What is TDD

WHAT IS TDD

- Test-Driven Development (or test driven design) is a methodology.

Common TDD misconception:

- TDD is not about testing
- TDD is about design and development
- By testing first you design your code

WHAT IS TDD

- Short development iterations.
- Based on requirement and pre-written test cases.
- Produces code necessary to pass that iteration's test.
- Refactor both code and tests.
- The goal is to produce working clean code that fulfills requirements.

TEST-DRIVEN DEVELOPMENT

- Test-driven development (TDD) is a software development technique that uses short development iterations based on pre-written test cases that define desired improvements or new functions. Each iteration produces code necessary to pass that iteration's tests. Finally, the programmer or team refactors the code to accommodate changes. A key TDD concept is that preparing tests before coding facilitates rapid feedback changes. Note that test-driven development is a software design method, not merely a method of testing.

Principle of TDD

Kent Beck defines:

- Never write a single line of code unless you have a failing automated test.
- Eliminate duplication.

Red (Automated test fail)

Green (Automated test pass because dev code has been written)

Refactor (Eliminate duplication, Clean the code)

HOW DOES TDD HELP

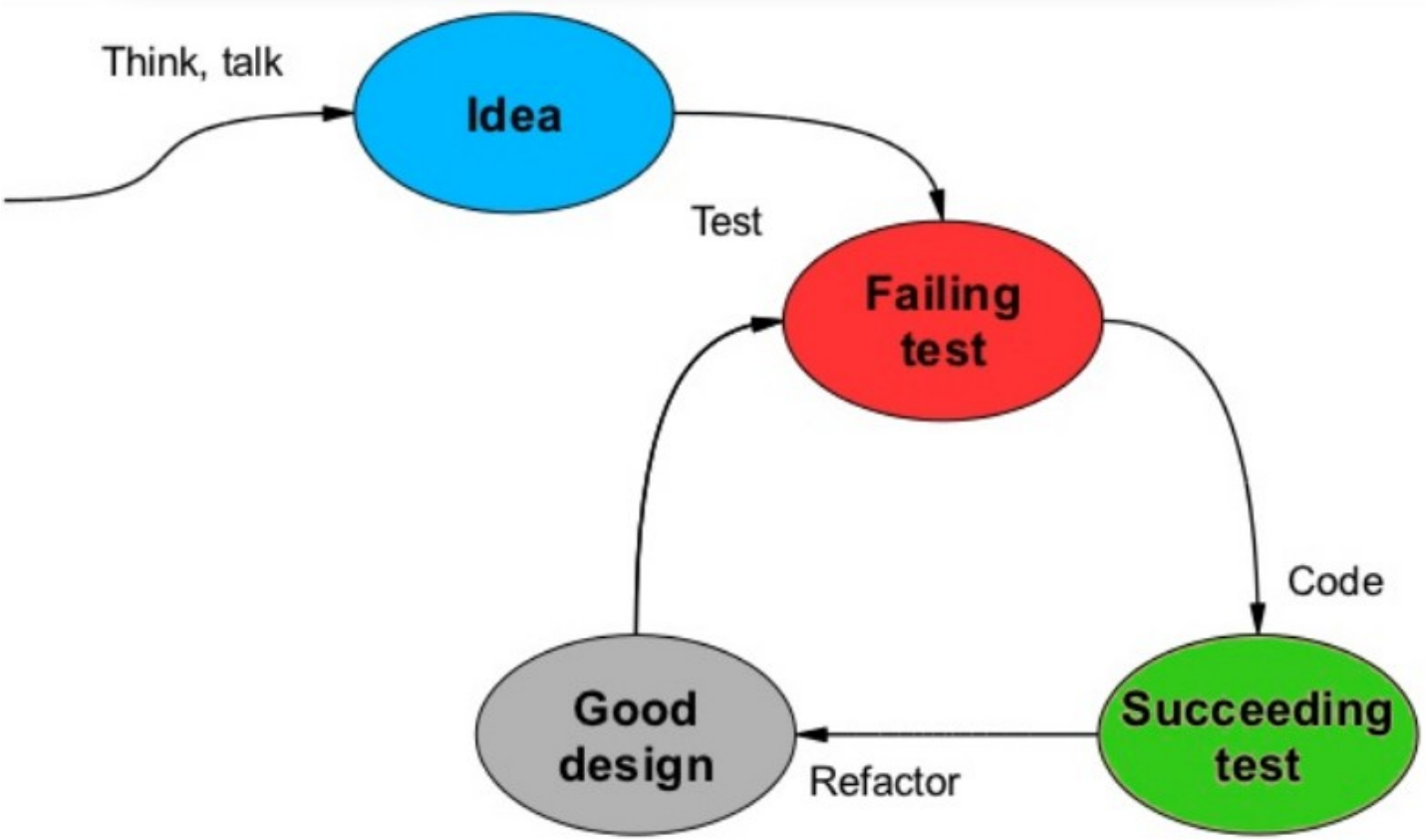
- TDD helps you produce clean working code that fulfills requirements

- Write Test Code
  - Code that fulfills requirements
- Write Functional Code
  - Working code that fulfills requirements
- Refactor
  - Clean working code that fulfills requirements

