



# University of Dublin Trinity College



## CS7CS3 – Test-Driven Development

Prof. Siobhán Clarke (*she/her*)

Ext. 2224 – L2.15

[www.scss.tcd.ie/Siobhan.Clarke/](http://www.scss.tcd.ie/Siobhan.Clarke/)

# What is Test-Driven Development? - 1

---

TDD focuses on what you want software to do

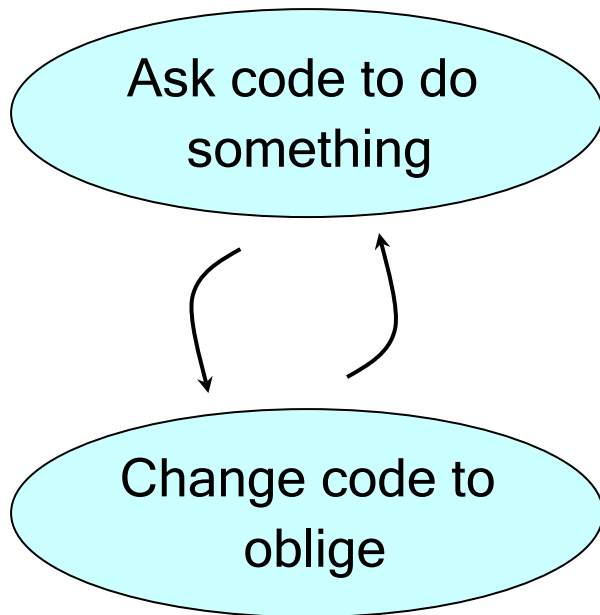
You know when your software “works” when:  
it does what you want it to do and doesn’t break!

TDD starts with thinking about how you would use the code to get it to do what you want.

- Think about what you want the code to do
- Think about how you would test it to do that

