TDD – Test-Driven Development

Разработка через тестирование

Разработка через тестирование

Техника разработки ПО, основывающаяся на повторении очень коротких циклов разработки:

1. Написать тест, покрывающий желаемое изменение.
2. Написать код, который позволяет пройти тест.
3. Выполнить рефакторинг.

Тест

Тест – это процедура, которая позволяет либо подтвердить, либо опровергнуть работоспособность кода.

Тесты бывают ручные и автоматические.

Ручное тестирование

Ручное тестирование состоит из двух этапов:

- Стимулирование кода.
- Проверка результата.

Автоматическое тестирование

Вместо программиста стимулирование кода и проверкой результатов занимается компьютер, который отображает на экране результат выполнения теста:

- Код работоспособен

  Или

- Код неработоспособен.

Инверсия ответственности

От логики тестов и от их качества зависит, будет ли код соответствовать техническому заданию.

Модульное тестирование

- Юнит-тестирование (unit testing) – процесс, позволяющий в автоматическом режиме проверить на корректность отдельные модули программы.

Методика TDD

Методика TDD заключается, в основном, в организации автоматических тестов (unit testing) и выработке определенных навыков тестирования.

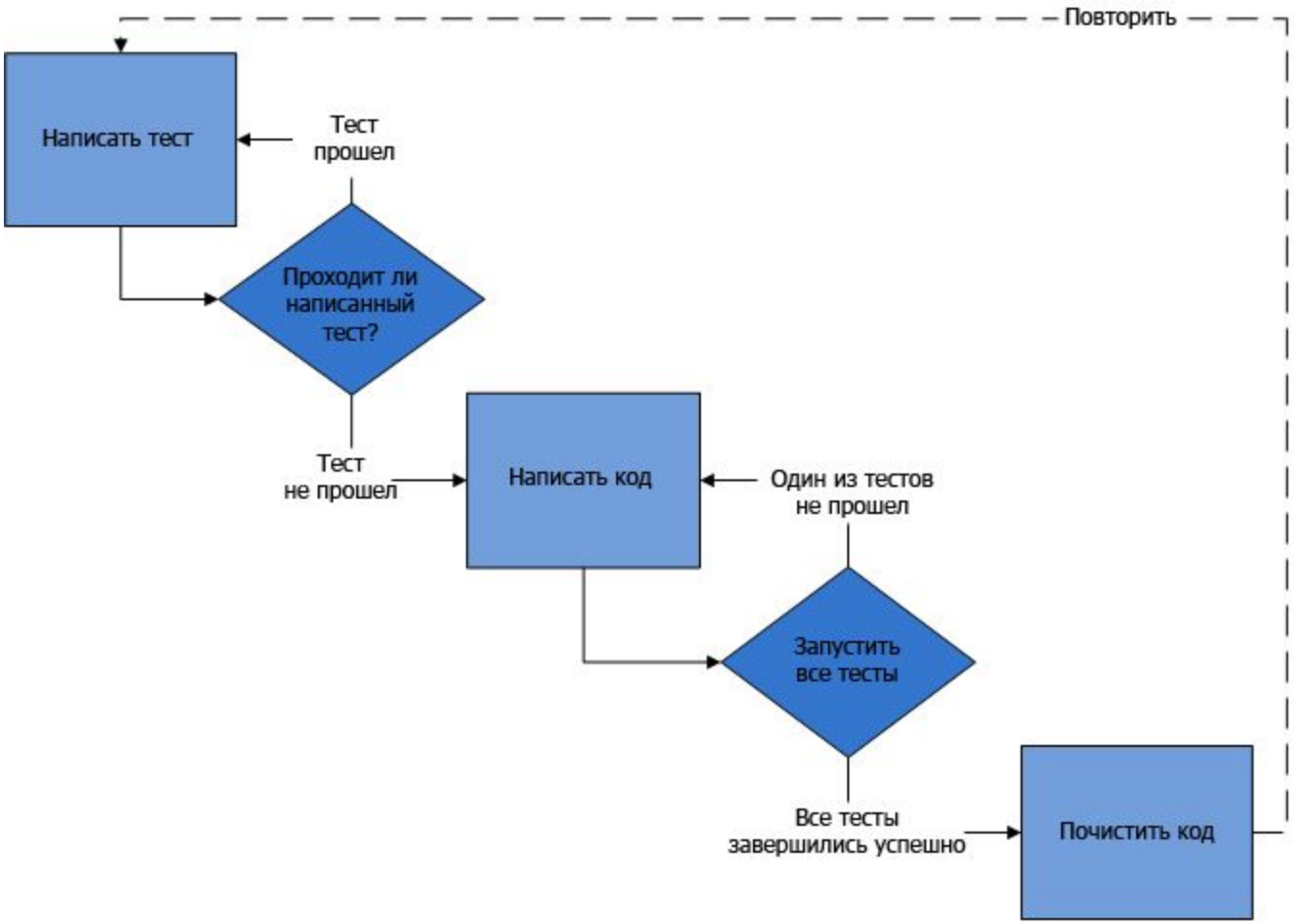
Одной из особенностей является написание тестов ДО написания кода.

Test-first
Сначала тест

Мантра TDD

- Написать тест;
- Добиться, чтобы тест сработал;
- Устранить дублирование (выполнить *рефакторинг*).

