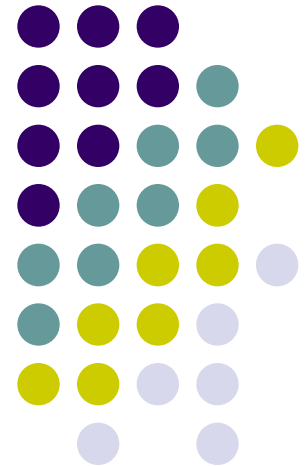


# Разработка через тестирование



# Определение



**Разработка через тестирование** (англ. test-driven development) — техника программирования, при которой модульные тесты для программы или ее фрагмента пишутся до самой программы и, по существу, управляют ее разработкой.

- ❑ **Методика:** Пишем новый код только тогда, когда автоматический тест не сработал.
- ❑ Удаляем дублирование

# Цикл разработки (кратко)



**Красный** — напишите небольшой тест, который не работает, а возможно, даже не компилируется.

**Зеленый** — заставьте тест работать как можно быстрее, при этом не думайте о правильности дизайна и чистоте кода. Напишите ровно столько кода, чтобы тест сработал.

**Рефакторинг** — удалите из написанного кода любое дублирование.

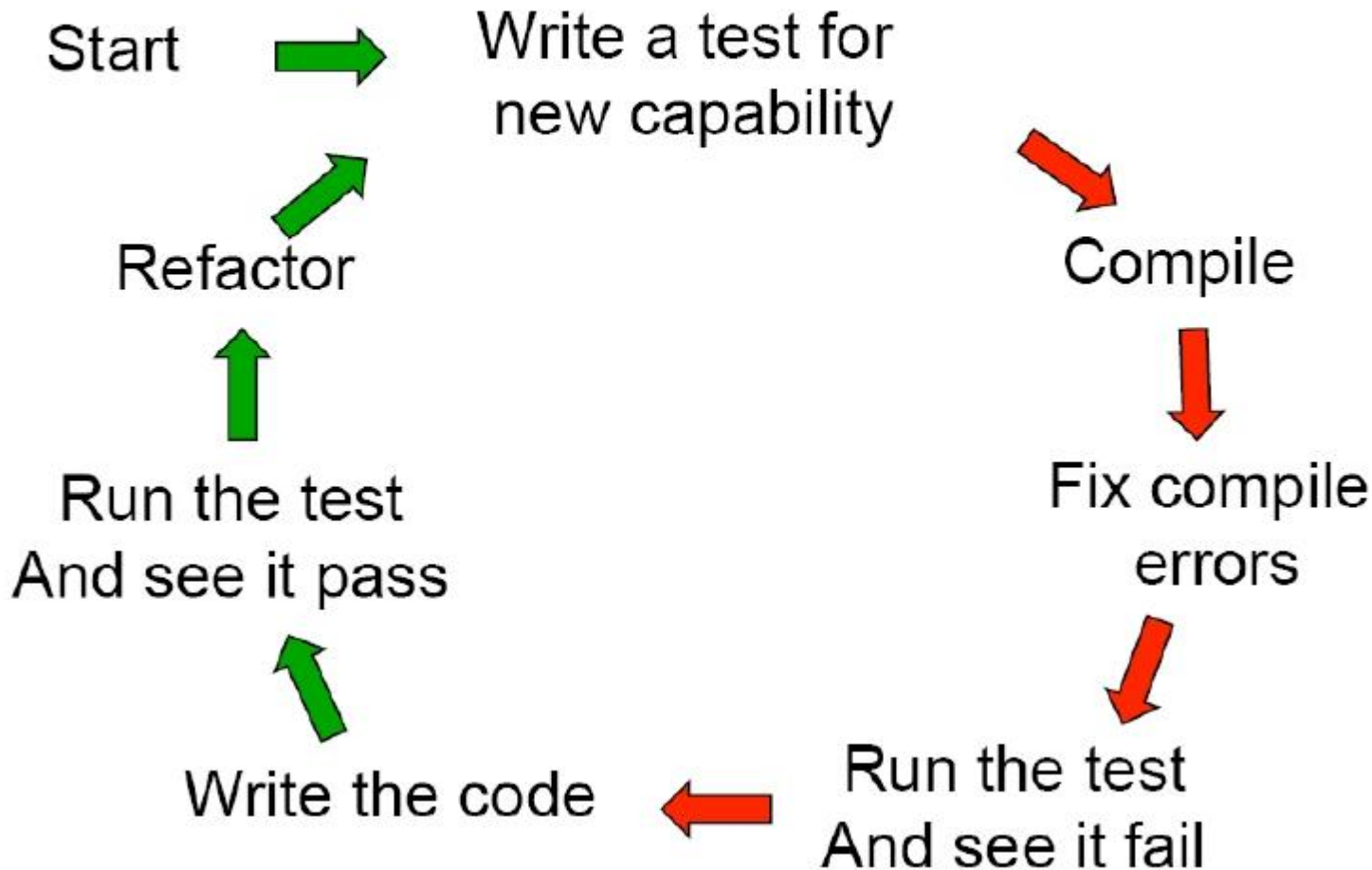
**Красный** — **зеленый** — **рефакторинг**

**Рефакторинг** — процесс полного или частичного преобразования внутренней структуры программы при сохранении ее внешнего поведения.