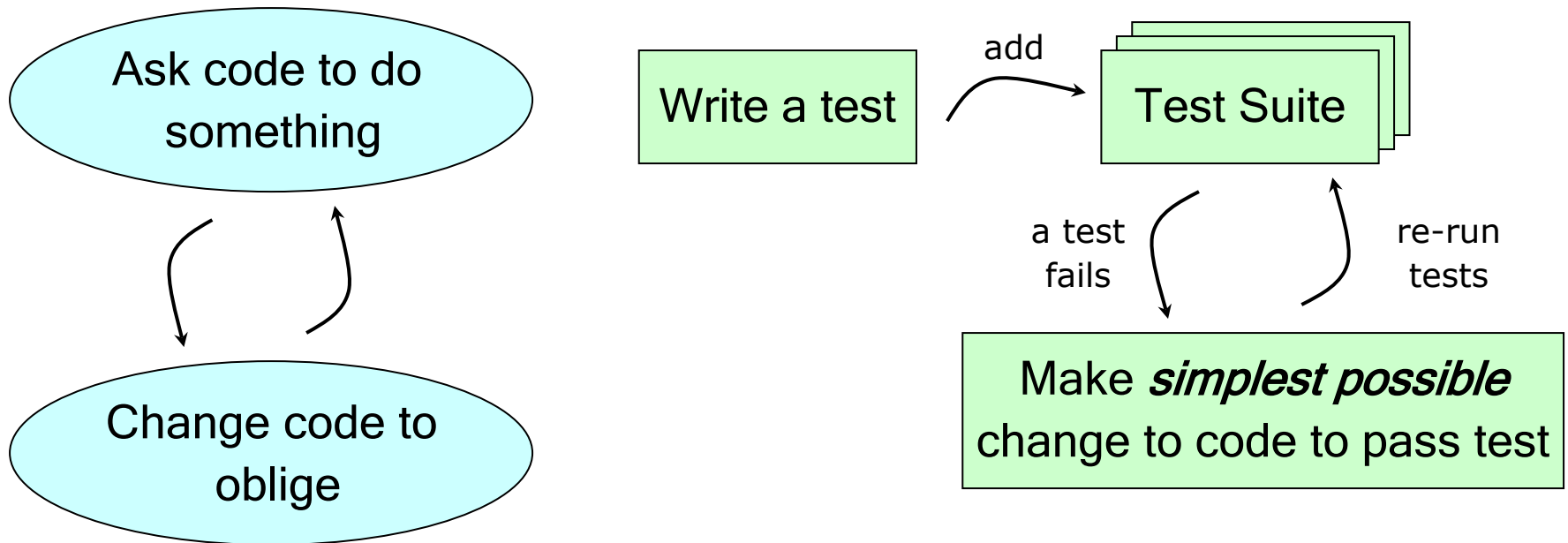
# What is Test-Driven Development? - 2

---



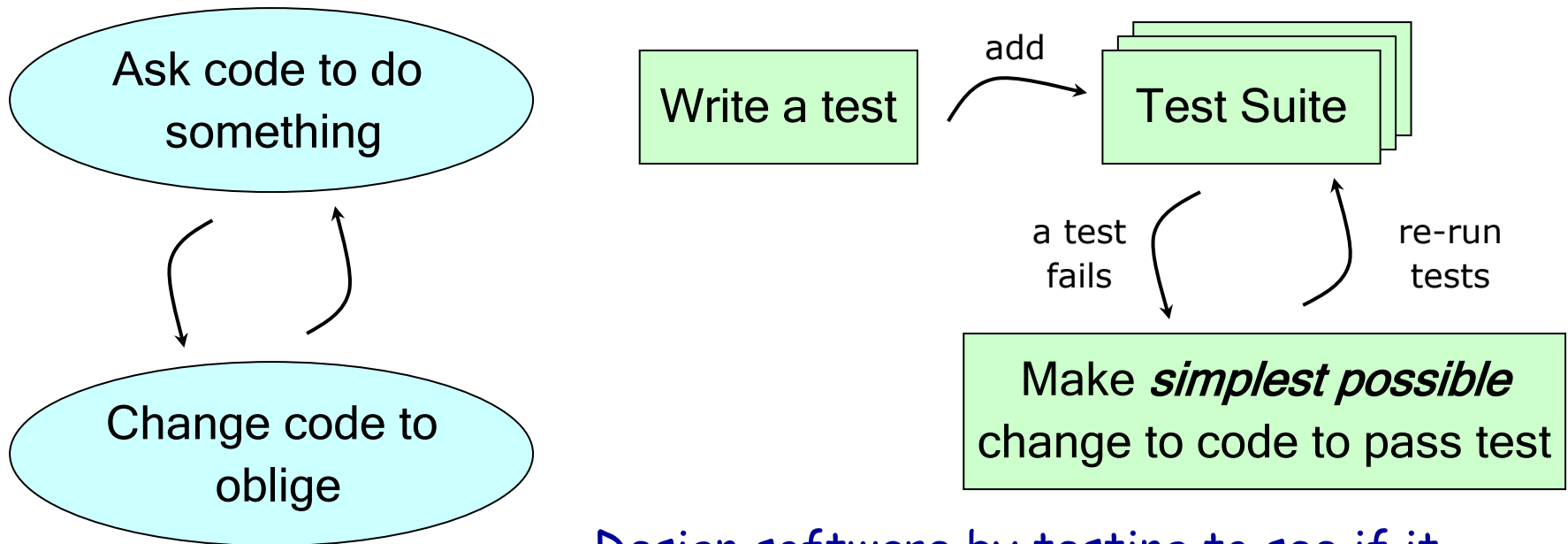
# What is Test-Driven Development? - 2

---



# What is Test-Driven Development? - 2

---