TDD BASICS - UNIT TESTING

- Red, Green, Refactor
- Make it Fail
  - No code without a failing test
- Make it Work
  - As simply as possible
- Make it Better
  - Refactor

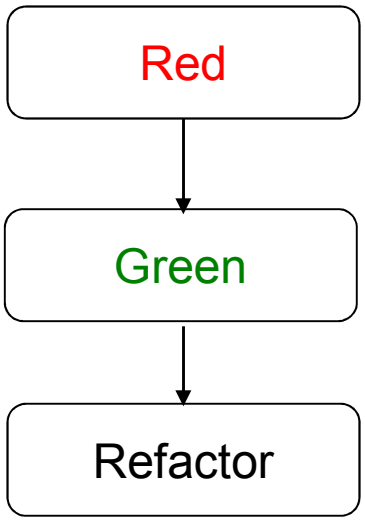
HOW DOES TDD HELP



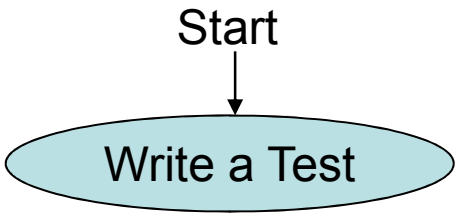
TDD CYCLE

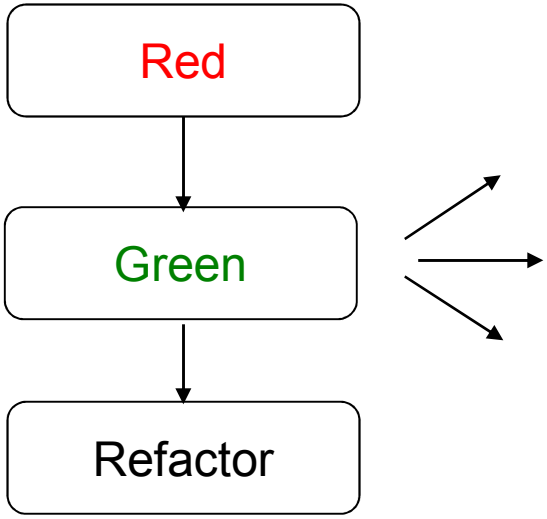
- Write Test Code
  - Guarantees that every functional code is testable
  - Provides a specification for the functional code
  - Helps to think about design
  - Ensure the functional code is tangible
- Write Functional Code
  - Fulfill the requirement (test code)
  - Write the simplest solution that works
  - Leave Improvements for a later step
  - The code written is only designed to pass the test
    - no further (and therefore untested code is not created).
- Refactor
  - Clean-up the code (test and functional)
  - Make sure the code expresses intent
  - Remove code smells
  - Re-think the design
  - Delete unnecessary code

Principle of TDD (In Practice)

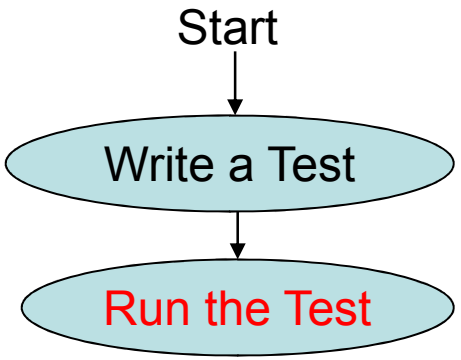


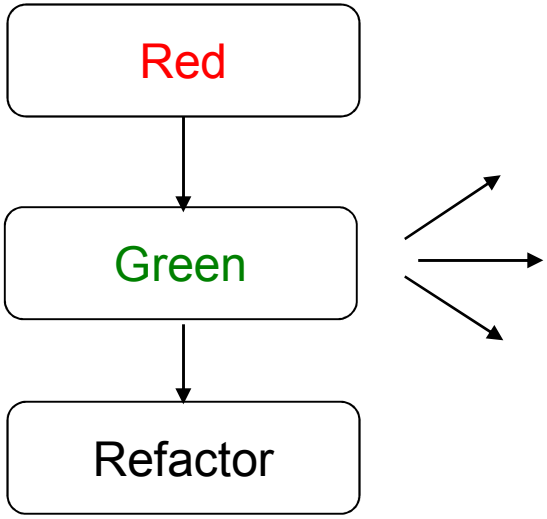
Principle of TDD (In Practice)