Цикл разработки TDD



Связанные принципы

- KISS – keep it simple, stupid (делай проще, тупица). Keep it short and simple (делай короче и проще).
- YAGNI – you ain’t gonna need it (вам это не понадобится).

Изменение кода без изменения функциональности

РЕФАКТОРИНГ

Рефакторинг

- Процесс изменения внутренней структуры программы, не затрагивающий её внешнего поведения и имеющий целью облегчить понимание её работы.

Не путать РЕФАКТОРИНГ с оптимизацией производительности и реинжинирингом!

Причины применения рефакторинга

Рефакторинг применяется постоянно при разработке кода. Основными стимулами являются:

1. Необходимо добавить новую функцию, которая недостаточно укладывается в принятое архитектурное решение.
2. Необходимо исправить ошибку, причины возникновения которой сразу не ясны. (*Плохой код*).
3. Преодоление трудностей в разработке, которые обусловлены сложной логикой программы. (*Плохой код*).

Признаки плохого кода

1. Дублирование кода;
2. Длинный метод;
3. Большой класс;
4. Длинный список параметров;
5. «Завистливые» функции и «завистливые» классы;
6. Избыточные временные переменные;
7. Классы данных;
8. Несгруппированные данные.
9. И т.д…

Мультивалютные деньги

TDD НА ПРИМЕРЕ

Кент Бек. Экстремальное программирование: разработка через тестирование

Test-driven Development by Example

ISBN 5-8046-0051-6, 0-321-14653-0; 2003 г.

Отчет о портфеле акций

Компания	Кол-во акций	Цена, USD	Сумма, USD
Google	100	750	75 000
Facebook	1000	20	20 000
		ИТОГО:	95 000

Отчет о портфеле акций

Компания	Кол-во акций	Цена	Сумма
Google	100	750 USD	75 000 USD
Facebook	1000	20 USD	20 000 USD
Газпром	500	4500 руб.	2 250 т.р.
		ИТОГО:	170 000 USD*

*1 USD = 30 руб

Требуемая функциональность

- Сложение величин в одной валюте;
- Умножение величины в одной валюте (стоимость акции) на число (количество акций). Результатом должна быть величина в валюте;
- Сложение двух величин в разных валютах и конвертировать результат с учетом курса обмена.

ToDo List

- $5 + 150 RUR = $10, если курс 1:30
- **$5 * 2 = $10**

Сначала тест!

```
[Test]
public void MultiplicationTest()
{
    Dollar five = new Dollar(5);
    five.times(2);

    Assert.AreEqual(five.Amount, 10);
}
```

Тоже самое, но без NUnit

```
public static void MultiplicationTest()
{
    Dollar five = new Dollar(5);
    five.times(2);

    Debug.Assert(five.Amount == 10);
}
```

Подключить System.Diagnostics

```
using System.Diagnostics;
```

Вызвать функцию теста

```
using System.Diagnostics;
namespace TDD_Demo2
{
    public class Program
    {
        static void Main(string[] args)
        {
            MultiplicationTest();
        }

        public static void MultiplicationTest()
        {
            Dollar five = new Dollar(5);
            five.times(2);

            Debug.Assert(five.Amount == 10);
        }
    }
}
```

ТЕСТ -> КОД -> РЕФАКТОРИНГ

TDD ШАГ ЗА ШАГОМ

Наш первый тест

```
[Test]
public void MultiplicationTest()
{
    Dollar five = new Dollar(5);
    five.times(2);

    Assert.AreEqual(five.Amount, 10);
}
```

```
[Test]
public void MultiplicationTest()
{
    Dollar five = new Dollar(5);
    five.times(2);

    Assert.AreEqual(five.Amount, 10);
}
```

Не компилируется – 4 ошибки!