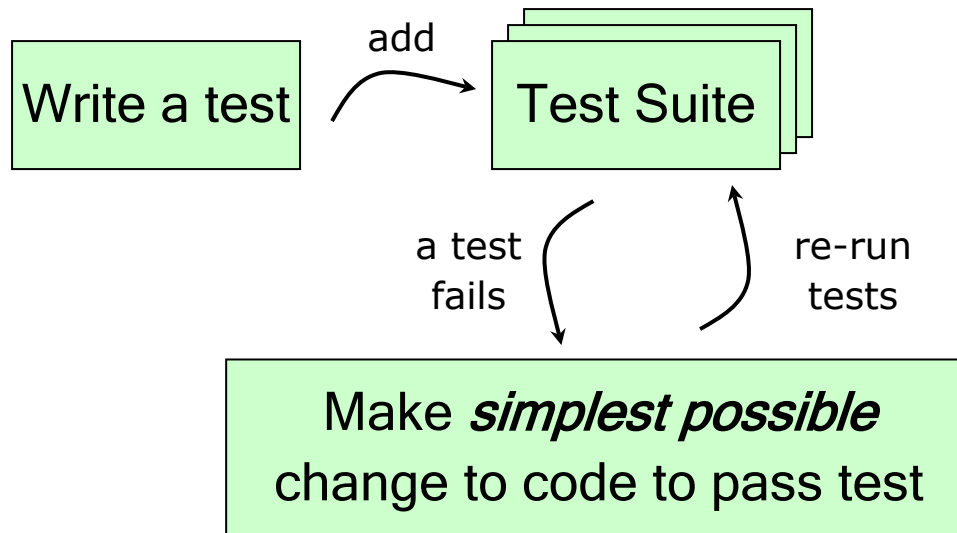


Design software by testing to see if it  
does what you want,  
small increment by small increment.

# Refactoring and Test-Driven Development

---

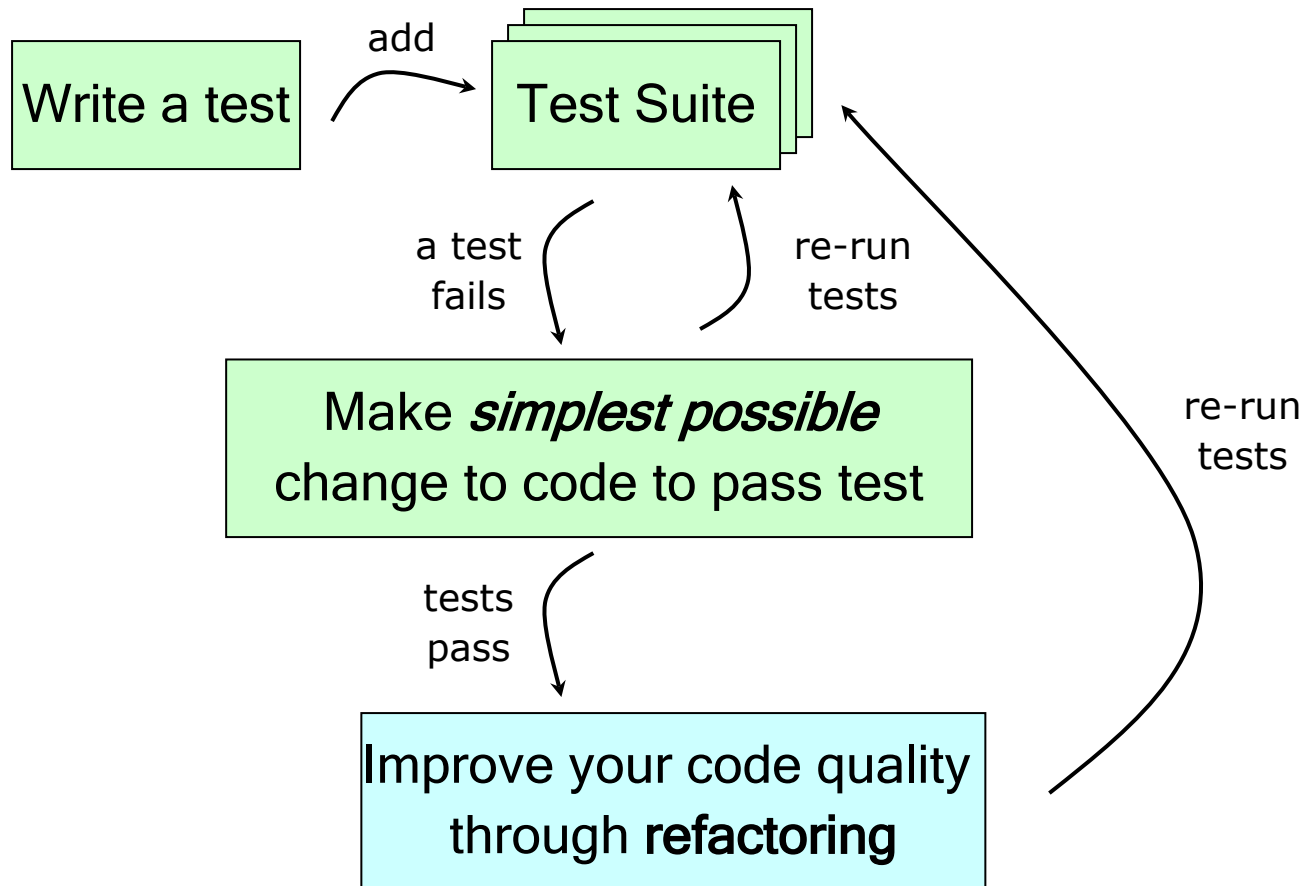
Go together, really..



