Test Driven Development

SOLID Principles

- SRP – Single Responsibility Principle
  - Every class (or similar structure) should only have one job to do
- OCP – Open Closed Principle
  - Classes should be open for extension but closed for modification
- LSP – Liskov Substitution Principle
  - In inheritance, design your classes so that dependencies can be substituted without needing modification in the client (use interfaces)
    - If it looks like a Duck, quacks like a Duck, but needs batteries, you probable have the wrong extraction (Tractor was not a child of FarmAnimal)
- ISP – Interface Segregation Principle
  - Keep interfaces small so you don’t force classes to provide methods that have no meaning
- DIP – Dependency Inversion Principle
  - High-level modules should not depend on low-level modules, they should depend on abstractions

https://www.jrebel.com/blog/solid-principles-in-java

Objectives

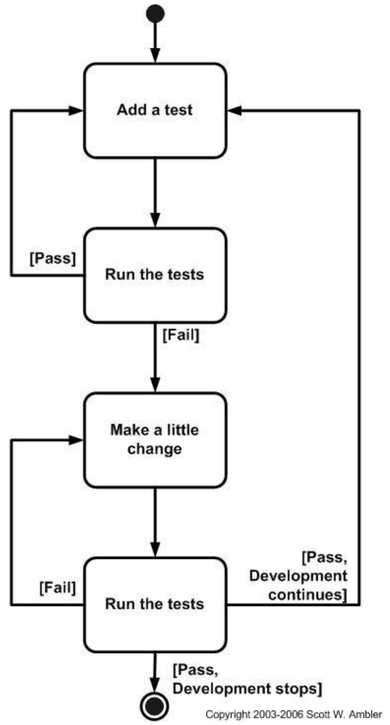
- Know what are the potential disadvantages of conducting unit testing at the end of the development cycle
- Describe the concept of Test-Driven Development (TDD) and it's benefits
- "Circle of Life" for Test Driven Development
- Explain the concept of refactoring and some simple techniques for its accomplishment
- Design a basic strategy for Test Driven Development

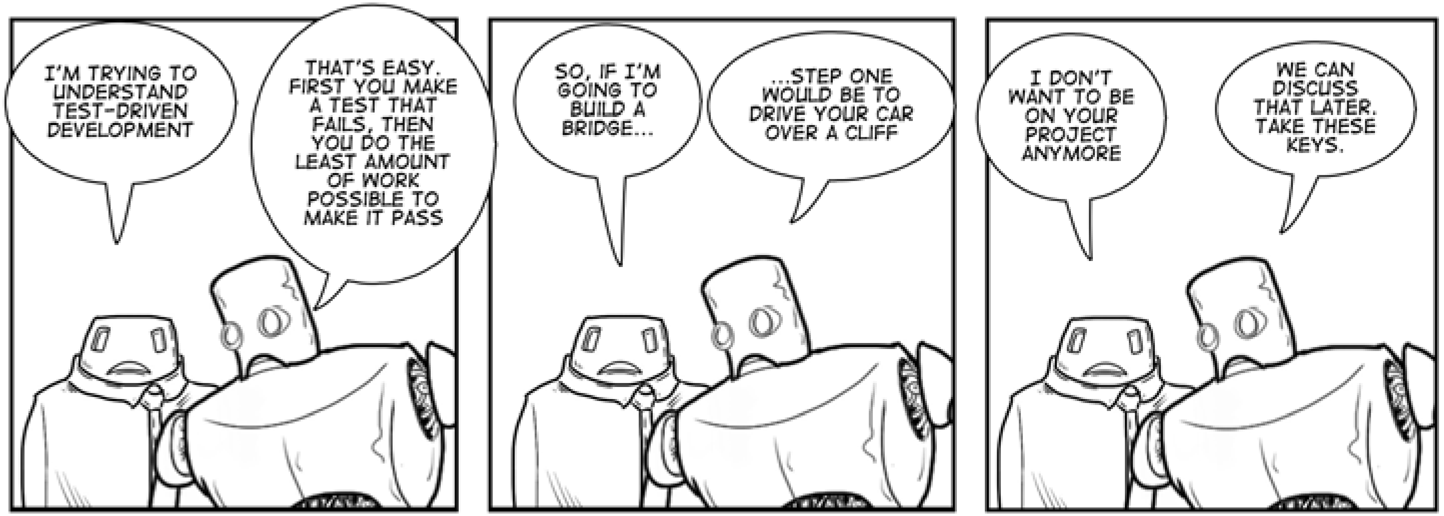
Principles of SOLID

- SRP – Single Responsibility Principle –

What is TDD?