# Цикл разработки (схема)



## Test Driven Development Cycle



# Цикл разработки



1. Из репозитория извлекается программная система, находящаяся в согласованном состоянии, когда весь набор модульных тестов выполняется успешно.
2. Добавляется новый тест. Он может состоять в проверке, реализует ли система некоторое новое поведение или содержит ли некоторую ошибку, о которой недавно стало известно.
3. Успешно выполняется весь набор тестов, кроме нового теста, который выполняется неуспешно. Этот шаг необходим для проверки самого теста - включен ли он в общую систему тестирования и правильно ли отражает новое требование к системе, которому она, естественно, еще не удовлетворяет.
4. Программа изменяется с тем, чтобы как можно скорее выполнялись все тесты. Нужно добавить самое простое решение, удовлетворяющее новому тесту, и одновременно с этим не испортить существующие тесты. Большая часть нежелательных побочных и отдаленных эффектов от вносимых в программу изменений отслеживается именно на этом этапе, с помощью

# Цикл разработки (продолжение)



5. Весь набор тестов выполняется успешно.
6. Теперь, когда требуемая в этом цикле функциональность достигнута самым простым способом, программа подвергается *рефакторингу* для улучшения структуры и устранения избыточного, дублированного кода.
7. Весь набор тестов выполняется успешно.
8. ~~Комплек~~ ~~т изменений~~, сделанных в этом цикле в тестах и программе ~~записаны~~ ~~в репозитории~~, после чего программа снова находится в согласованном состоянии и содержит четко осязаемое улучшение по сравнению с предыдущим состоянием.

Продолжительность каждого цикла — порядка нескольких минут.

# Что дает TDD?

- Актуальное описание намерений, дизайна и использования системы
- Легкое обнаружение слабых мест в дизайне
- Автоматическое регрессионное тестирование
- Безопасный рефакторинг
- Быстрое обнаружение дефектов

## Пример 1. Числа Фибоначчи



Последовательность Фибоначчи определяется следующим соотношением:

$$F(n) = F(n - 1) + F(n - 2)$$

$$F(1) = 1$$

$$F(0) = 0$$

Требуется написать функцию для определения  $n$ -го члена последовательности





## Пример 1. (продолжение)

Первый тест

```
[Test]
public void TestFirstFibonacciNumber()
{
    Assert.AreEqual(0,
MathUtils.Fibonacci(0));
}
```

Реализация функции

```
public class MathUtils
{
    public static uint Fibonacci(uint n)
```

# Пример 1. (продолжение)



The screenshot shows the NUnit application window titled "FibonacciNumbers.dll - NUnit". The interface includes a menu bar (File, View, Project, Test, Tools, Help), a tree view on the left showing the test hierarchy, and a main panel on the right for test execution. The tree view shows a path: D:\tihonove\Documents\Visual Studio 2010\SUSU\TDDDemo\Tests\Tests\TestFirstFibonacci. The main panel has "Run" and "Stop" buttons, a progress bar (100% green), and a summary: "Passed: 1 Failed: 0 Errors: 0 Inconclusive: 0 Invalid: 0 Ignored: 0 Skipped: 0 Time: 0.0680068". At the bottom, there are tabs for "Errors and Failures", "Tests Not Run", and "Text Output", and a status bar showing "Completed" and "Test Cases : 1 Tests Run : 1 Errors : 0 Failures : 0 Time : 0.0680068".

## Пример 1. (продолжение)



Проверяем второй член последовательности

```
public void TestSecondFibonacciNumber()  
{  
    Assert.AreEqual(1, MathUtils.Fibonacci(1));  
}
```

# Пример 1. (продолжение)



The screenshot shows the NUnit application window titled "FibonacciNumbers.dll - NUnit". The interface includes a menu bar (File, View, Project, Test, Tools, Help), a tree view on the left showing the test hierarchy, and a main results area on the right. The tree view shows the following structure:

- Tests
  - SUSU (failed)
  - TDDDemo (failed)
    - Tests (failed)
      - TestFirstFibonacciNumber (passed)
      - TestSecondFibonacciNumber (failed)

The main results area displays the following information:

- Buttons: Run, Stop
- Path: D:\tihonove\Documents\Visual Studio 2010\Projects\Tests\2.FibonacciNumbers\FibonacciNumbers\bin\Debug
- Progress bar: 20 red blocks, indicating 20 tests.
- Summary: **Passed: 1 Failed: 1 Errors: 0 Inconclusive: 0 Invalid: 0 Ignored: 0 Skipped: 0 Time: 0.1510151**
- Test Output:

```
SUSU.TDDemo.Tests.Tests.TestSecondFibonacciNumber:  
Expected: 1  
But was: 0
```
- Stack Trace:

```
at SUSU.TDDemo.Tests.Tests.TestSecondFibonacciNumber() in D:\tihonove\Documents\Visual Studio 2010\Projects\Tests\2.FibonacciNumbers\FibonacciNumbers\Tests.cs:line 22
```
- Navigation: Errors and Failures, Tests Not Run, Text Output
- Status Bar: Completed, Test Cases : 2, Tests Run : 2, Errors : 0, Failures : 1, Time : 0.1510151