# Refactoring and Test-Driven Development

---

Go together, really..



# TDD Characteristics

---

Exhaustive suite of tests is maintained

No code goes into production unless it has associated tests

Tests written FIRST (a means to design)

Tests determine what code you need to write



# How to go about writing a test

---

1. Read the application requirements
2. Pick a small piece to work on from the requirements – hereinafter called the function

# How to go about writing a test

---

1. Read the application requirements
2. Pick a small piece to work on from the requirements – hereinafter called the function
3. Think about the object(s) that should execute the function – name the corresponding Class(es).
4. Think about what variables the object(s) need to do the work for the function – name them.
5. Think about what methods in the object(s) should do the work for the function – name them.

USUAL OO  
DESIGN  
NO CODE

# How to go about writing a test

---

1. Read the application requirements
2. Pick a small piece to work on from the requirements – hereinafter called the function
3. Think about the object(s) that should execute the function – name the corresponding Class(es).
4. Think about what variables the object(s) need to do the work for the function – name them.
5. Think about what methods in the object(s) should do the work for the function – name them.
6. Start coding the test:
  - Instantiate the objects with the variables they need
  - Think of some values for the objects/variables that would test the function.  
(There are likely to be lots of different values to test boundary conditions)
  - Set the data for the tests in the variables of the objects working on the functions
  - Call the method to do the function
  - Check whether the response from the method is as expected
7. Run the test

USUAL OO  
DESIGN  
NO CODE

WRITE TEST  
CODE FIRST  
NO FUNCTION  
CODE

# How to go about writing a test

---

1. Read the application requirements
2. Pick a small piece to work on from the requirements – hereinafter called the function
3. Think about the object(s) that should execute the function – name the corresponding Class(es).
4. Think about what variables the object(s) need to do the work for the function – name them.
5. Think about what methods in the object(s) should do the work for the function – name them.

USUAL OO  
DESIGN  
NO CODE

6. Start coding the test:
  - Instantiate the objects with the variables they need
  - Think of some values for the objects/variables that would test the function.  
(There are likely to be lots of different values to test boundary conditions)
  - Set the data for the tests in the variables of the objects working on the functions
  - Call the method to do the function
  - Check whether the response from the method is as expected

7. Run the test

WRITE TEST  
CODE FIRST  
NO FUNCTION  
CODE

8. When the test fails, update the application's code (objects, variables, methods from above) to pass the test

FINALLY!