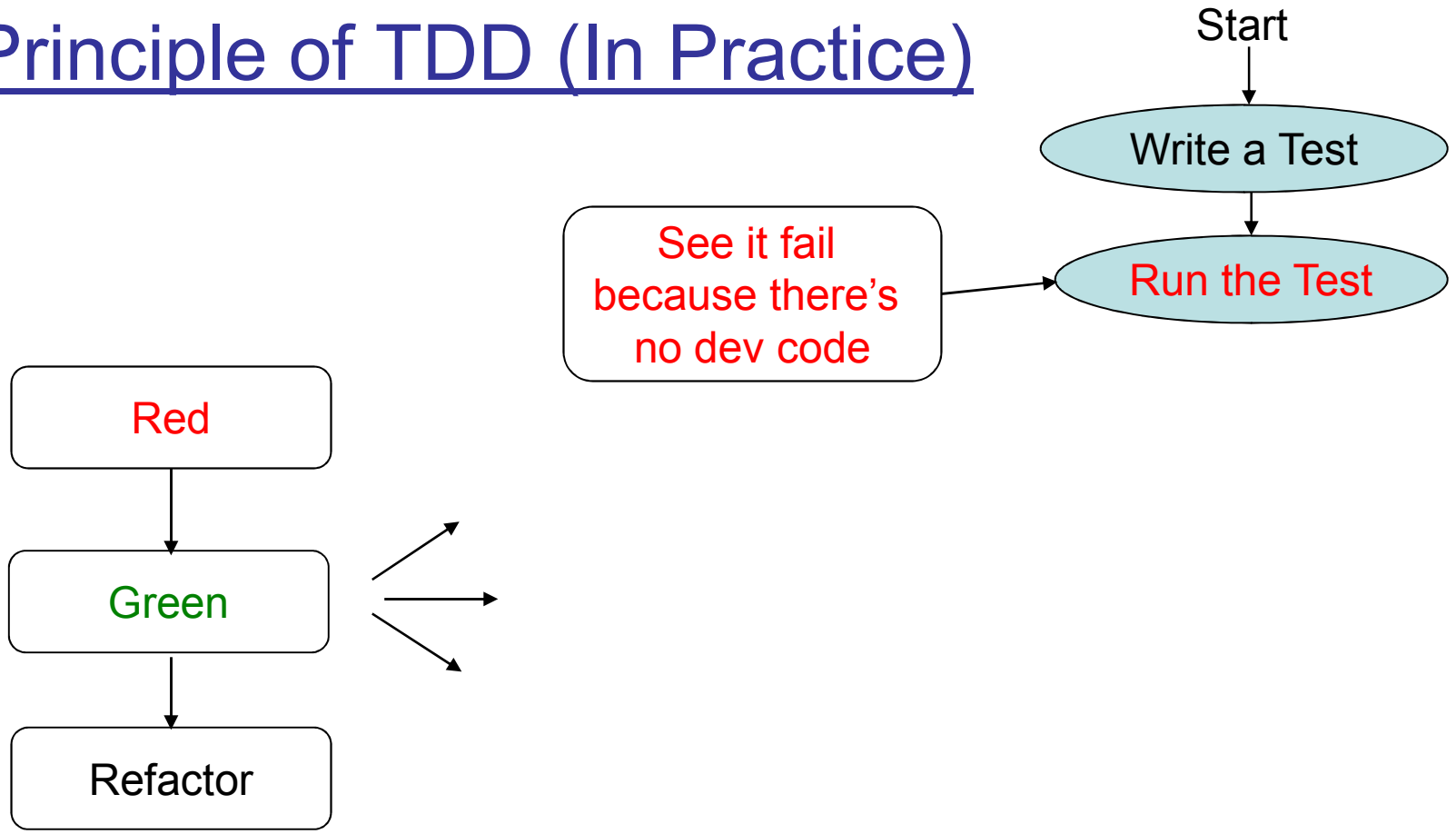


Principle of TDD (In Practice)