# Пример 1. (продолжение)



Модифицируем функцию

```
public static uint Fibonacci(uint n)
{
    if (n == 0)
        return 0;
    else
        return 1;
}
```

# Пример 1. (продолжение)



The screenshot shows the NUnit application window titled "FibonacciNumbers.dll - NUnit". The interface includes a menu bar (File, View, Project, Test, Tools, Help) and a tree view on the left showing the test hierarchy: SUSU, TDDDemo, Tests, and two sub-tests: TestFirstFibonacciNumb and TestSecondFibonacciN. The main area contains a "Run" button, a "Stop" button, and a progress bar consisting of 20 green segments. Below the progress bar, the test results are displayed: "Passed: 2 Failed: 0 Errors: 0 Inconclusive: 0 Invalid: 0 Ignored: 0 Skipped: 0 Time: 0,0710071". At the bottom, there are tabs for "Errors and Failures", "Tests Not Run", and "Text Output", and a status bar showing "Completed" and summary statistics: "Test Cases : 2 Tests Run : 2 Errors : 0 Failures : 0 Time : 0,0710071".

## Пример 1. (продолжение)



Добавляем новые тесты, проверяющие третий и четвертый члены.

```
public void TestThirdFibonacciNumber()  
{  
    Assert.AreEqual(1, MathUtils.Fibonacci(2));  
}
```

```
public void TestFourthFibonacciNumber()  
{  
    Assert.AreEqual(2, MathUtils.Fibonacci(3));  
}
```

# Пример 1. (продолжение)



The screenshot shows the NUnit application window titled "FibonacciNumbers.dll - NUnit". The interface includes a menu bar (File, View, Project, Test, Tools, Help), a tree view on the left showing the test hierarchy, and a main results area on the right. The tree view shows a project "SUSU" containing a "TDDDemo" folder, which has a "Tests" folder. Inside "Tests", there are four test items: "TestFirstFibonacciNum" (passed), "TestFourthFibonacciNur" (failed), "TestSecondFibonacciN" (passed), and "TestThirdFibonacciNum" (passed). The main results area shows a progress bar with 3 green bars and 1 red bar. Below the progress bar, the summary text reads: "Passed: 3 Failed: 1 Errors: 0 Inconclusive: 0 Invalid: 0 Ignored: 0 Skipped: 0 Time: 0.1560156". The output window shows the following error message: "SUSU.TDDemo.Tests.Tests.TestFourthFibonacciNumber: Expected: 2 But was: 1". The stack trace below indicates the failure occurred in "D:\tihonove\Documents\Visual Studio 2010\Projects\Tests\3.FibonacciNumbers\FibonacciNumbers\Tests.cs:line 34". At the bottom of the window, the status bar shows "Completed" and "Test Cases : 4 Tests Run : 4 Errors : 0 Failures : 1 Time : 0.1560156".



# Пример 1. (продолжение)



Модифицируем функцию

```
public static uint Fibonacci(uint n)
{
    if (n == 0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

# Пример 1. (продолжение)



The screenshot shows the NUnit application window titled "FibonacciNumbers.dll - NUnit". The interface includes a menu bar (File, View, Project, Test, Tools, Help) and a tree view on the left showing the test hierarchy: SUSU, TDDDemo, Tests, and four individual test cases, all marked with green checkmarks. The main area features "Run" and "Stop" buttons, a progress bar of 20 green segments, and a summary: "Passed: 4 Failed: 0 Errors: 0 Inconclusive: 0 Invalid: 0 Ignored: 0 Skipped: 0 Time: 0,0760076". At the bottom, there are tabs for "Errors and Failures", "Tests Not Run", and "Text Output", and a status bar showing "Completed" and summary statistics: "Test Cases : 4 Tests Run : 4 Errors : 0 Failures : 0 Time : 0,0760076".

## Пример 2. Функция CombinePaths



Требуется реализовать функцию, складывающую два пути.

Например:

`C:\Data\ + MySQL\data.sql = C:\Data\MySQL\data.sql`

Необходимо учесть следующие варианты использования:

1. Разные вариации наличия/отсутствия завершающего и начального слэша в первом и втором путях соответственно
  - `C:\folder + file.txt`
  - `C:\folder + \file.txt`
  - `C:\folder\ + file.txt`
  - `C:\folder\ + \file.txt`
2. Если первый путь пуст
  - `" + file.txt`
3. Если второй путь абсолютный
  - `C:\folder + D:\folder2\file.txt`

## Пример 2. (продолжение)



### Первый тест

```
typ
e TCombinePathTests =
  class (TestCase)
    procedure TestCombineSimplePaths;
  end
;

procedure TCombinePathTests.TestCombineSimplePaths;
begin
  CheckEquals
  ( 'C:\file_name.txt'
    CombinePaths('C:', 'file_name.txt')
  )
;
end
;
```

## Пример 2. (продолжение)



### Реализация функции