# How to go about writing a test

---

1. Read the application requirements
2. Pick a small piece to work on from the requirements – hereinafter called the function
3. Think about the object(s) that should execute the function – name the corresponding Class(es).
4. Think about what variables the object(s) need to do the work for the function – name them.
5. Think about what methods in the object(s) should do the work for the function – name them.

USUAL OO  
DESIGN  
NO CODE

6. Start coding the test:
  - Instantiate the objects with the variables they need
  - Think of some values for the objects/variables that would test the function.  
(There are likely to be lots of different values to test boundary conditions)
  - Set the data for the tests in the variables of the objects working on the functions
  - Call the method to do the function
  - Check whether the response from the method is as expected

WRITE TEST  
CODE FIRST  
NO FUNCTION  
CODE

7. Run the test

8. When the test fails, update the application's code (objects, variables, methods from above) to pass the test

FINALLY!

9. When the test passes, return to step 2, until all application requirements have been met

# Calculator example

---

1. Requirements: An application that does various calculations
2. Select initial (simple) calculations with simple tests – e.g., add two numbers, or multiply two numbers:

# Calculator example

---

1. Requirements: An application that does various calculations
2. Select initial (simple) calculations with simple tests – e.g., add two numbers, or multiply two numbers:

3-5. Thinking (i.e., designing!) -

- need an object to do the calculation – a calculator, perhaps?
- need to give this calculator two numbers and ask it to add them

# Calculator example

---

1. Requirements: An application that does various calculations
2. Select initial (simple) calculations with simple tests – e.g., add two numbers, or multiply two numbers:

3-5. Thinking (i.e., designing!) -

- need an object to do the calculation – a calculator, perhaps?
- need to give this calculator two numbers and ask it to add them

6-7. Write and run the test



# Calculator example

---

1. Requirements: An application that does various calculations
2. Select initial (simple) calculations with simple tests – e.g., add two numbers, or multiply two numbers:

3-5. Thinking (i.e., designing!) -

- need an object to do the calculation – a calculator, perhaps?
- need to give this calculator two numbers and ask it to add them

6-7. Write and run the test

8. Write the Calculator code and re-run test

*(make sure to test the calculator adding all sorts of different numbers for boundary testing)*

# Calculator example

---

1. Requirements: An application that does various calculations

2. Select initial (simple) calculations with simple tests – e.g., add two numbers, or multiply two numbers:

3-5. Thinking (i.e., designing!) -

- need an object to do the calculation – a calculator, perhaps?
- need to give this calculator two numbers and ask it to add them

6-7. Write and run the test

8. Write the Calculator code and re-run test

*(make sure to test the calculator adding all sorts of different numbers for boundary testing)*

9. Do even more complicated calculations and tests, until you have an application that does all the calculations you want

# Calculator example – first test 1


---

“need to give this calculator two numbers and ask it to add them”

# Calculator example – first test 1

---

“need to give this calculator two numbers and ask it to add them”



1. We need a Class - let's  
call it a `Calculator`

# Calculator example – first test 1

---

“need to give this calculator two numbers and ask it to add them”

1. We need a Class - let's  
call it a `Calculator`

2. We need a way to  
ask it  
to add - let's  
call that `add()`

# Calculator example – first test 1

---

“need to give this calculator two numbers and ask it to add them”

1. We need a Class - let's  
call it a `Calculator`

2. We need a way to  
ask it  
to add - let's  
call that `add()`

3. We need a way to give it  
two numbers to add -  
parameters to `add(int, int)`

# Calculator example – first test 1

---

“need to give this calculator two numbers and ask it to add them”

1. We need a Class - let's  
call it a `Calculator`

2. We need a way to  
ask it  
to add - let's  
call that `add()`

3. We need a way to give it  
two numbers to add -  
parameters to `add(int, int)`

4. Oh - and we want the answer... return type `int` on `add()`

# Calculator example – first test 2

---

Would like to be able to write the following TEST CODE

```
Calculator calc = new Calculator();  
int result = calc.add(5, 10);
```

And be confident that `result` holds 15 after `add()` called

Let's try writing test using JUnit – a testing framework integrated with many IDEs, including Eclipse

([www.junit.org](http://www.junit.org))

(JUnit written by Erich Gamma  
and Kent Beck)



# Calculator example – first test 3

---

Seems ridiculous returning 15 to pass test...

