Fundamentals of High Discipline Test Driven Development

Learning Objectives

- Be able to justify TDD
- Understand the fundamentals of TDD
- Use TDD to write maintainable and self documenting code
- Understand how TDD aids in software design
- Learn how to build a system from the top, downwards
- Use TDD to grow the design of a system organically
- Use tests for fast feedback
- Be exposed to complimentary XP practices

Why TDD?

- Guard against regression
- Enable fearless refactoring
- Executable documentation
- Short feedback loops
- Avoid scope creep – identify when work is complete
- Reduced use of the debugger
- Facilitates team members to work on code simultaneously

TDD Encourages Simple Code

- Small classes focused on one thing
- Test small increments of code
- Encourages:
  - Low coupling
    - Tightly coupled systems are difficult to evolve
    - A change in one area often triggers undesired changes in a different area
  - High cohesion
    - Placing related concepts and components near each other
    - Aids in the readability of the design
    - Reduces error because it aids in readability

Preparing for TDD

- Cyclic dependency rules:
  - Test Code:
    - must be able to depend on production code
    - must be able to depend on testing libraries
  - Production Code:
    - must not be able to reference test code
    - must not be able to reference testing libraries
  - IDE and build tools can enforce these rules

Test Directory Structure

- Separate test and production code
  - Make it difficult to deploy test code
  - Ensures running all tests is easy
- One production class per file

Example Directory Structures

Ruby

```
.
├── lib
│   └── <project_name>
│       └── calculations
│           └── shortest_path.rb
├── spec
│   └── <project_name>
│       └── calculations
│           └── shortest_path_spec.rb
└── test
    └── <project_name>
        └── calculations
            └── shortest_path_test.rb
```

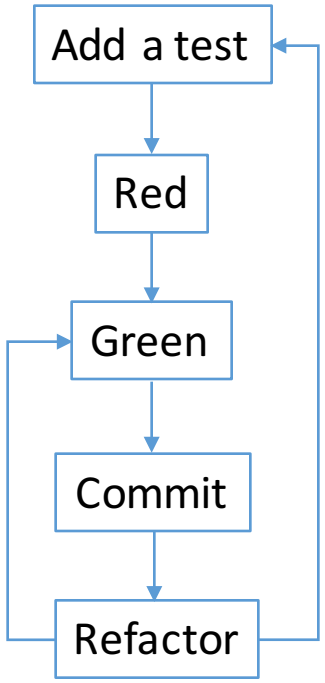
Apache Maven Standard for Java

```
.
└── src
    ├── main
    │   └── java
    │       └── com
    │           └── stride
    │               └── calculations
    │                   └── ShortestPath.java
    └── test
        └── java
            └── com
                └── stride
                    └── calculations
                        └── ShortestPathTest.java
```

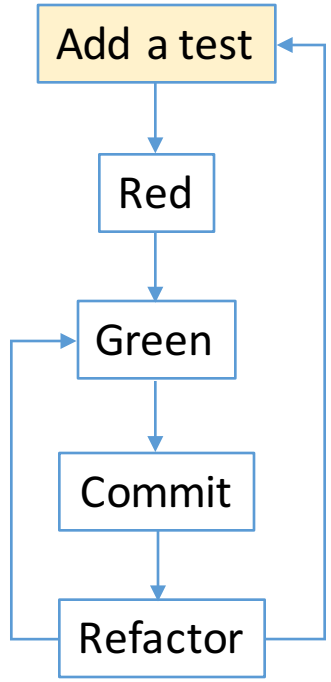
Task the functionality out

- On index cards create a list of tasks needed to complete the exercise
- Tasks to consider:
  - Boundary cases:
    - Null inputs
    - Empty strings
    - Invalid strings
    - Values above or below a certain amount (e.g. is -50 miles per hour valid)
  - Tasks required for completion
- Need not be exhaustive yet
- Create new tasks as work continues

Test Driven Development Cycle



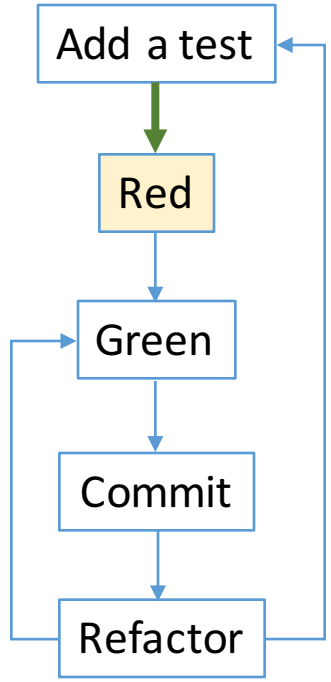
Add a test



- From your list of tasks choose the simplest task
- Write the simplest, tiniest, test possible
- Test naming
  - Name tests from perspective of business functionality
  - Avoid restating in words the body of the test
  - Prefer: “Require An Umbrella When There Is Rain” instead of: “Return True When Rain Is”
  - Make a note that test names should be refactored too
- Prior to running the test, make the code runnable:
  - Static languages: ensure the tests compile
  - Dynamic languages: you can likely just run the test