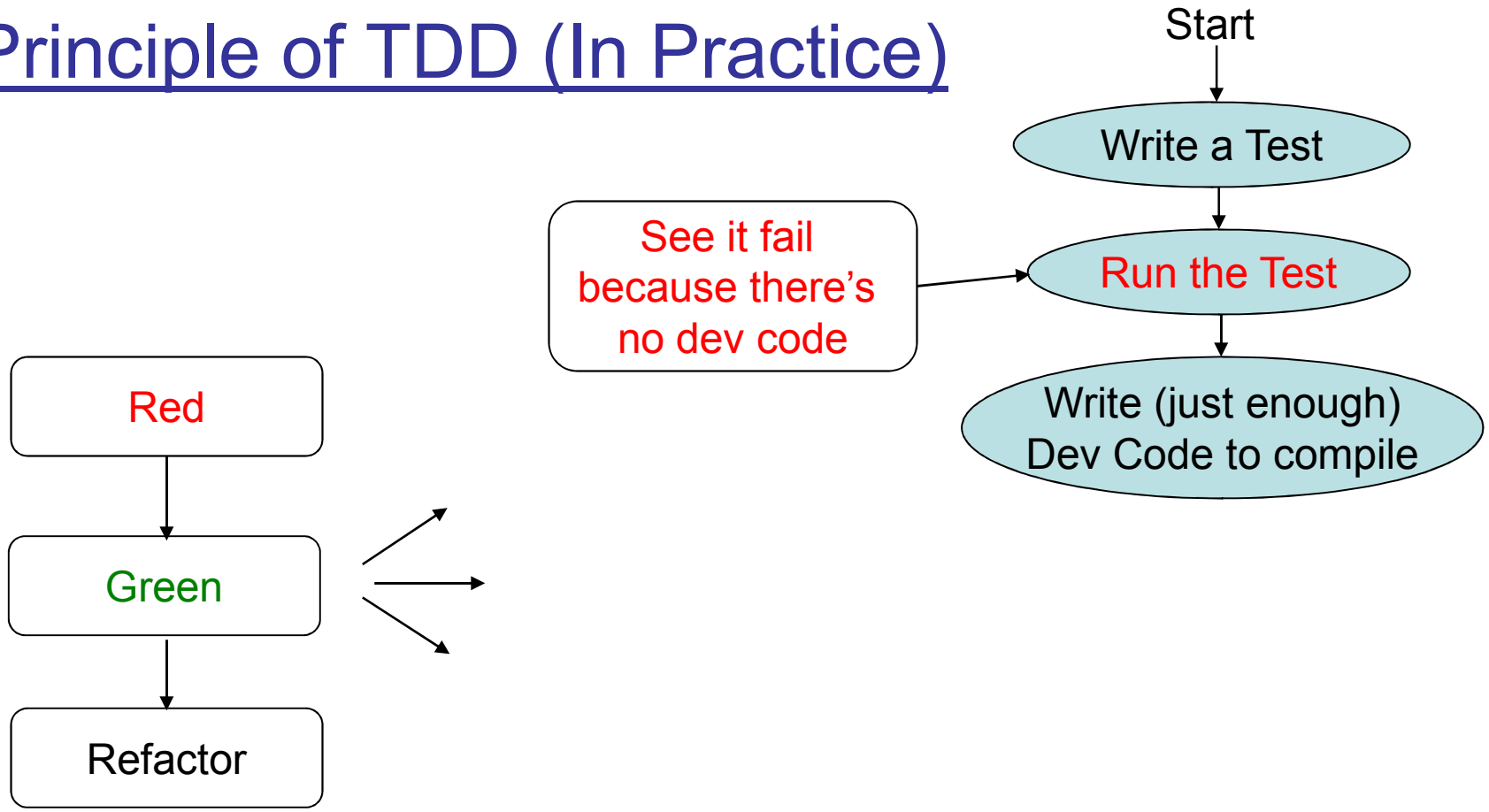




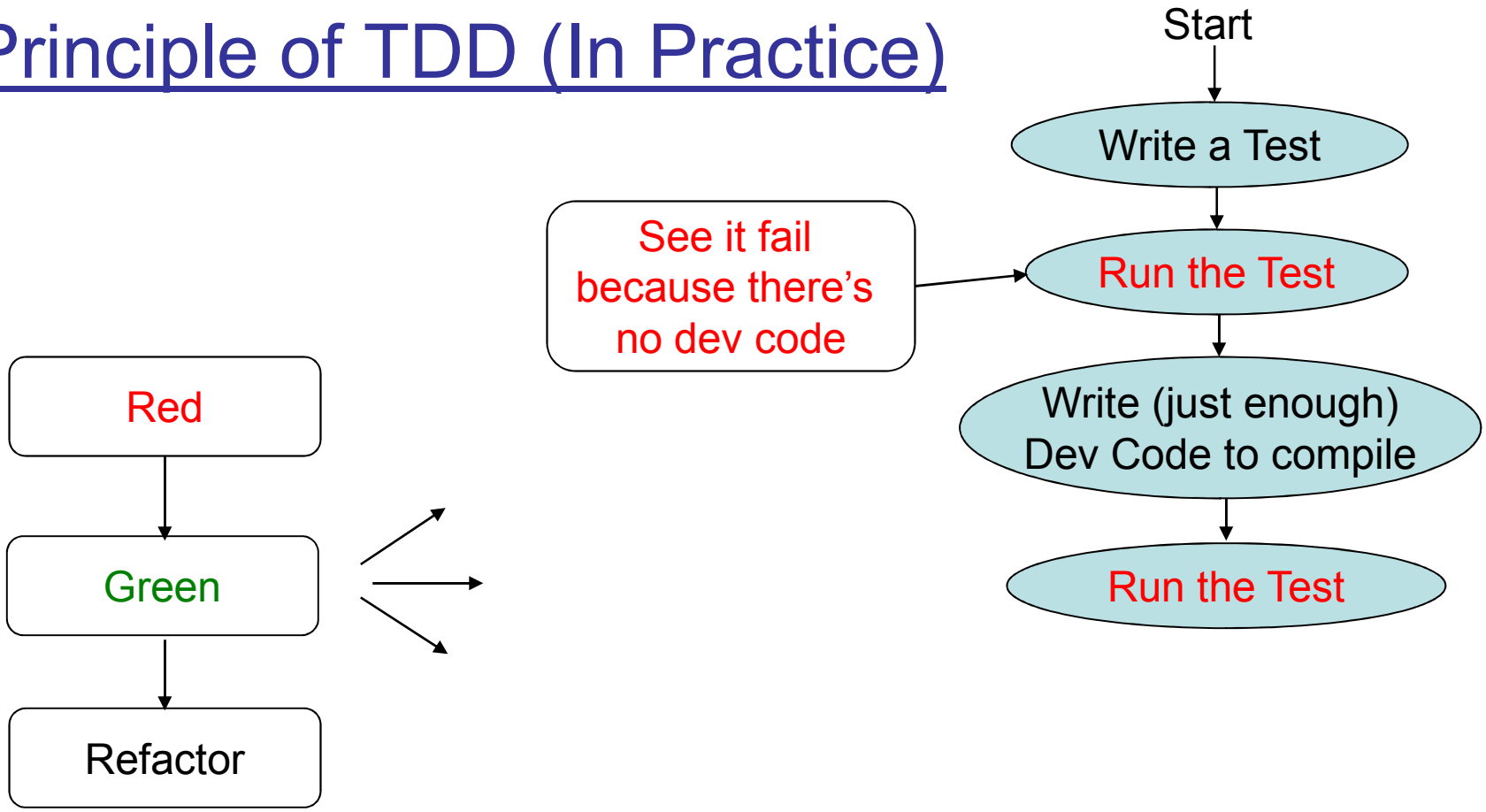
Principle of TDD (In Practice)