... but with complicated problems, better to build from the simplest test up to figure out exactly where a difficulty might be.

# Maintain an Exhaustive Test Suite

Characteristic 1

---

## Two Types of Test...

Unit Test: a white-box test that is concerned with a small part of implementation-oriented functionality. You need to test *all boundary conditions and error conditions*

Acceptance Test: a black-box test that assesses whether system features operate correctly or not (from an end-user point of view). Makes sure sum of small unit tests actually meet the requirements

All relevant code should have tests associated with it – the test suite is, by definition, exhaustive

# No Production without Tests

Characteristic 2

---

A feature does not exist unless it has associated tests

Everything must be tested so you have a safety net –  
the main benefit of TDD

Your safety net gives you confidence to improve your  
code quality, and integrate with others' work

# Write the Tests First

Characteristic 3

---

Decide what piece of functionality you want to implement, then write a test for it

Write a little bit of the test (e.g., test a single method), followed by just enough functionality to make the test pass, then a bit more test code, and so on...

(refactor to ensure you have the simplest code possible)

# Tests Determine what Code Written

Characteristic 4

---

You write only the code needed to pass the test, no more (YAGNI)

This means you do the simplest thing that could possibly work

Compilation errors tell you which classes and methods you need to implement to make the tests pass

# Benefits of TDD

---

Code quality is higher

You are empowered to continuously improve code quality

Developers know when they are finished implementing a piece of functionality

Integration errors and regressions are automatically caught by your test safety net

Over time, developers can program faster, with less stress

Tests act as system-level documentation

Greater customer confidence and satisfaction

# What to Test?

---

Test code when:

- You want to design the interface from the client's perspective.
- You want to check if code works.
- You want to protect code against future regressions and integrations.
- You want to explain the code. A full set of tests serve as an example of how to use the code – what happens with boundary conditions, how to properly use code, what happens in error conditions etc.

*experience will help with deciding granularity...*

---

# Test first as a design mechanism

---

- Don't forget that you're designing an object-oriented system.
- So, the code to be tested will be class-based, and will have methods.
- Everything you know about object-oriented design applies here, you're simply writing the tests first with the assumption that the classes you need exist – if they don't, the test will fail and you'll write the classes.



# A cautionary note...

---

Experience has shown that there may be difficulties using JUnit/NUnit with some kinds of system behaviours.

Investigate!

# How to go about writing a test

---

1. Read the application requirements
2. Pick a small piece to work on from the requirements – hereinafter called the function
3. Think about the object(s) that should execute the function – name the corresponding Class(es).
4. Think about what variables the object(s) need to do the work for the function – name them.
5. Think about what methods in the object(s) should do the work for the function – name them.

USUAL OO  
DESIGN  
NO CODE

6. Start coding the test:
  - Instantiate the objects with the variables they need
  - Think of some values for the objects/variables that would test the function.  
(There are likely to be lots of different values to test boundary conditions)
  - Set the data for the tests in the variables of the objects working on the functions
  - Call the method to do the function
  - Check whether the response from the method is as expected

WRITE TEST  
CODE FIRST  
NO FUNCTION  
CODE

7. Run the test

8. When the test fails, update the application's code (objects, variables, methods from above) to pass the test

FINALLY!

9. When the test passes, return to step 2, until all application requirements have been met
-

# Exercise – WRITE THE TEST FIRST

---

Write an application that can be used by a market researcher to determine whether a particular Sunday newspaper is suitable for a particular person. Each newspaper may have a fashion supplement, and/or a sports supplement, and/or an entertainment supplement, and/or a healthcare supplement. Research has found that:

- for people under 25, there should be a fashion supplement;
- for women under 40, there should be a fashion supplement;
- for men, and for women under 35, there should be a sports supplement;
- for people from urban areas, there should be an entertainment supplement;
- for people over 60, there should be a healthcare supplement.