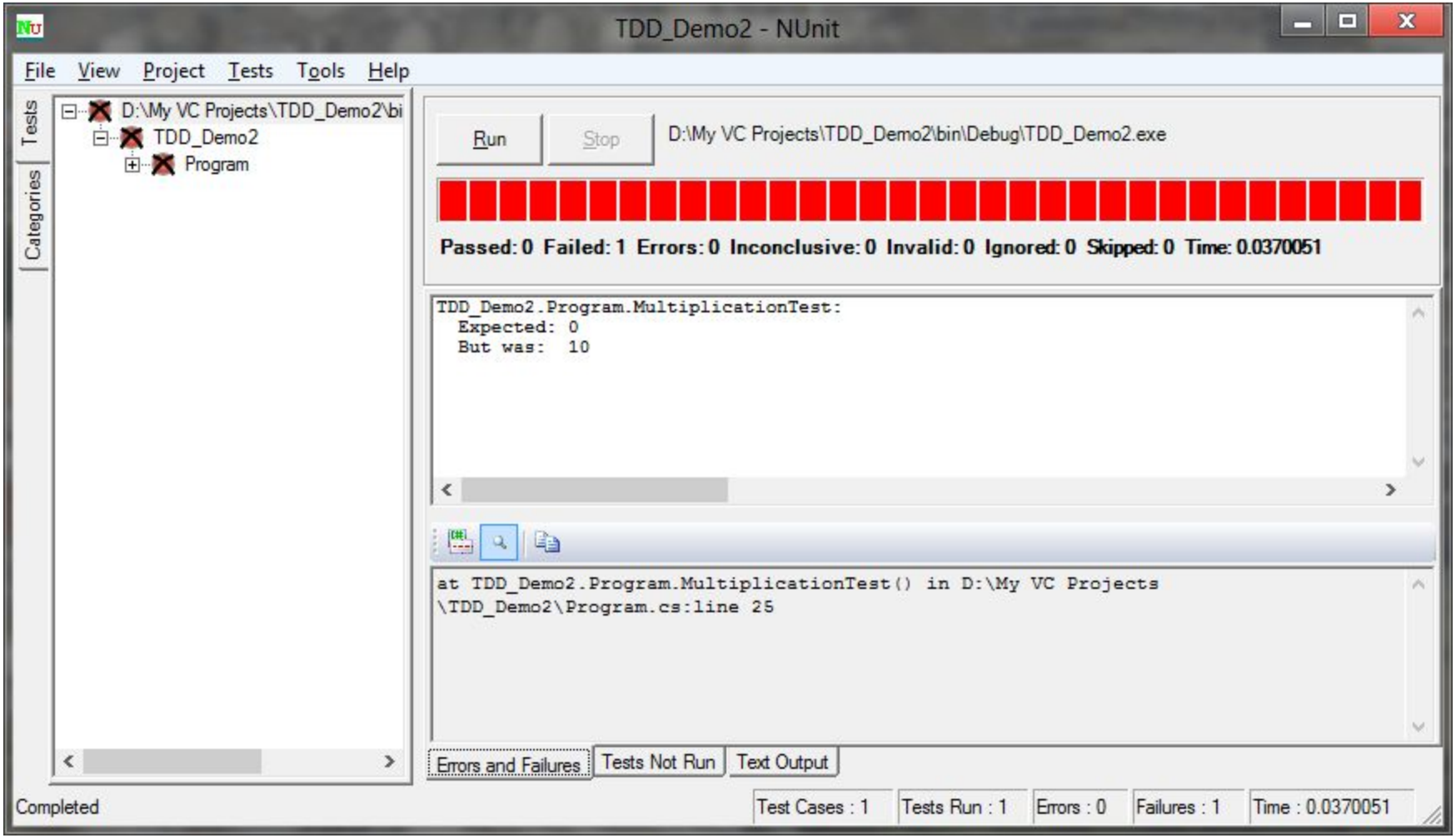
1. Нет класса Dollar;
2. Нет конструктора;
3. Нет метода times()
4. Нет переменной класса Amount

Маленькие шаги

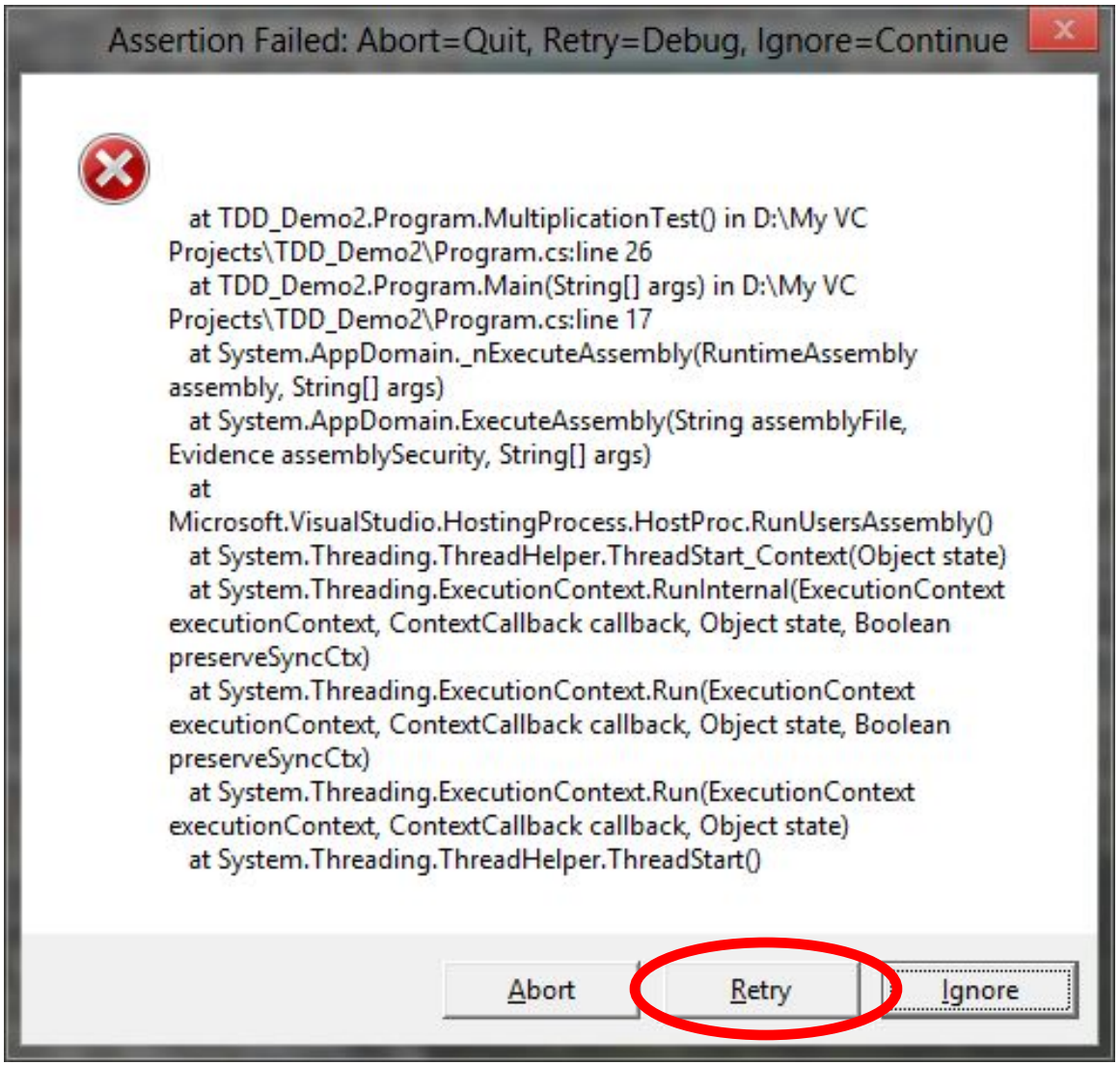
Добавим класс Dollar

```
public class Dollar
{
    public Dollar(int amount)
    {
    }

    public int Amount;

    public void times(int n)
    {
    }
}
```