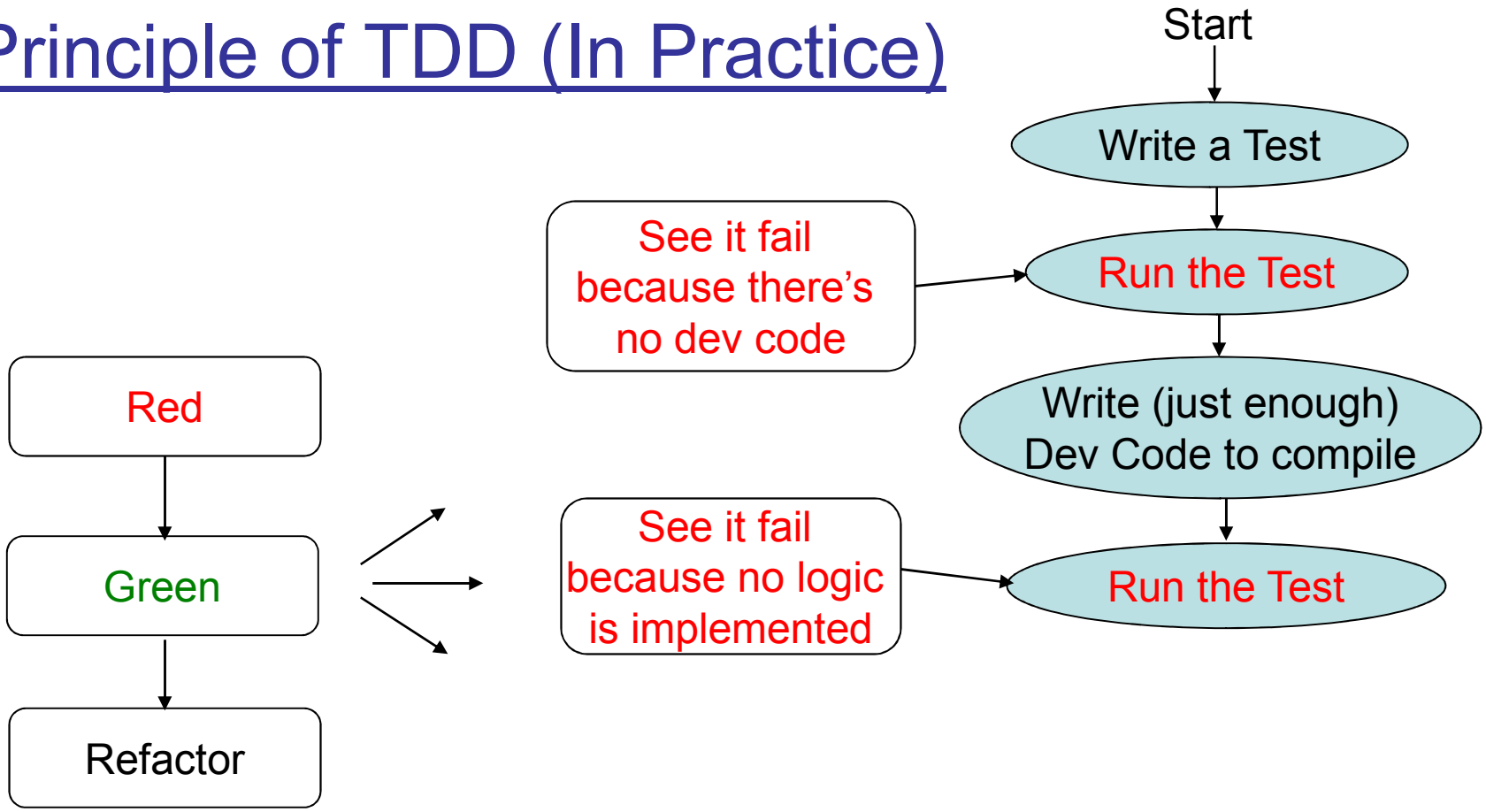
Principle of TDD (In Practice)