```
function CombinePaths (const APath1 ,  
    APath2: string): string;  
begin  
  n Result := EmptyStr;  
end  
;
```

# Пример 2. (продолжение)



The screenshot shows the DUnit testing framework window. The title bar reads "DUnit: An Xtreme testing framework". The menu bar includes "File", "Test Tree", "Options", and "Actions". Below the menu bar is a toolbar with icons for test tree, checkboxes, and a play button. The "Test Hierarchy" section shows a tree structure with the following items checked:

- CombinePathsTest.exe
- TCombinePathTests
- TestCombineSimplePaths

Below the hierarchy is a progress bar and a score bar. The score bar shows "Score: 0%". Below the progress bar is a summary table:

Tests	Run	Failures	Errors	Elapsed
1	1	1	0	0:00:00.003

Below the summary table is a table of test results:

Test Name	Failure Type	Message	Location
TestCombineSimpl...	ETestFailure	expected: <C:\file_name.txt> but was:...	\$0048CAC4

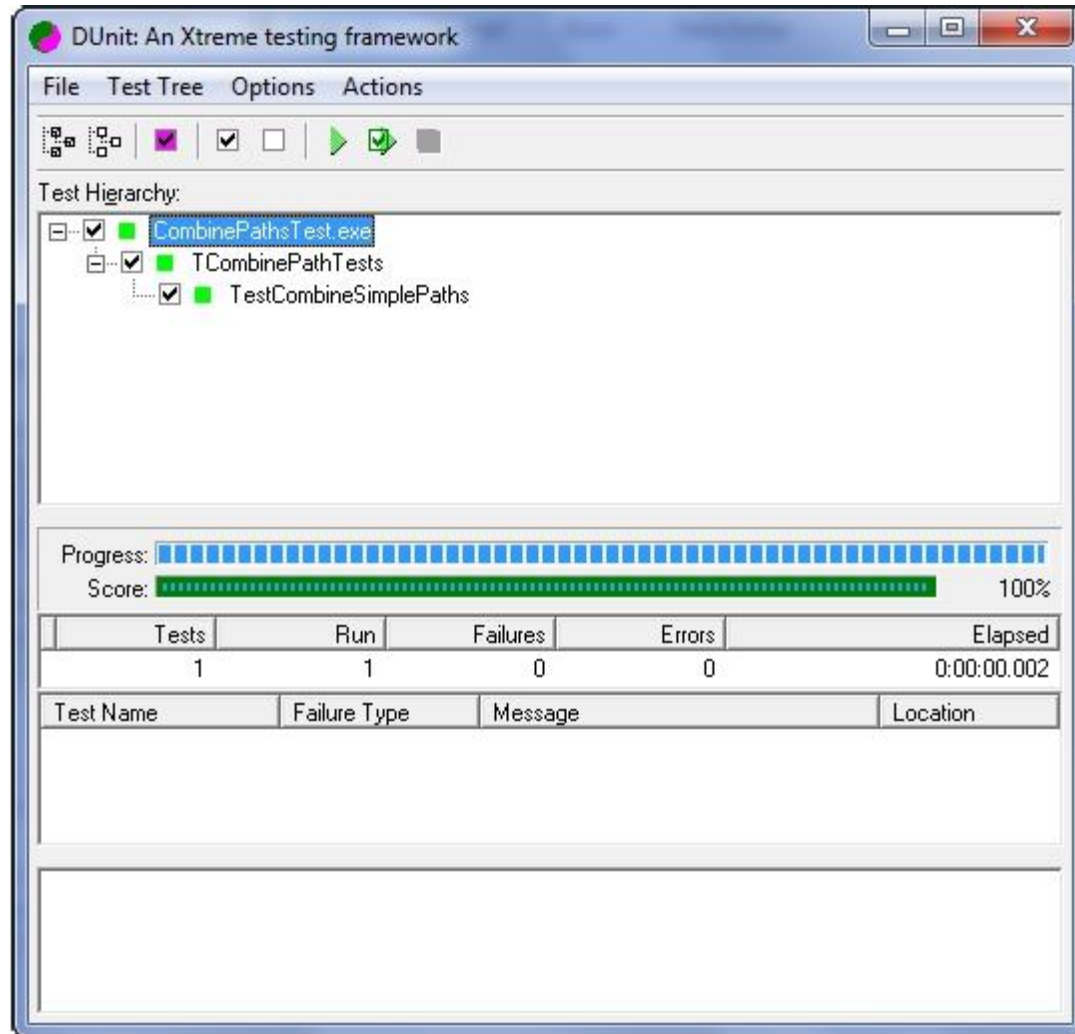
## Пример 2. (продолжение)



Модифицируем функцию

```
function CombinePaths (const APath1 ,  
    APath2: string) : string;  
begi  
n Result := APath1 + APath2 ;  
end  
;
```

# Пример 2. (продолжение)





## Пример 2. (продолжение)



```
procedure TCombinePathTests.  
    TestCombinePathsWithoutTrailingSlash  
begin  
n CheckEquals  
    ( C:\file_name.txt'  
      CombinePaths('C:', 'file_name.txt')  
    )  
;  
end  
;
```

# Пример 2. (продолжение)



DUnit: An Xtreme testing framework

File Test Tree Options Actions

Test Hierarchy:

- CombinePathsTest.exe
  - TCombinePathTests
    - TestCombineSimplePaths
    - TestCombinePathsWithoutTrailingSlash

Progress:

Score: 50%

Tests	Run	Failures	Errors	Elapsed
2	2	1	0	0:00:00.003

Test Name	Failure Type	Message	Location
TestCombinePaths...	ETestFailure	expected: <C:\file_name.txt> but was:...	\$0048CAFO

## Пример 2. (продолжение)



Модифицируем функцию