Результат выполнения теста



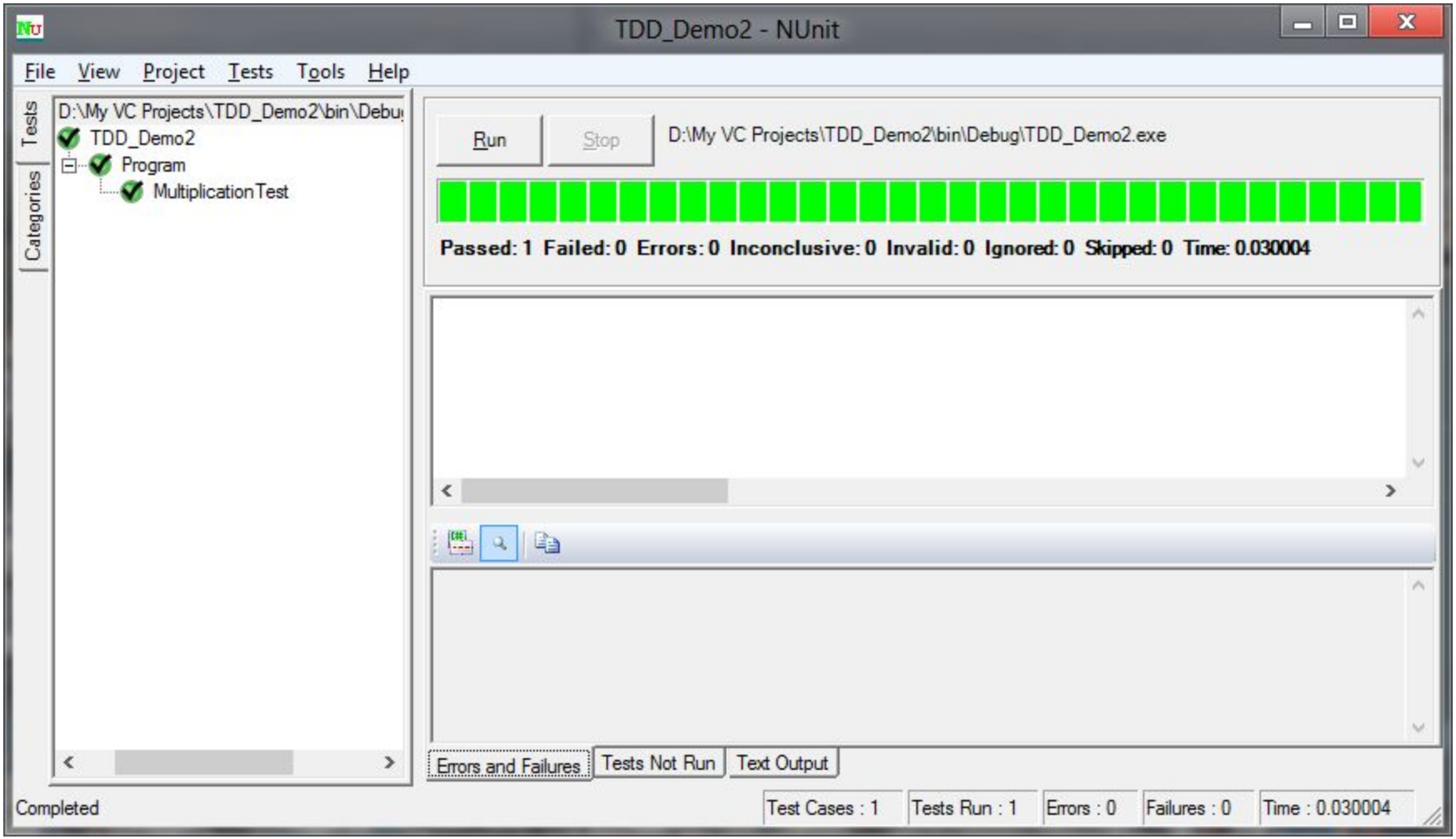
Сообщение Assert (без NUnit)



Зеленая полоса как можно быстрее!

```
public class Dollar
{
    public Dollar(int amount)
    {
    }

    public int Amount = 10;

    public void times(int n)
    {
    }
}
```

Успешный тест – зеленая полоса!