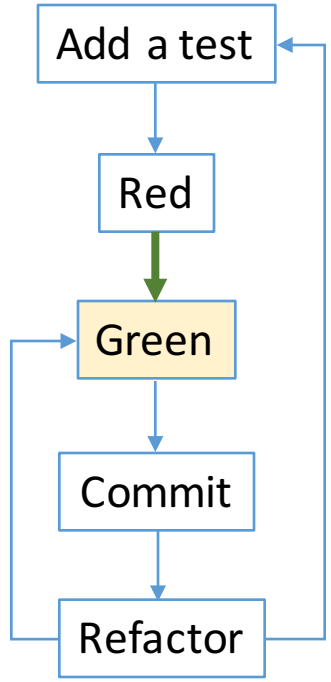
Red

- Run the test, ensure it fails
- At this stage a failing test is good
- Ensures there exists a test that can be made to pass



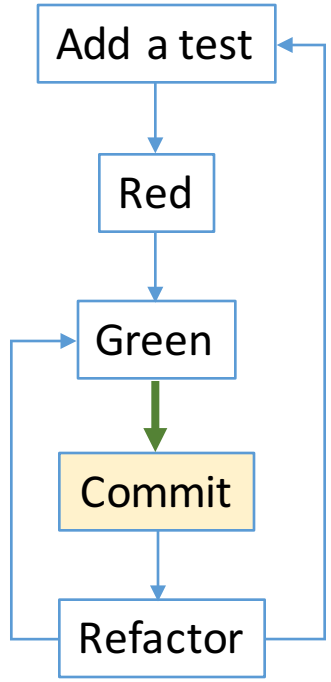
Green

- Write the obvious implementation
- Add only enough code to solve problem at hand
- Implementation does not need to be production quality
- If possible, consider a hard coded value



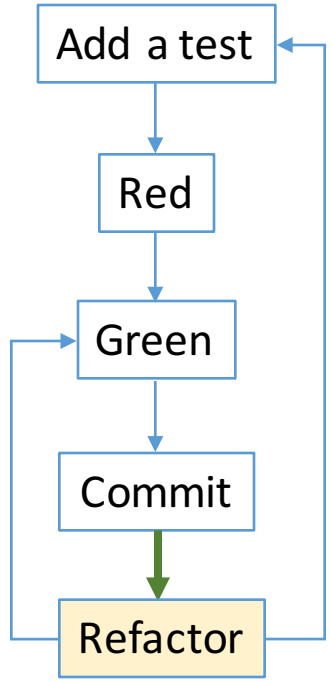
Commit

- Frequent, small commits reduce lost effort when new functionality does not work and code needs reverting
- Review the diff of all work before committing
- Commit all files, not doing so runs the risk of build/test failures
- Check the passing test into version control
- Destroy task if code is production quality – unlikely at this stage



Refactor

- "…is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior." – refactoring.com
- We will look at:
  - Data duplication
  - Code duplication, later



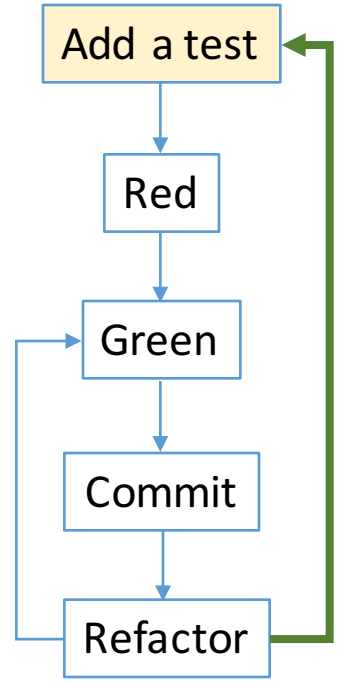
YAGNI

- You Aint Gonna Need It - YAGNI
- Add only enough code to make the new test pass
- Avoid Big Design Upfront
  - Allow your tasks to guide what is needed
- Don’t add features in anticipation of what may come
  - …for it might not