```
function CombinePaths (const APath1 ,
    APath2 : string) : string;
begin
    n Result :=
        IncludeTrailingPathDelimiter (APath1)
        +
end APath2 ;
;
```

# Пример 2. (продолжение)



DUnit: An Xtreme testing framework

File Test Tree Options Actions

Test Hierarchy:

- CombinePathsTest.exe
  - TCombinePathTests
    - TestCombineSimplePaths
    - TestCombinePathsWithoutTrailingSlash

Progress:

Score: 100%

Tests	Run	Failures	Errors	Elapsed
2	2	0	0	0:00:00.002

Test Name	Failure Type	Message	Location
-----------	--------------	---------	----------

## Пример 2. (продолжение)



```
procedure TCombinePathTests.  
    TestCombinePathsWithEmptyFirstPath  
begin  
n CheckEquals  
    ( 'file_name.txt'  
      CombinePaths('', 'file_name.txt')  
    )  
;  
end  
;
```

# Пример 2. (продолжение)



DUnit: An Xtreme testing framework

File Test Tree Options Actions

Test Hierarchy:

- CombinePathsTest.exe
  - TCombinePathTests
    - TestCombineSimplePaths
    - TestCombinePathsWithoutTrailingSlash
    - TestCombinePathsWithEmptyFirstPath

Progress:

Score: 66%

Tests	Run	Failures	Errors	Elapsed
3	3	1	0	0:00:00.004

Test Name	Failure Type	Message	Location
TestCombinePaths...	ETestFailure	expected: <file_name.txt> but was: <\...	\$0048CBCD

## Пример 2. (продолжение)



Модифицируем функцию

```
function CombinePaths(const APath1,  
    APath2: string): string;  
begin  
  if APath1 = EmptyStr then  
    Result := APath2  
  else  
    Result :=  
      IncludeTrailingPathDelimiter(APath1) +  
      APath2;  
end  
;
```

# Пример 2. (продолжение)



DUnit: An Xtreme testing framework

File Test Tree Options Actions

Test Hierarchy:

- CombinePathsTest.exe
  - TCombinePathTests
    - TestCombineSimplePaths
    - TestCombinePathsWithoutTrailingSlash
    - TestCombinePathsWithEmptyFirstPath

Progress:

Score: 100%

Tests	Run	Failures	Errors	Elapsed
3	3	0	0	0:00:00.002

Test Name	Failure Type	Message	Location
-----------	--------------	---------	----------



## Пример 2. (продолжение)



### Окончательный вариант функции

```
function CombinePaths(const APath1,  
    APath2: string): string;  
begin  
    if IsAbsolutePath(APath2) or (APath1 = EmptyStr) then  
        Result := APath2  
    else  
        Result :=  
            IncludeTrailingPathDelimiter(APath1) +  
            DeletePrecedingPathDelimiter(APath2);  
    end  
;
```

# Приёмы (паттерны) TDD



- Изолированный тест (Isolated Test)
- Список тестов (Test List)
- Сначала утверждение (Assertion First)
- Тестовые данные (Test Data)
- Понятные данные (Evident Data)
- Arrange – Act – Assert (AAA)

# Изолированный тест



- Если не проходит один тест, другие не должны свалиться вслед за ним.
- Если тесты изолированы, порядок их выполнения значения не имеет.
- Тесты не должны использовать общие ресурсы. Общие ресурсы, используемые тестами, не должны изменяться в ходе тестирования.

# Список тестов



- Запишите все тесты, которые хотите реализовать, и придерживайтесь этого списка.
- От зелёной полосы всегда должен отделять один тест, поэтому не стоит сразу программировать тесты.
- Тесты, в которых возникает необходимость в процессе написания другого, просто занесите в список.

# Сначала утверждение



- Такой подход позволяет мгновенно ответить на два важных вопроса: «Что считать правильным результатом теста?» и «Каким образом можно проверить его правильность?».
- Сначала мы определяем, что нужно получить, а потом создаем необходимые условия, чтобы `assert` прошел.
- В тесте не должно быть слишком много утверждений (идеальный вариант — один,

# Тестовые данные



- Не используйте одинаковые данные. Если нет разницы между 1 и 2, берите 1.
- Если 3 набора дадут тот же результат, что и 10, используйте 3.
- Используйте реалистичные тестовые данные.

# Понятные данные



- При тестировании должно быть очевидно, откуда берется тот или иной результат.
- Не прячьте вычисления за константами, так будет понятно, что же нужно запрограммировать.
- Код теста должен читаться с первого раза.

# Понятные данные (пример)



```
Bank bank = new
Bank().addRate("USD", "GBP",
STANDARD_RATE);
MonkeyResults r = bank(STANDARD_COMMISSION);
assertEquals(new Note(49.25, "GBP"), result);
Или более очевидно:
```

```
Bank bank = new
Bank().addRate("USD", "GBP",
2);
MonkeyResults r = bank(0.015);
assertEquals(new Note(100, "USD"), "GBP");
assertEquals(new Note(100 / 2 * (1 - 0.015), "GBP"),
result);
```



# Arrange – Act – Assert



[Test]