Полный цикл TDD

1. Добавить небольшой тест.
2. Запустить все тесты, при этом обнаружить, что что-то не срабатывает.
3. Внести небольшое изменение.
4. Снова запустить тесты и убедиться, что все они успешно выполняются.
5. Устранить дублирование с помощью рефакторинга.

Где дублирование?

```
public class Dollar
{
    public Dollar(int amount)
    {
    }

    public int Amount = 5 * 2;

    public void times(int n)
    {
    }
}
```

Перенос умножения в функцию times()

```
public class Dollar
{
    public Dollar(int amount)...

    public int Amount = 0;

    public void times(int n)
    {
        Amount = 5 * 2;
    }
}
```

Где взять 5?

```
public class Dollar
{
    public Dollar(int amount)
    {
        this.Amount = amount;
    }

    public int Amount = 0;

    public void times(int n)...
}
```

Перепишем times, используя значение из Amount

```
public class Dollar
{
    public Dollar(int amount)...

    public int Amount = 0;

    public void times(int n)
    {
        Amount = Amount * 2;
    }
}
```

Где взять 2?

```
public class Dollar
{
    public Dollar(int amount)...

    public int Amount = 0;

    public void times(int n)
    {
        Amount = Amount * n;
    }
}
```

Последний штрих

```
public class Dollar
{
    public Dollar(int amount)
    {
        this.Amount = amount;
    }

    public int Amount = 0;

    public void times(int n)
    {
        Amount *= n;
    }
}
```

ToDo List

- $5 + 150 RUR = $10, если курс 1:30
- *$5 * 2 = $10* **<- ГОТОВО**
- Сделать переменную Amount закрытым членом класса
- Побочные эффекты в классе Dollar?
- Округление денежных величин?

Clean code that works © Ron Jeffries

ЧИСТЫЙ КОД, КОТОРЫЙ РАБОТАЕТ

Обычный цикл TDD

1. Напишите тест.

- Представьте, как будет реализована в коде воображаемая операция.
- Придумывая её интерфейс, опишите все элементы, которые, как вам кажется, понадобятся.
- Пример: в первом тесте мы добавили класс Dollar, функцию times и переменную-член Amount.

2. Заставьте тест работать

- Первоочередная задача – получить зеленую полосу.
- Если ОЧЕВИДНО простое и элегантное решение – создайте его.
- Если на реализацию такого решения потребуется время – ОТЛОЖИТЕ его. Просто отметьте, что к нему придется вернуться, когда будет решена основная задача – быстро получить зеленый индикатор.
- Противоречит правилам хорошей разработки?

3. Улучшите решение

- После того, как система работает, избавьтесь от прошлых прегрешений и вернитесь к хорошей разработке.
- Удалите дублирование (и другие огрехи) и быстро сделайте так, чтобы полоска снова стала зеленой.

Чистый код, который работает

1. Сначала мы получаем код, который работает.
2. Затем делаем из него чистый код.