New Tasks

- Reduce context switching and multi-tasking
- Capture new ideas on index cards as tasks when they come to mind
- Avoid distracting yourself with new tasks during a task
  - If you must take on a distraction, allow only a single distraction

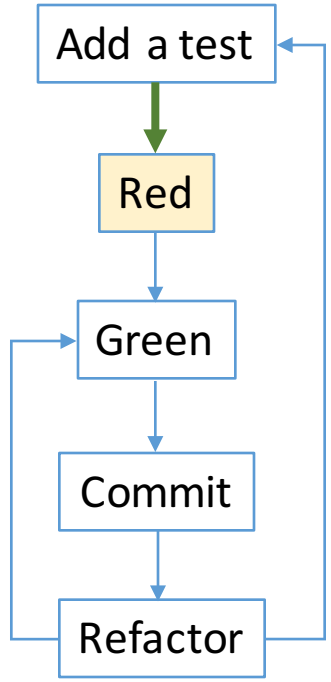
Add a test - Triangulation



- Assuming your previous test was hard coded: triangulate
- First documented by Kent Beck
- Drawn from Radar Triangulation
  - Definition: http://encyclopedia2.thefreedictionary.com/radar+triangulation
- Triangulation works as follows:
  1. Start with obvious implementation
  2. Follow up with a general implementation
- Pushes original implementation toward production quality code
- Potentially monotonous, useful for ambiguous implementations

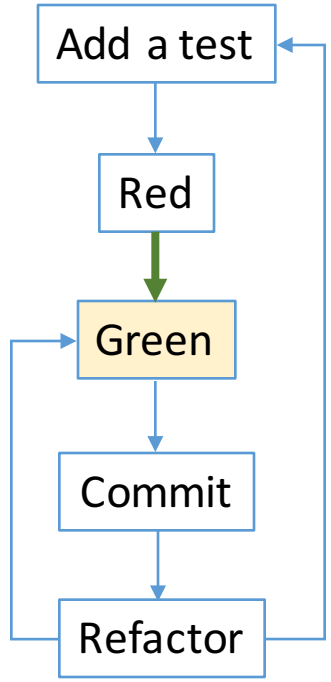
Red

- Run the test, ensuring it fails
- If it doesn’t fail, change it so it does
- If you’re still unable to: identify what you’re trying to test
- Devise a new test that does fail
- Test is to result in a small increment of new functionality



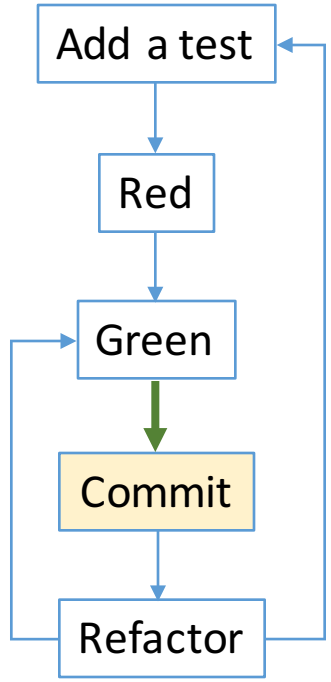
Green

- Make a small change to pass the failing test
- If triangulating, you'll likely remove previous hard coding



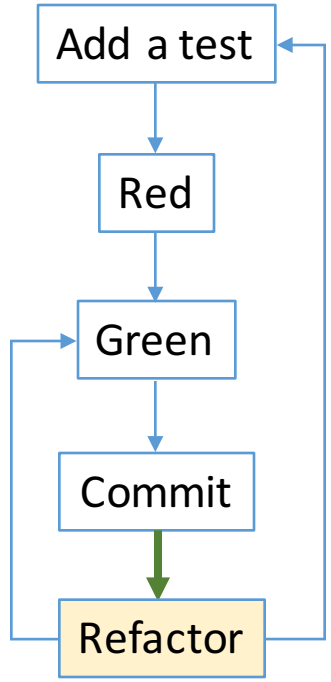
Commit

- Small commits are the goal
- Destroy the current task if code is production ready