```
public void TestTranslate()
{
    // Arrange.
    // Здесь выставляются начальные условия
    ITranslator translator = new EngRusTranslator();

    // Act.
    // Обработка тестируемого функционала.
    string result = translator.Translate("Hello, World!");

    // Assert.
    // Сверка ожидаемых значений с полученными
    Assert.AreEqual("Привет, Мир!", result);
}
```

# Arrange – Act – Assert (пример 2)



```
// Arranging
```

```
var annualSalary = 120000M;
```

```
var period = 3; // for a quarter profit
```

```
var calc = new SalaryCalculator();
```

```
// Acting
```

```
var result = calc.CalculateProfit(annualSalary, period);
```

```
// Assertion
```

```
Assert.AreEqual(40000, result);
```

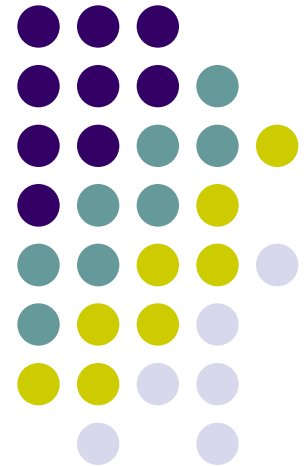
# Arrange – Act – Assert



Преимущества использования этого паттерна:

- Assert-методы никогда не перемешаются с Act-методами
- Неявное навязывание писать ОДИН Assert на ОДИН тест
- Упрощенный рефакторинг, Вам легко будет обнаружить Arrange-блоки, которые можно вынести в SetUp-метод

# Инструменты unit-тестирования



# Список инструментов



- Java: JUnit;
- C++: CppUnit, Boost Test;
- Delphi: DUnit;
- PHP: PHPUnit;
- C#: NUnit.

Полный список:

[http://en.wikipedia.org/wiki/List\\_of\\_unit\\_testing\\_frameworks](http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks)

# BDD

- BDD (behavior-driven development) — расширение подхода TDD к разработке и тестированию, при котором особое внимание уделяется поведению системы/модуля в терминах бизнеса(заказчика).
- Как правило, такие тесты иллюстрируют и тестируют различные сценарии, которые интересны непосредственно клиенту системы. В связи с этим при составлении таких тестов часто используется фреймворки, обладающие синтаксисом, обеспечивающим читаемость тестов не только программистом, но и представителями заказчика

# На каких принципах базируется TDD?

- «Делай проще, дурачок» (*keep it simple, stupid, KISS*)
- «Вам это не понадобится» (*you ain't gonna need it, YAGNI*)
- «Подделай, пока не сделаешь» (*fake it till you make it*)

# Выводы

- Гораздо больше времени уходит на реализацию простого класса, чем это требуется при написании кода «в лоб» (прямой реализации). Вот чего боятся менеджеры и программисты, которые не владеют TDD
- Вы всегда можете сравнить первую и последнюю реализации, используя TDD. Главной разницей будет то, что последняя реализация будет действительно объектно-ориентированной. Вы можете работать с модулем как с независимым объектом, запрашивать его состояние и пытаться исполнить отдельные операторы. Поэтому код, который будет работать с таким классом, сам будет более объектно-ориентированным.



# Выводы

- Помимо самого класса вы получите полный набор тестов для него. Для разработчика нет больше преимущества, чем полное покрытие кода тестами. Если тесты пишутся после кода, будет сложно обеспечивать полное покрытие: вы забудете написать тест на какую-нибудь функцию, что-то покажется слишком легким, чтобы писать на него тест и т.д. Такие тесты часто сложные, потому что вы хотите проверить сразу несколько аспектов в одном тесте. Используя TDD, вы получите набор простых понятных тестов, которые будет легко поддерживать.
- Вы получите живую документацию на код. Любой может прочитать названия тестовых методов и понять их смысл, цель и поведение. Будет гораздо легче понять код, имея такую информацию, вам не нужно будет отвлекать коллег просьбами объяснить их код.

# Выводы

- При использовании TDD легче обдумывать, что вы хотите получить от класса, его поведение и варианты использования. Имея хорошее понимание уже разработанных компонентов, можно понять, как написать более сложные компоненты.
- При использовании TDD вы реально оцениваете сложность задачи, всегда знаете, когда нужно закончить. Если у вас нет тестов, у вас часто будет появляться ощущение что все уже написано. А потом окажется, что вам нужно много времени на исправление ошибок и модификацию кода.

# Выводы

- Самое главное: уменьшаются зависимости между классами, увеличивается сцепление классов!

# Вопросы для самостоятельного изучения

- В каких задачах методология TDD не применима?
- Чем отличаются тесты, созданные по методологии TDD, от модульного тестирования? От интеграционного тестирования?
- Когда удобно использовать fake- и mock-объекты при использовании TDD?

# Дополнительный материал

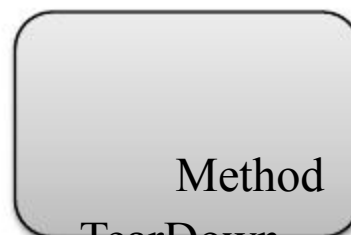
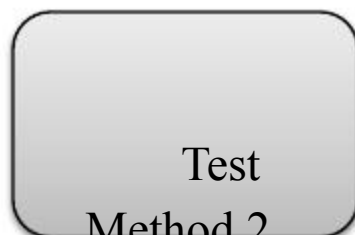
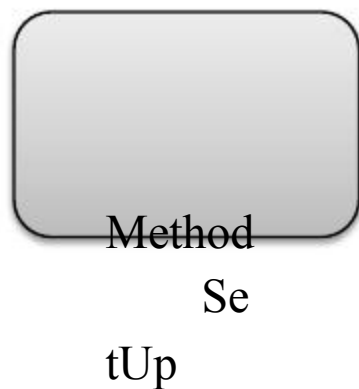
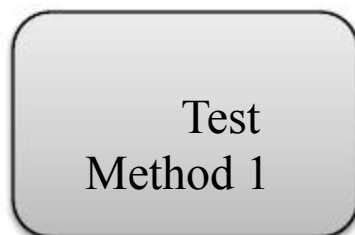
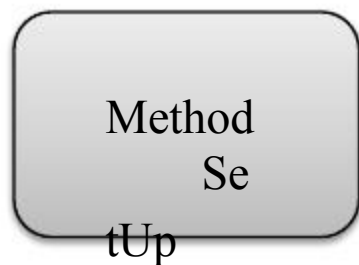
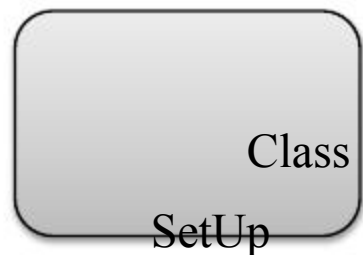
- Описание терминологии и технологии разработки через тестирование в различных инструментах:
  - xUnit
  - Junit
  - NUnit
  - CppUnit
  - DUnit

# xUnit. Терминология



- **Тестовый метод (test method)**  
Метод, в котором выполняется проверка работы тестируемого объекта.
- **Утверждение (assertion)**  
Метод сравнения ожидаемых и фактических результатов.
- **Фикстура уровня метода (method fixture)**  
Набор операций, выполняемый до и после каждого тестового метода.
- **Фикстура уровня класса (class fixture)**  
Набор операций, выполняемый до и после всех

# xUnit. Порядок вызова



# JUnit 3



- JUnit — библиотека для тестирования программного обеспечения на языке Java.
- Создана Кентом Беком и Эриком Гаммой
- Берёт своё начало в SUnit (тестовая среда для Smalltalk)



# JUnit 3. Организация тестов



## Класс TestCase

- Разработчик наследует свои классы тестов от этого класса.
- Тестовые методы — все методы класса, имя которых начинается с **test**.
- Класс TestCase реализует паттерн Шаблонный Метод (Template Method) и ссылается на виртуальные методы:
  - setUp()
  - runTest()
  - tearDown()

# JUnit 3. Организация тестов



## Класс TestSuite

- Используется для формирования набора тестов.
- Наборы тестов могут включать в себя одиночные тесты и другие наборы тестов.
- Реализует паттерн Компоновщик (Composite).
- Разработчику предлагается переопределять метод `suite()`.

# JUnit 3. Утверждения



## Класс Assert

- Содержит специальные методы сравнения
  - Сравнение на равенство  
(`assertEquals`)
  - Сравнение на истинность/ложность  
(`assertTrue`/`assertFalse`)
  - Сравнение с нулевым указателем  
(`assertNull` и `assertNotNull`)
  - Сравнение на идентичность/неидентичность  
(`assertSame`/`assertNotSame`)
- Является базовым классом для `TestCase`

# JUnit 3. Пример