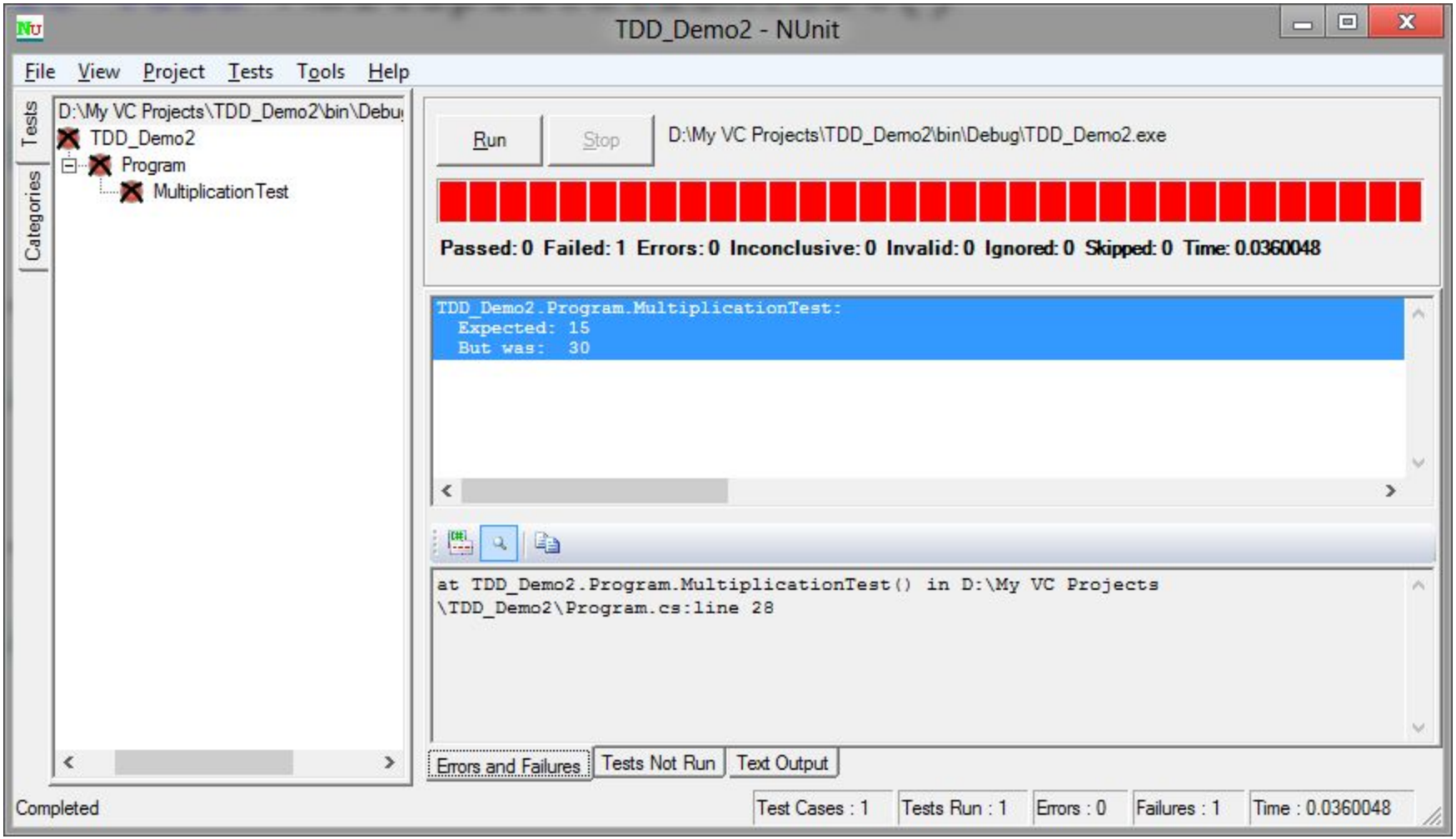
ToDo List

- $5 + 150 RUR = $10, если курс 1:30
- *$5 * 2 = $10*
- Сделать переменную Amount закрытым членом класса
- **Побочные эффекты в классе Dollar?**
- Округление денежных величин?

Вырождающиеся объекты

```
[Test]
public void MultiplicationTest()
{
    Dollar five = new Dollar(5);

    five.times(2);
    Assert.AreEqual(10, five.Amount);

    five.times(3);
    Assert.AreEqual(15, five.Amount);
}
```

Красная полоса: 30 вместо 15!



Сначала измените тест!

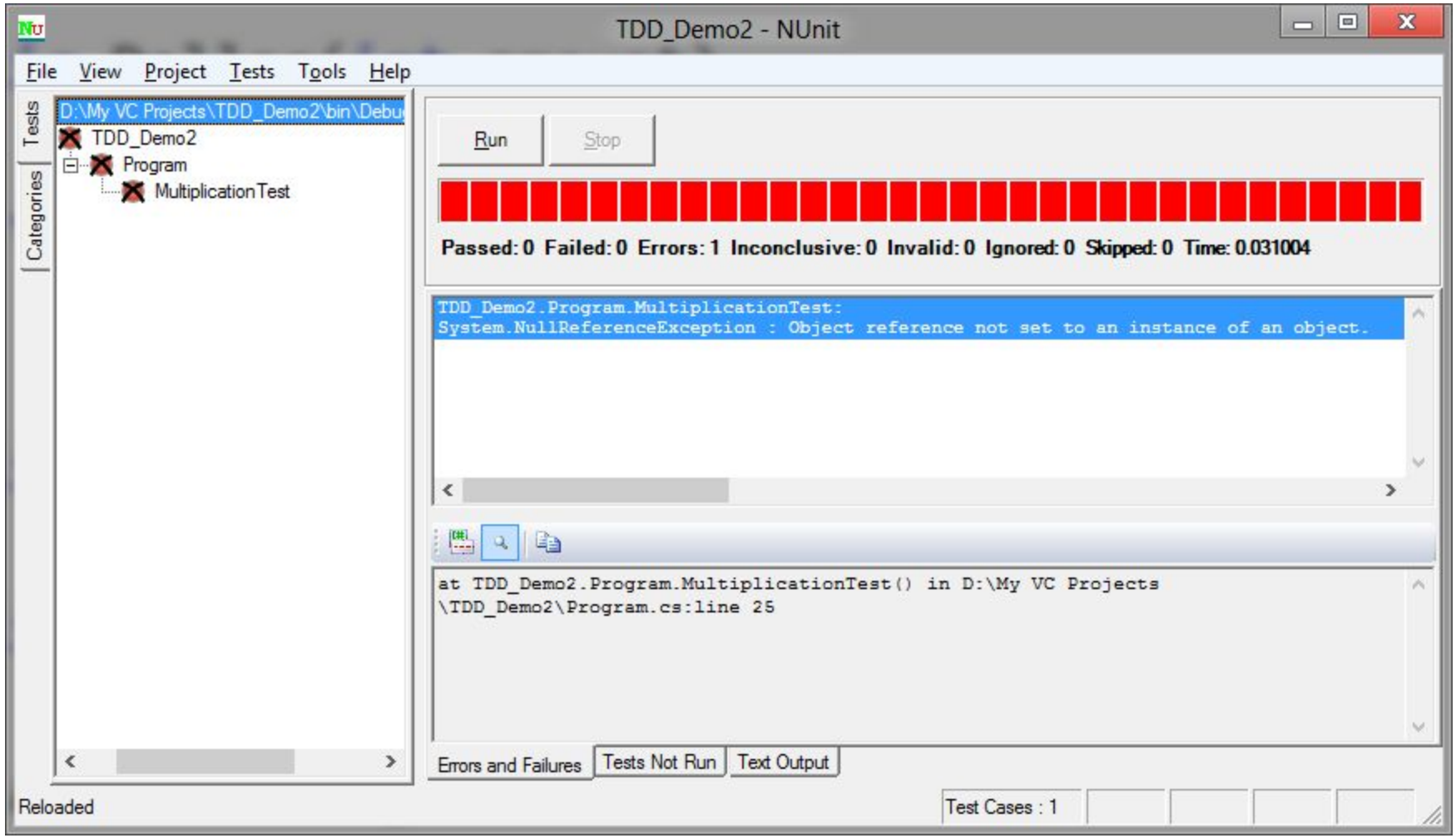
```
[Test]
public void MultiplicationTest()
{
    Dollar five = new Dollar(5);

    Dollar result = five.times(2);
    Assert.AreEqual(10, result.Amount);

    result = five.times(3);
    Assert.AreEqual(15, result.Amount);
}
```