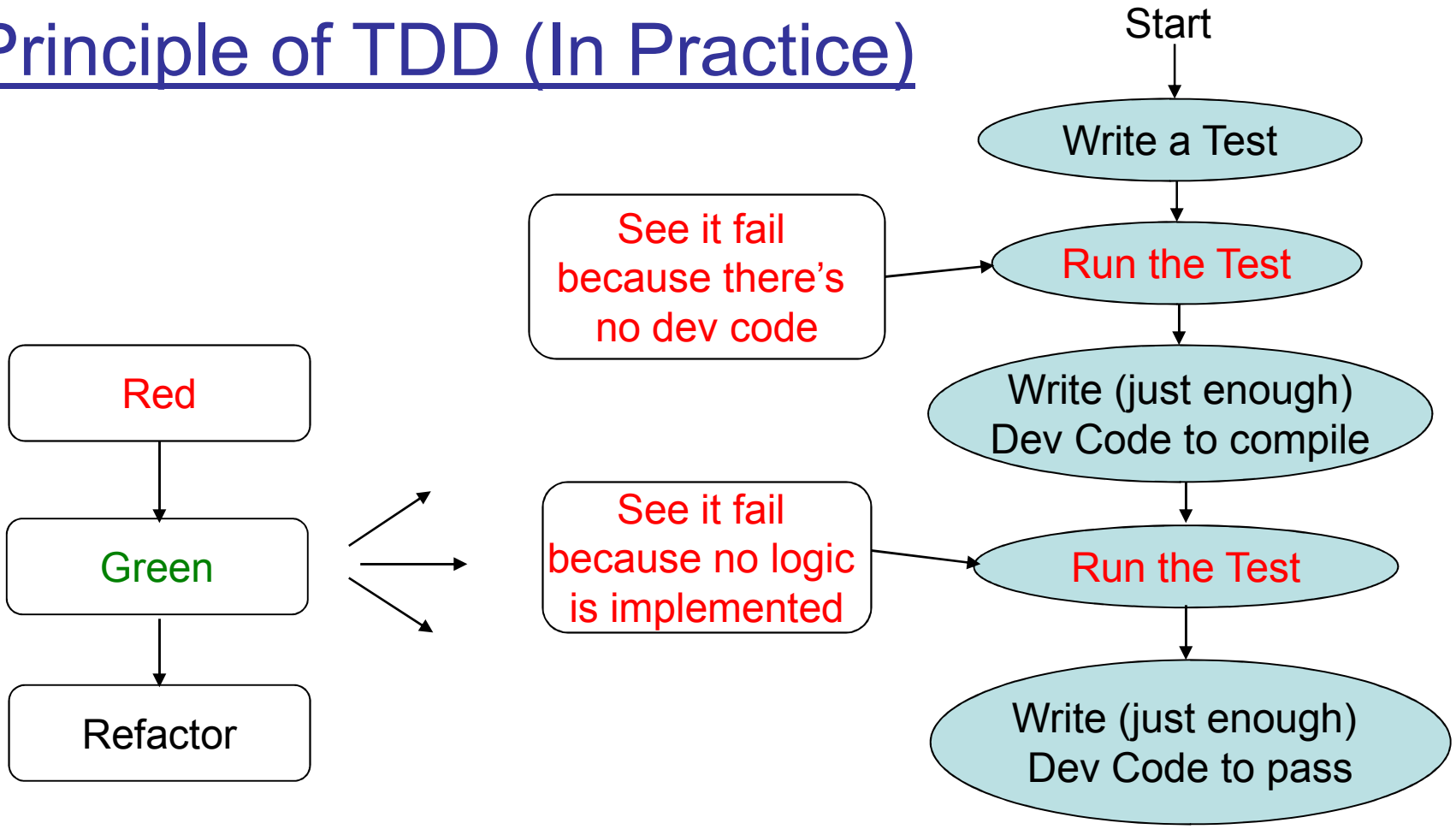
Principle of TDD (In Practice)