```
public class TestGame extends TestCase
{
    private Game g;

    public void setUp()
    {
        g = new Game();
    }

    public void testTwoThrowsNoMark()
    {
        g.add(5)
        ;
        assertEquals(9,
        g.score());
    }
}
```

# NUnit



- NUnit — открытая среда юнит-тестирования приложений для .NET (включая платформу Mono).
- Портирован с языка java и написан на J#.
- Новые версии написаны на C# с учётом таких новшеств как атрибуты.
- Текущая версия: 2.5.7.

# NUnit. Организация тестов



Для организации тестов используются атрибуты:

- **[Test]**  
помечает тестовый метод.
- **[TestFixture]**  
помечает класс с набором тестов.
- **[SetUp], [TearDown]**  
помечает любую процедуру без параметров как фикстуру уровня метода.

# NUnit. Утверждения



## Класс Assert

- Класс содержит статические методы проверки фактических значений с ожидаемыми:
  - AreEqual, AreNotEqual.
  - AreSame, AreNotSame.
  - IsTrue, IsFalse.
  - Greater, GreaterOrEqual, и т.п.
  - IsNotNull, IsNull.

# NUnit. Пример



```
[TestFixture
public class TestGame {
    private Game game;

    [SetUp
public void SetUp() {
    game = new Game();
}

[Test
public void TestTwoThrowsNoMark() {
    game.Add(5)
    ;
    game.Add(4);
    Assert.AreEqual(9, game.GetScore());
}
}
;
```



# NUnit. Утверждения



С версии NUnit 2.4 введены constraint-based утверждения.

Новый тип утверждений базируется на статической функции `Assert.That()`