Потом код

- Сначала сделайте, чтобы просто компилировалось.

```
public class Dollar
{
    public Dollar(int amount)
    {
        this.Amount = amount;
    }

    public int Amount = 0;

    public Dollar times(int n)
    {
        Amount *= n;
        return null;
    }
}
```

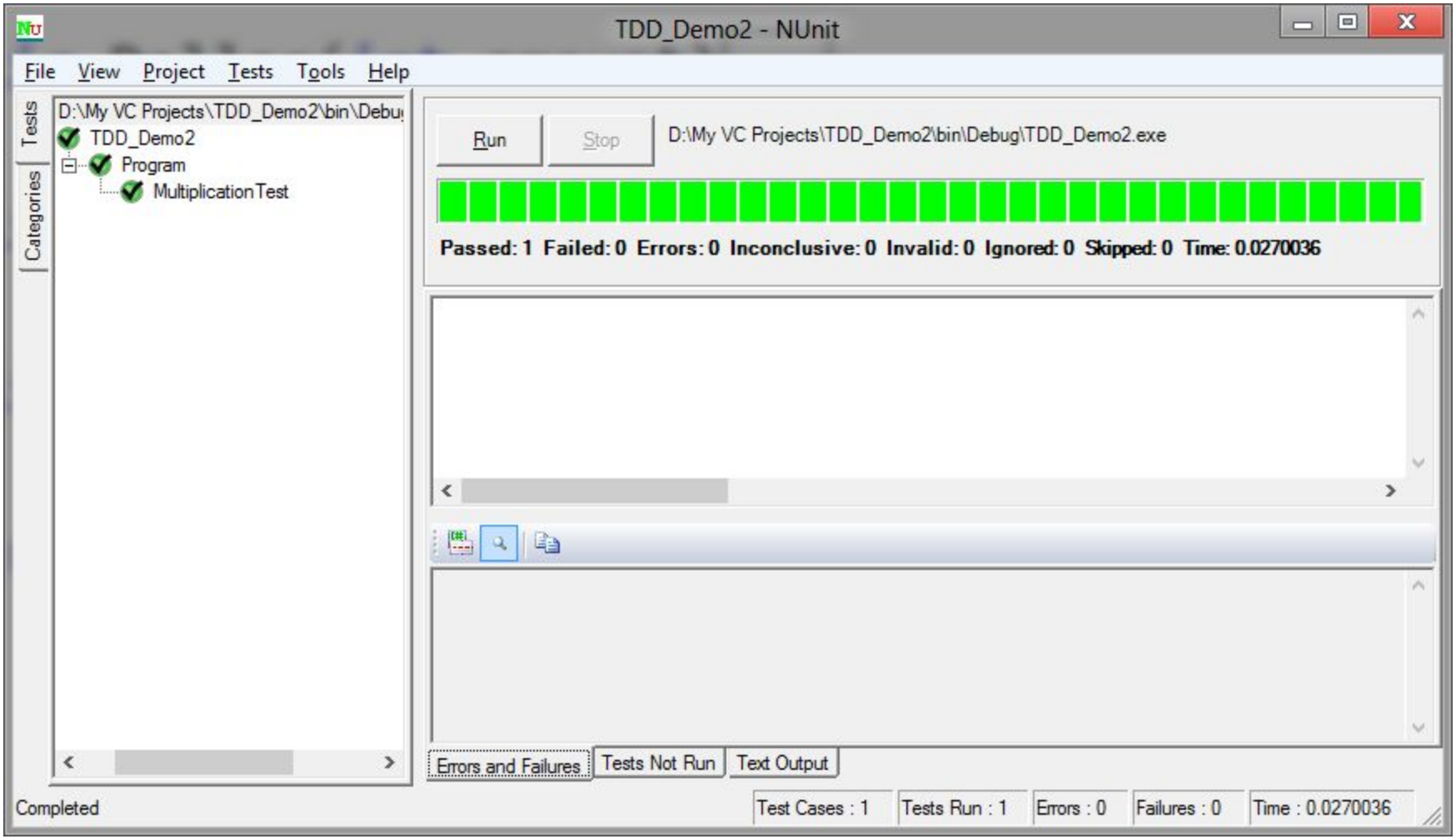
Компилируется, но полоса снова красная. А это тоже прогресс!