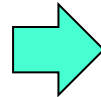
This means that a 20-year-old man from an urban area will only be interested in a Sunday newspaper that has fashion, sports and entertainment supplements. He won't, of course, care one way or another if there's a healthcare supplement.

# Steps for exercise

## Step 1

design the code (in your head or on paper! **NOT IN CODE**)

- What class should handle newspapers?
  - *What should its name be? What should its instance variables and methods be?*
- What class should handle people who might read the newspaper?
  - *What should its name be? What should its instance variables and methods be?*
- What class should do the work of the market researcher?
  - *What should its name be? What should its instance variables and methods be?*



1. Read the application requirements
2. Pick a small piece to work on from the requirements – hereinafter called the function

**USUAL OO DESIGN NO CODE**

3. Think about the object(s) that should execute the function – name the corresponding Class(es).
4. Think about what variables the object(s) need to do the work for the function – name them.
5. Think about what methods in the object(s) should do the work for the function – name them.

**WRITE TEST CODE FIRST NO FUNCTION CODE**

6. Start coding the test:
  - Instantiate the objects with the variables they need
  - Think of some values for the objects/variables that would test the function. (There are likely to be lots of different values to test boundary conditions)
  - Set the data for the tests in the variables of the objects working on the functions
  - Call the method to do the function
  - Check whether the response from the method is as expected

7. Run the test

**FINALLY!**

8. When the test fails, update the application's code (objects, variables, methods from above) to pass the test

9. When the test passes, return to step 2, until all application requirements have been met

# Steps for exercise

## Step 2

### write the test

- Instantiate the relevant objects
- Initiate values in the objects for a test
- Call the method in the market researcher to do the checking work
- Assert whether the answer is correct
- First time: will NOT be correct (no code)
- Write the code to pass the test you have set up
- Assert whether the answer is correct (repeat previous step until test passes)
- Repeat steps 1 and 2 until all parts of the exercise are complete (in step 1, the classes may already be there, just adding more checks)

1. Read the application requirements
2. Pick a small piece to work on from the requirements – hereinafter called the function

**USUAL OO DESIGN NO CODE**

3. Think about the object(s) that should execute the function – name the corresponding Class(es).
4. Think about what variables the object(s) need to do the work for the function – name them.
5. Think about what methods in the object(s) should do the work for the function – name them.

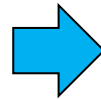
**WRITE TEST CODE FIRST NO FUNCTION CODE**

6. Start coding the test:
  - Instantiate the objects with the variables they need
  - Think of some values for the objects/variables that would test the function. (There are likely to be lots of different values to test boundary conditions)
  - Set the data for the tests in the variables of the objects working on the functions
  - Call the method to do the function
  - Check whether the response from the method is as expected
7. Run the test

**FINALLY!**

8. When the test fails, update the application's code (objects, variables, methods from above) to pass the test

9. When the test passes, return to step 2, until all application requirements have been met



# A cautionary note...

---

Experience has shown that there may be difficulties using JUnit/NUnit with some kinds of system behaviours.

Investigate!

# Test-Driven Development Works

also the view of previous MSc classes

---

## Benefits:

- Can verify whole system at any time, very easily
- Benefits of a test framework are uncountable!
- High level of confidence of quality
- Rarely felt bogged down with the exact same problem for a long time.
- Always had “previous” working version – psychologically good!
- Extremely helpful when integrating

## Drawbacks:

- Takes a while to get used to (but worth it!)
- Big learning curve
- A lot of time designing tests
- Should make tests independent of other tests, or of having specific data in the system

*...in the words of previous classes*

---

# References

---

[www.extremeprogramming.org](http://www.extremeprogramming.org)

[www.xprogramming.com](http://www.xprogramming.com)

[www.junit.org](http://www.junit.org)

[www.httpunit.org](http://www.httpunit.org)