```
Assert.That(object actual,  
            IResolveConstraint constraint,  
            string message);
```

# NUnit. Утверждения



Примеры использования constraint-based утверждений:

```
Assert.That(myString, Is.EqualTo("Hello"));
```

```
Assert.That(7, Is.GreaterThanOrEqualTo(3));
```

```
Assert.That(phrase,  
    Contains.Substring("tests fail"));
```

```
Assert.That(phrase,  
    Is.Not.StringContaining("tests pass"));
```

```
Assert.That(3, Is.LessThan(5) | Is.GreaterThan(10));
```

# CppUnit



- CppUnit – библиотека тестирования для языка C++.
- Является портом с JUnit на C++.
- Текущая версия: 1.12.1

# CppUnit. Организация тестов



- **CPPUNIT\_TEST\_SUITE,**  
**CPPUNIT\_TEST\_SUITE\_END**  
создают набор тестов.
- **CPPUNIT\_TEST**  
создаёт тестовый метод.
- **setUp(), tearDown()**  
фикстуры уровня метода, виртуальные функции класса `TestFixture`.

# CppUnit. Утверждения



- Проверочные методы реализованы в виде макросов:
  - CPPUNIT\_ASSERT
  - CPPUNIT\_ASSERT\_EQUAL
  - CPPUNIT\_ASSERT\_DOUBLES\_EQUAL
  - CPPUNIT\_ASSERT\_THROW
  - CPPUNIT\_ASSERT\_NO\_THROW

# CppUnit. Пример



```
class TestGame : public CPPUNIT_NS::TestFixture
{
    CPPUNIT_TEST_SUITE(TestGame)
    CPPUNIT_TEST(testTwoThrowsNoMark)
    ;
protected:
    CPPUNIT_TEST_SUITE_END();
    : Game * game;
protected
: void testTwoThrowsNoMark();
public
: void setUp();
  void tearDown();
}
;
```

# CppUnit.



## Пример

```
void TestGame::setUp() {  
    game = new Game();  
}
```

```
void TestGame::tearDown() {  
    delete game;  
    game->Add(5);  
    game->Add(4);  
    CPPUNIT_ASSERT( game->GetScore() == 9 );  
}
```

# JUnit 4



- JUnit 4 — новая версия библиотеки, построенная на появившихся в Java 5 *аннотациях*.
- Текущая версия: 4.8.2



# JUnit 4. Организация тестов



- Набор тестов помещается в отдельный класс.
- Для организации тестов используются

аннотации:

- **@Test**  
помечает тестовый метод.
- **@Before, @After**  
помечает любую процедуру без параметров как фикстуру уровня метода.
- **@BeforeClass, @AfterClass**  
помечает любую процедуру без параметров как фикстуру уровня класса

# JUnit 4. Утверждения



## Класс Assert

- Содержит набор статических методов,
  - аналогичный набору JUnit 3
    - assertEquals,
    - assertSame/assertNotSame,
    - assertEquals,
    - assertFalse/assertTrue,
    - assertNull/assertNotNull.

# JUnit 4. Пример 1



```
public class TestGame
{
    @Test
    public void
    testTwoThrowsNoMark()
    {
        g = new
        Game();
        g.add(5);
        Assert.assertEquals(9,
        g.add(4),
        g.score());
    }
}
```

# JUnit 4. Пример 2



```
public class TestGame
{
    private Game g;

    @Before
    public void setUp()
    {
        g = new Game();
    }

    @Test
    public void twoThrowsNoMark()
    {
        g.add(5)
        ;
        g.add(4);
        assertEquals(9,
        g.score());
    }
}
```

# DUnit



- Dunit — инструмент тестирования для среды Borland Delphi.
- Первоначальная версия написана Juanco Anez в 2000г.
- DUnit стал стандартной частью в среде разработки Delphi 2005.

# DUnit. Организация тестов



## Класс TTestCase

- Разработчик наследует свои классы тестов от этого класса.
- Тестовые методы — это все методы без параметров, которые находятся в `published` секции класса.
- Фикстуры уровня метода определяются перегрузкой виртуальных функций `SetUp` и `TearDown`.

# DUnit. Утверждения



Класс `TTestCase` содержит набор проверочных методов:

- `CheckEquals/CheckNotEquals`
- `CheckNull/CheckNotNull`
- `CheckSame`
- `CheckIs`
- `CheckInherits`

# DUnit. Пример



```
typ
e TGameTest = class(TTestCase)
  private
    FGame: TGame;
  protecte
  d procedure SetUp; override;
    procedure TearDown; override;
  publishe
  d procedure TestTwoThrowsNoMark;
end
;
```



# DUnit. Пример



```
{ TGameTest }  
procedure  
TGameTest.SetUp;  
begin inherited  
    FGame := TGame.Create;  
end  
;  
procedure  
TGameTest.TearDown;  
begin FGameAndNil(FGame)  
    ;  
end inherited;  
;  
procedure  
TGameTest.TestTwoThrowsNoMark;  
begin FGame.Add(5)  
    ;  
    FGame.Add(4)9,  
end FGame.Score);  
.
```