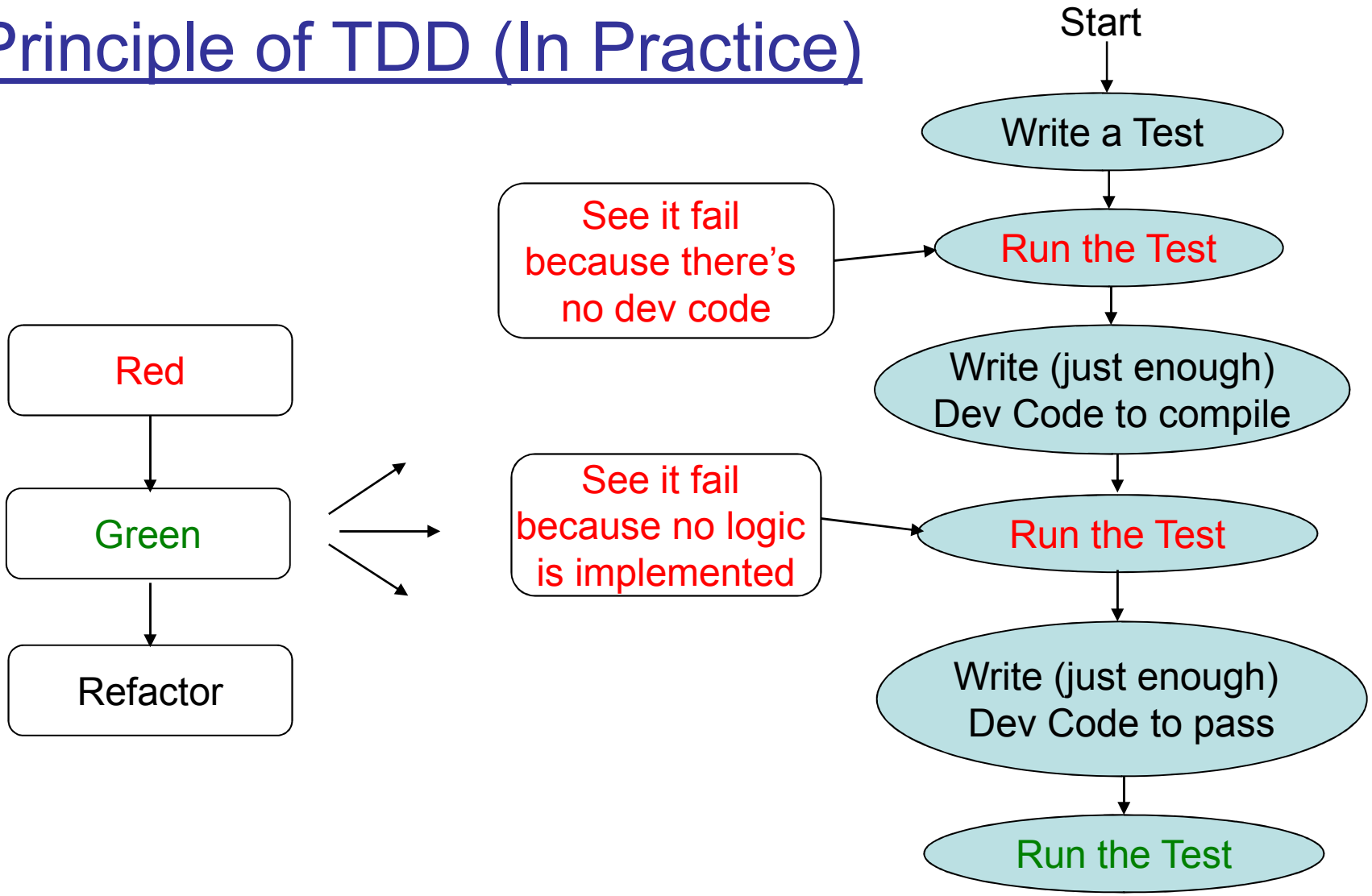
Principle of TDD (In Practice)



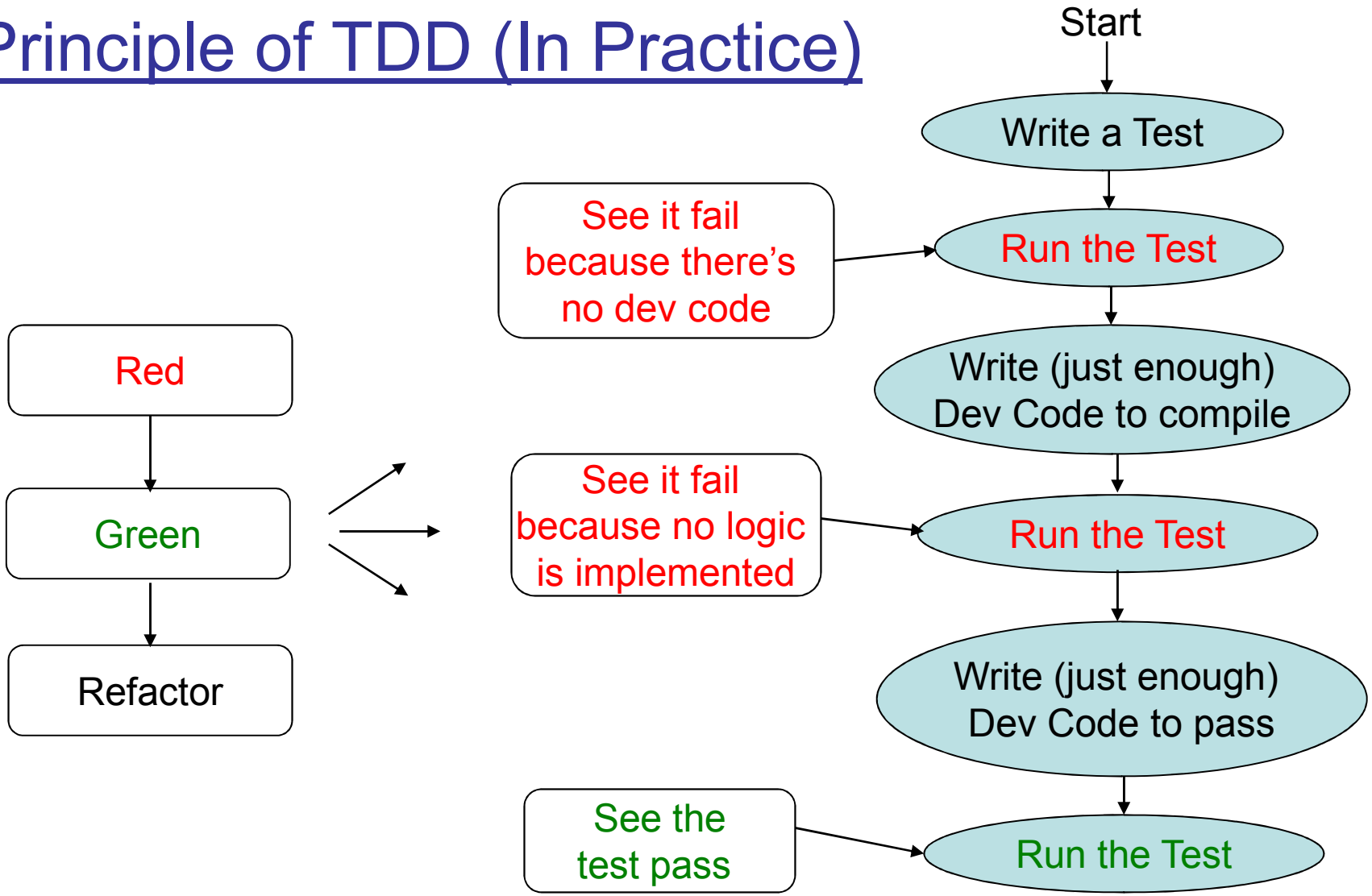
Principle of TDD (In Practice)