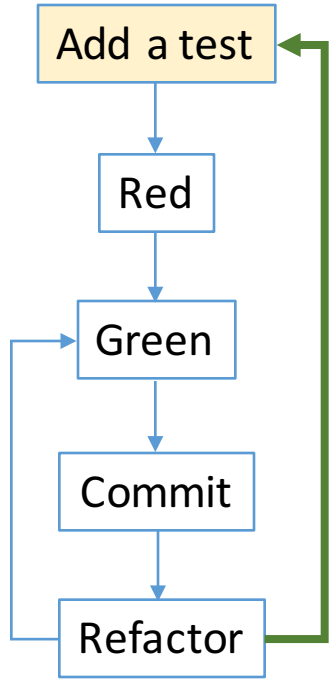
Refactor

- As a system evolves, test semantics may change, such tests require renaming
- At this point there is likely to be little code to refactor



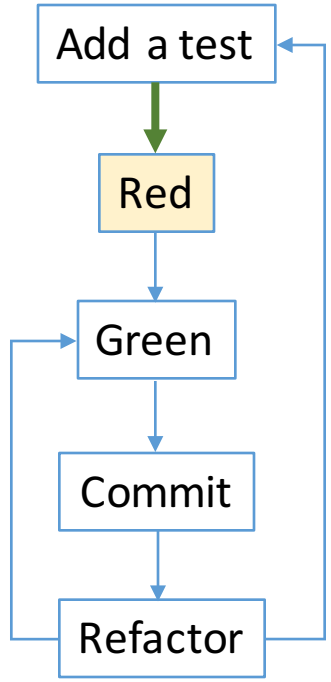
Add a test

- Find the next simple task
- Write the simplest failing test you can think of
- Make it compilable or runnable



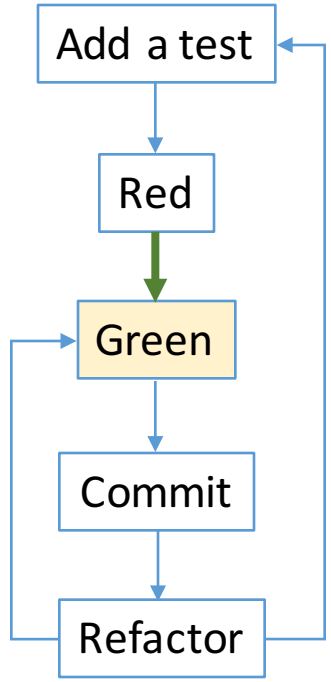
Red

- Ensure the test fails



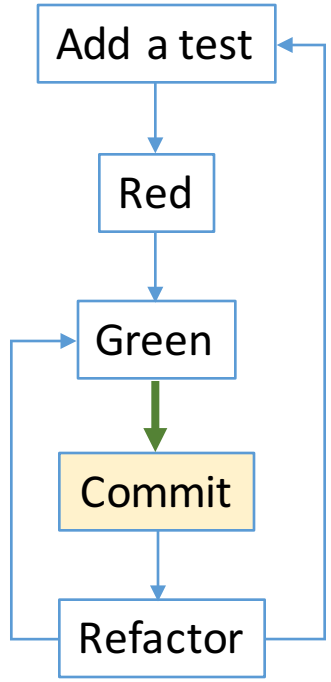
Green

- Make the test pass



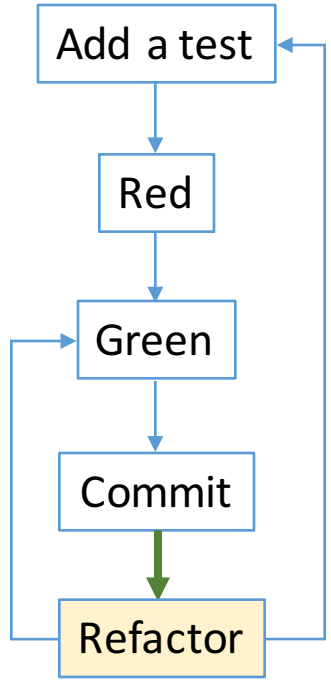
Commit

- Time to create the baseline
- Destroy the task card if the task is complete



Refactor

- Don’t tolerate duplicate code
- Two types of duplication:
  - Code duplication
  - Data duplication



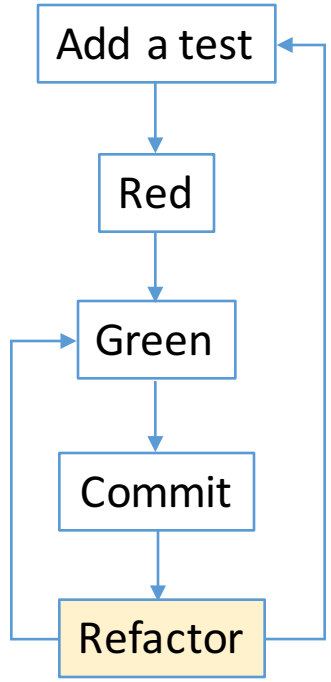
Refactor – Code Duplication

- Duplication of code patterns
- e.g.

