40. What is the Template Method design pattern?

Brief summary: Template Method is a design pattern that defines the skeleton of an algorithm in the base class. Specific steps of this algorithm are implemented in derived classes.

Template Method is a design pattern that defines the skeleton of an algorithm in the base class. Specific steps of this algorithm are implemented in derived classes.

Let's consider the following example: we are developing a platform that allows users to play board games online. The first board game we deliver is Settlers of Catan. Here is the (slightly simplified) implementation:

```
class SettlersOfCatan
{
    private Random _random = new Random();

    1 reference
    public void Play()
    {
        SetupBoard();
        bool isFinished = false;
        while (!isFinished)
        {
            isFinished = PlayTurn();
        }
        SelectWinner();
    }

    1 reference
    private void SetupBoard()
    {
        Console.WriteLine("Randomly placing hexagonal tiles.");
    }

    1 reference
    private bool PlayTurn()
    {
        Console.WriteLine("Building, trading, etc.");
        return _random.Next(5) >= 4;
    }

    1 reference
    private void SelectWinner()
    {
        Console.WriteLine(
            "Winner is the one who first got 12 points");
    }
}
```

All right. Soon after we are asked to implement another game - this time it's Terraforming Mars:

```
class TerraformingMars
{
    private Random _random = new Random();

    0 references
    public void Play()
    {
        SetupBoard();
        bool isFinished = false;
        while (!isFinished)
        {
            isFinished = PlayTurn();
        }
        SelectWinner();
    }

    1 reference
    private void SetupBoard()
    {
        Console.WriteLine("Choosing from two available maps.");
    }

    1 reference
    private bool PlayTurn()
    {
        Console.WriteLine(
            "Raising oxygen level, placing oceans, etc.");
        return _random.Next(5) >= 4;
    }

    1 reference
    private void SelectWinner()
    {
        Console.WriteLine(
            "Winner is the one with most points at game's end.");
    }
}
```

Huh. This is quite similar to the code we had before. After implementing couple more board games, we come to a revelation: all bords games follow a similar **template**! We first set up the board, then we play turns until the game is finished, and finally, we select the winner.

Instead of repeating this logic in each class, we could define it once in the base class, and ask the subclasses to only provide the details of the implementation of each step. This way, if the template changes for some reason, we will only have one place to fix.

Let's use the Template Method design pattern in this code. First, let's define the template itself. It will be done by using an abstract class:

```
abstract class BoardGame
{
    protected Random Random = new Random();

    0 references
    public void Play()
    {
        SetupBoard();
        bool isFinished = false;
        while (!isFinished)
        {
            isFinished = PlayTurn();
        }
        SelectWinner();
    }

    1 reference
    protected abstract void SetupBoard();
    1 reference
    protected abstract bool PlayTurn();
    1 reference
    protected abstract void SelectWinner();
}
```

Now we can implement the concrete games:

```
class SettlersOfCatan : BoardGame
{
    2 references
    protected override void SetupBoard()
    {
        Console.WriteLine("Randomly placing hexagonal tiles.");
    }

    2 references
    protected override bool PlayTurn()
    {
        Console.WriteLine("Building, trading, etc.");
        return Random.Next(5) >= 4;
    }

    2 references
    protected override void SelectWinner()
    {
        Console.WriteLine(
            "Winner is the one who first got 12 points");
    }
}
```

```
class TerraformingMars : BoardGame
{
    private Random _random = new Random();

    2 references
    protected override void SetupBoard()
    {
        Console.WriteLine("Choosing from two available maps.");
    }

    2 references
    protected override bool PlayTurn()
    {
        Console.WriteLine(
            "Raising oxygen level, placing oceans, etc.");
        return _random.Next(5) >= 4;
    }

    2 references
    protected override void SelectWinner()
    {
        Console.WriteLine(
            "Winner is the one with most points at game's end.");
    }
}
```

Great. Now the thing that those classes had in common - so the general template of each game - is enclosed in the base type. If this template changes, we will only need to adjust the base class. The derived classes only define what makes each board game special, and they don't replicate what they have in common.

The Template Method design pattern is useful everywhere where some base algorithm is needed, but the specific parts of it vary. A practical example could be the execution flow of tests in the unit tests framework. Typically such execution looks like this:

Foreach test:

1) **Run the SetUpMethod**
2) **Execute test**
3) **Run the TearDown method**

This could easily be achieved with the Template Method design pattern. First, let's define the base class for all test fixtures:

```
abstract class TestFixture
{
    1 reference
    public bool Run()
    {
        int failedTestsCount = 0;
        foreach (var test in GetTests())
        {
            SetUp();
            if (!test())
            {
                failedTestsCount++;
            }
            TearDown();
        }
        return failedTestsCount == 0;
    }

    2 references
    protected abstract IEnumerable<Func<bool>> GetTests();
    2 references
    protected abstract void SetUp();
    2 references
    protected abstract void TearDown();
}
```

And now, let's define some actual tests. We will be testing this super-complicated class:

```
class Calculator
{
    0 references
    public int Add(int a, int b)
    {
        return a + b;
    }
}
```

```
class CalculatorTests : TestFixture
{
    private Calculator _cut;

    2 references
    protected override void SetUp()
    {
        _cut = new Calculator();
        Console.WriteLine("SetUp of MyTests class");
    }

    2 references
    protected override IEnumerable<Func<bool>> GetTests()
    {
        return new List<Func<bool>>()
        {
            () =>
            {
                Console.WriteLine("3 + 2 shall be 5");
                return _cut.Add(3,2) == 5;
            },
            () =>
            {
                Console.WriteLine("10 + (-10) shall be 0");
                return _cut.Add(10, -10) == 0;
            }
        };
    }

    2 references
    protected override void TearDown()
    {
        Console.WriteLine("TearDown of MyTests class");
    }
}
```

When we run those tests, we will see that the SetUp and TearDown methods are executed before and after each test, as expected:

```
var calculatorTests = new CalculatorTests();
Console.WriteLine(calculatorTests.Run() ? "Success!" : "Failure");
```

```
SetUp of MyTests class
3 + 2 shall be 5
TearDown of MyTests class
SetUp of MyTests class
10 + (-10) shall be 0
TearDown of MyTests class
Success!
```

All right! As you can see, the Template Method design pattern can be quite handy everywhere where a generic algorithm shall be defined once, but the implementations of the particular steps of this algorithm may vary.

Bonus questions:

- "**What is the difference between the Template Method design pattern and the Strategy design pattern?**"
  Both patterns allow specifying what concrete algorithm or a piece of the algorithm will be used. The main difference is that with the Template Method, it is selected at compile-time, as this pattern uses the inheritance. With the Strategy pattern, the decision is made at runtime, as this pattern uses composition.