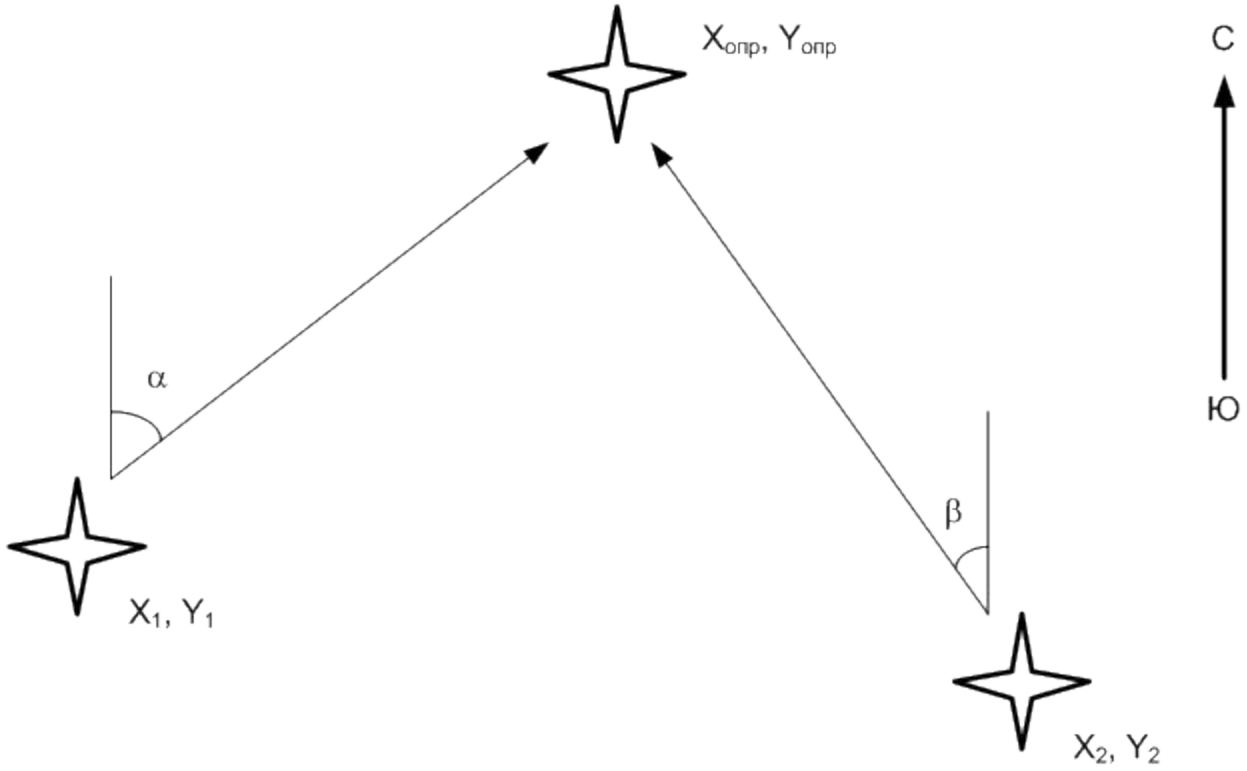
times() возвращает новый объект

```
public class Dollar
{
    public Dollar(int amount)...

    public int Amount = 0;

    public Dollar times(int n)
    {
        return new Dollar(Amount * n);
    }
}
```

Наш новый тест работает!



Триангуляция



Одного теста недостаточно! Необходимо минимум два.

Слабые места TDD

- Сложно привыкнуть.
- Сложно применять TDD в ряде случаев, например, при разработке GUI.
- Требуется больше времени на разработку, т.к. необходимо писать тесты.
- Т.к. модульные тесты обычно пишутся теми же, кто написал код, то в случае неверной трактовки требований к приложению, и тест и тестируемый код могут содержать ошибки.
- И др.

Задание на дом

Скачать и установить на свой компьютер NUnit testing framework для C#

- NUnit для .NET (C#) – http://www.nunit.org/
- Быстрый старт на русском: http://goo.gl/wIQBr

По желанию:

- JUnit для Java – http://www.junit.org/
- CPPUnit для C++ - http://cppunit.sourceforge.net
- Любой другой framework.

Задание на дом

Используя методологию TDD разработать класс или набор классов, обеспечив полное покрытие тестами.

Продемонстрировать владение умение владеть инструментом NUnit, знание основ TDD.

Величины в разных шкалах.

Обеспечить (в одной шкале): умножение и деление на число, (в одной и разных шкалах): сложение, вычитание, операции отношения (больше, меньше, эквивалентность) величин, принадлежащих разным шкалам: градусы и фаренгейты, метры и футы, мили и километры, Ватты и лошадиные силы, акры и гектары, паскали (бары) и атмосферы, узлы и км/ч, литры и галлоны, унции и караты.

Пример на следующем слайде.

Пример

- 20 см. * 2 = 40 см.
- 40 см. / 5 = 8 см.
- 8 см. + 1 дюйм = 10,54 см.
- 1 дюйм + 8 см. = 4,15 дюйма.

- 1 дюйм > 2 см => true
- 8 см < 4 дюйма => true
- 2.5 см == 1 дюйм => false