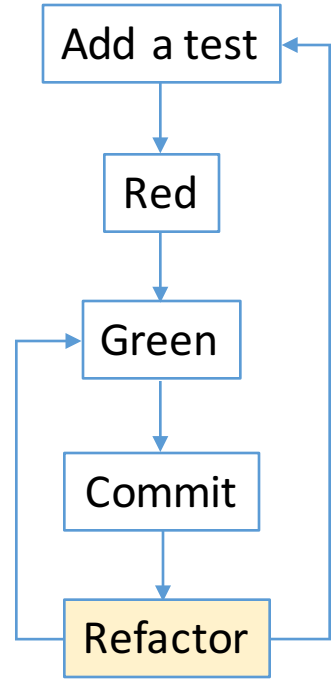
```
public void run() {
    dress();
    energy -= 100;
    shower();
}

public void walk() {
    dress();
    energy -= 10;
    shower();
}
```

```
public void activity(int energySpent) {
    dress();
    energy -= energySpent;
    shower();
}
```



Refactor – Data duplication



- Commonly found between a test and production code
- Helpful in determining the next test to write
- e.g.

```
public void testShouldHaveAFullTankOfFuel() {
    assertEquals(100, new Car().fuelLevel());
}

public class Car {
    public int fuelLevel() {
        return 100;
    }
}
```

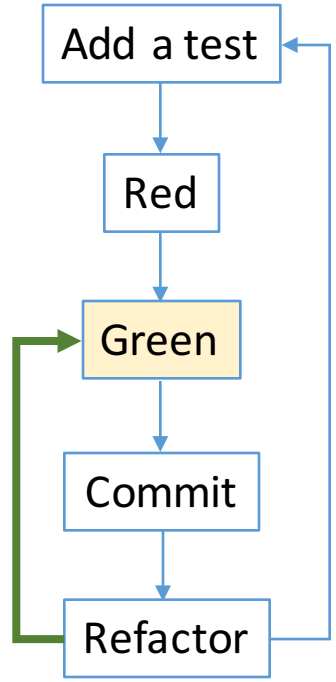
```
public void testShouldHaveAFullTankOfFuel() {
    final int fuelLevel = 100;
    assertEquals(fuelLevel, new Car(fuelLevel).fuelLevel());
}

public class Car {
    private int _fuelLevelPercentage;
    public Car(int fuelLevel) {
        _fuelLevelPercentage = fuelLevel;
    }

    public int fuelLevel() {
        return _fuelLevelPercentage;
    }
}
```

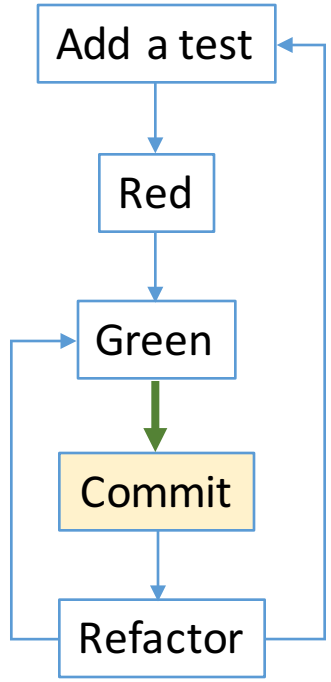
Green

- Ensure all tests still pass, after having performed a refactoring
- No matter how small the refactoring, ensure all tests pass

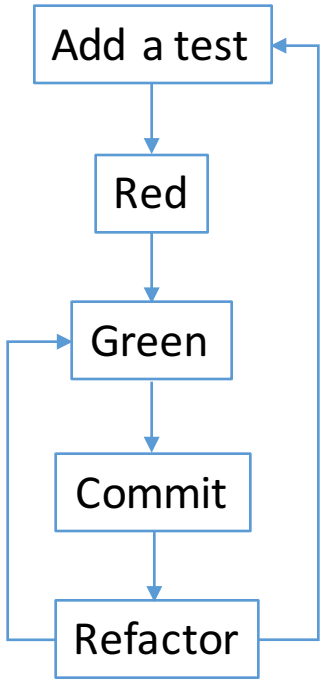


Commit

- Commit, so as to lock the refactoring in
- Destroy the card of the task you are presently working on



Test Driven Development Cycle



Object Mother