- Test-driven development
  - Test-first development
    - Add a test
    - Add just enough code for test to pass
    - Repeat until done

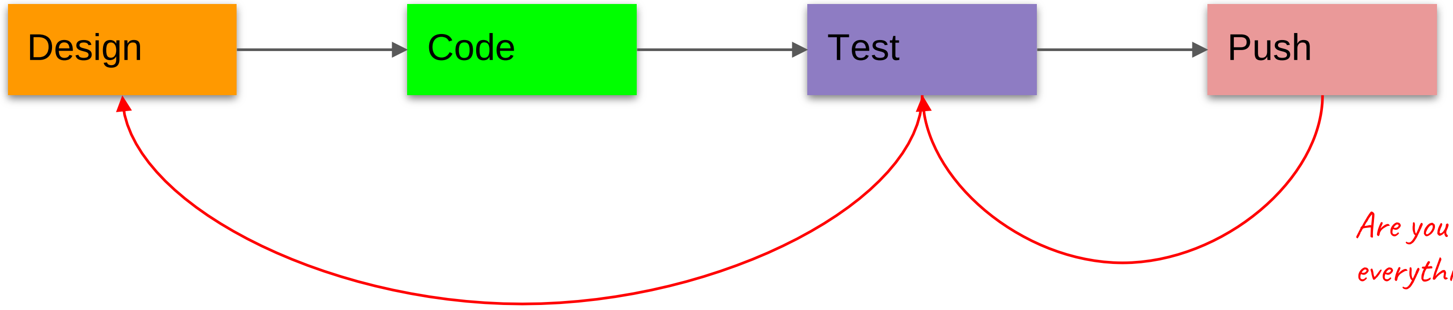

Add a test
[Pass]
Run the tests
[Fail]
Make a little change
[Fail]
Run the tests
[Pass, Development continues]
[Pass, Development stops]
Copyright 2003-2006 Scott W. Ambler


I'M TRYING TO UNDERSTAND TEST-DRIVEN DEVELOPMENT
THAT'S EASY. FIRST YOU MAKE A TEST THAT FAILS, THEN YOU DO THE LEAST AMOUNT OF WORK POSSIBLE TO MAKE IT PASS
SO, IF I'M GOING TO BUILD A BRIDGE...
...STEP ONE WOULD BE TO DRIVE YOUR CAR OVER A CLIFF
I DON'T WANT TO BE ON YOUR PROJECT ANYMORE
WE CAN DISCUSS THAT LATER. TAKE THESE KEYS.

Unit testing

- Relies on code that validates the correctness of our program using some assumptions.
  - Biased results
  - Run out of time
  - Likely to ignore or miss some of the tests

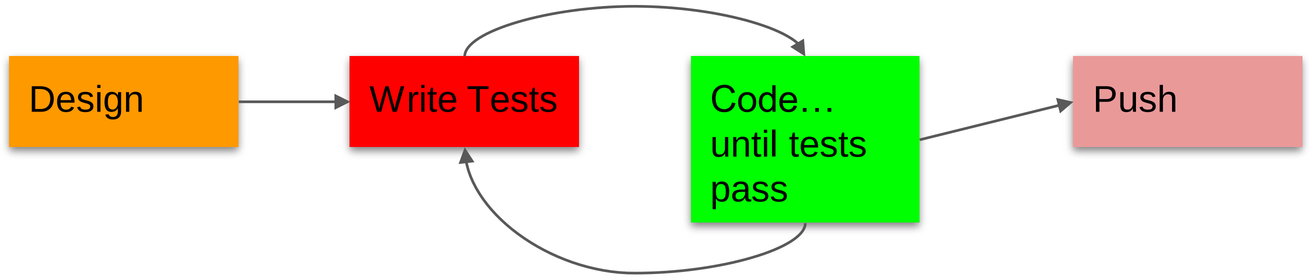
Build, then test… challenges



Are you sure you tested everything?

Now… what does the specifications document say about that requirement again?

Test Driven Development Flow



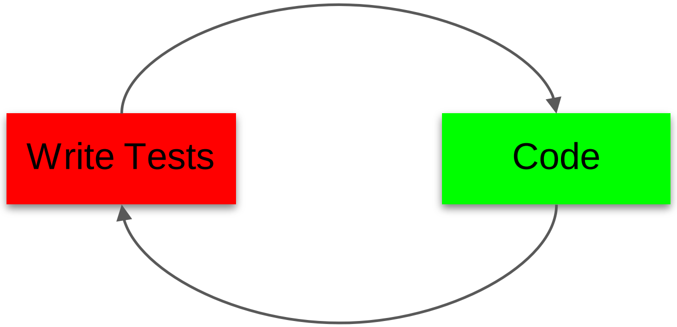
- Tests are written before actual production code, based directly on business requirements.

- Production code should not be written without an accompanying test.
    - ***Production code is driven by test code.***

- There is **working code** after every red/green cycle (more on this later).

- Over time, tests become a source of requirements and specifications.

The Red Green Cycle



- Write a Test
    - Run the test (it should fail) [Test Status: **Red**]
        - Modify your production code, cognizant of the test failure reason.
    - Re-run the test (it should now pass) [Test Status: **Green**]
    - Push the code.

- Write the next Test, start the cycle again.

- The Red Green cycle can also be used to refactor code that is not necessarily broken.

Benefits of TDD

- Program for specific conditions
- Incrementally add code
- Refactor with new patterns
- Previous tests hold value

Required tools for TDD Success

- A real IDE! (for Java: Eclipse, IntelliJ, NetBeans)
- Good mastery of your testing Framework of choice (for Java: Junit)
- Practice

Refactoring

- Modifications to code
  - To improve structure or design
  - Will not change functionality

How to Refactor

- Eliminate duplicate code
- Break down long methods into smaller methods
- Use variables for complex operations
- Use constants for magic numbers
- Simplify conditional expresssions

General Strategy for TDD

1. Create a list of tests needed.
2. Start writing just enough test code
3. Run test to see code fail
4. Write enough production code to make test build
5. Write enough code to make test pass (even if fake it)
6. If obvious, write implementation code. Otherwise repeat 4 and 5 until obvious
7. Generalize code by refactoring