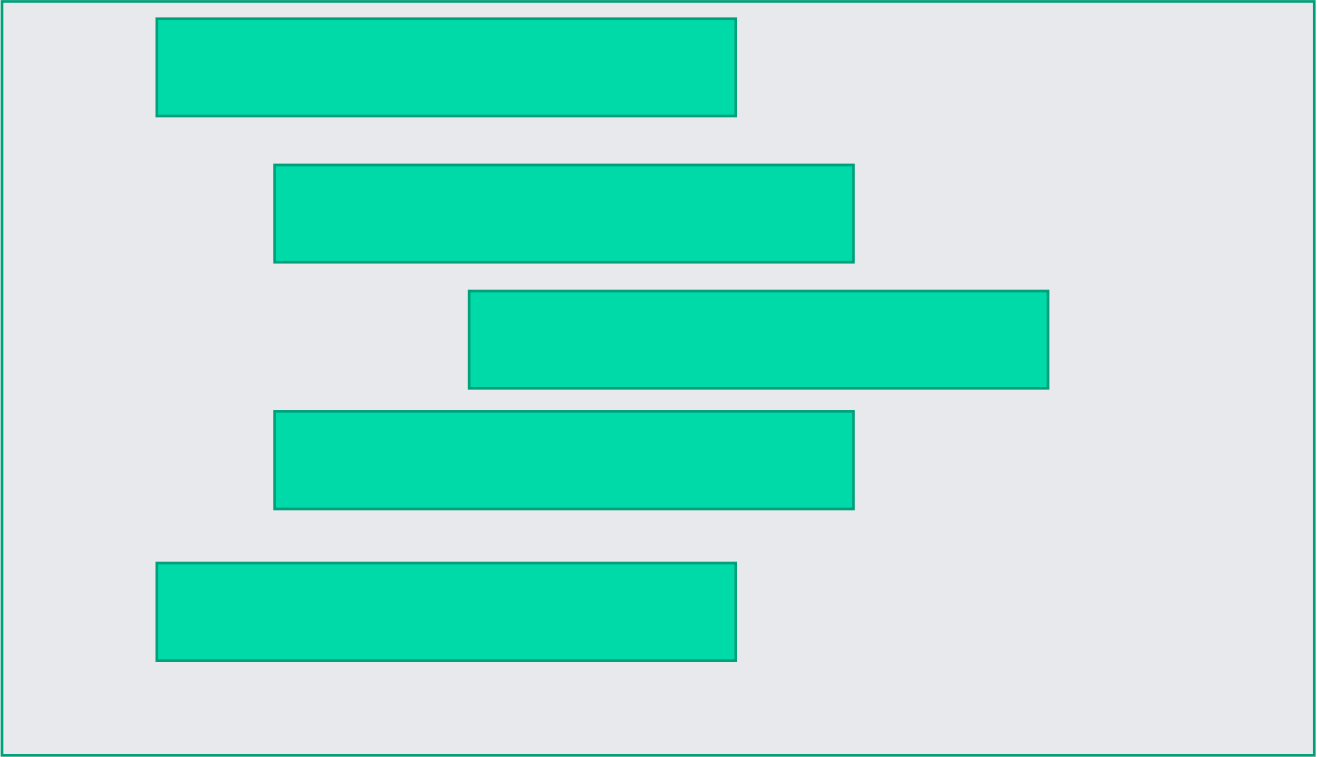
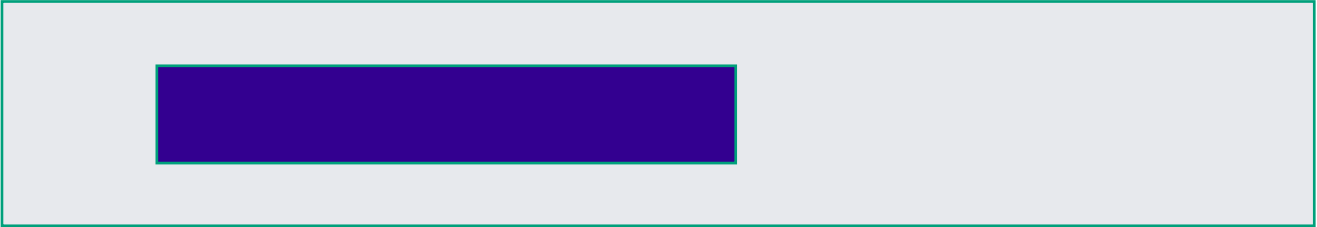
Peel



Peel



Peel



Slice



Slice



Slice