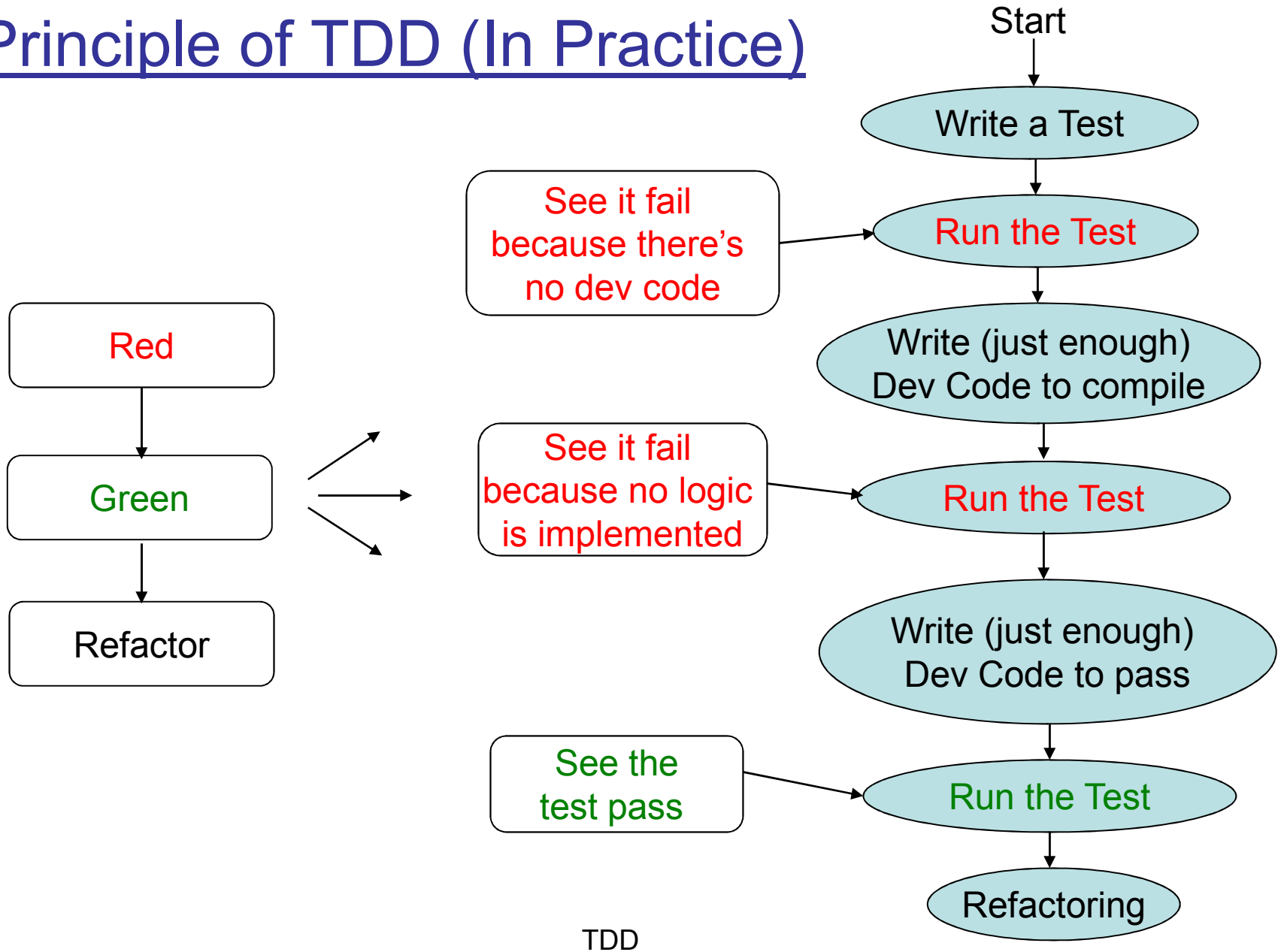
Principle of TDD (In Practice)



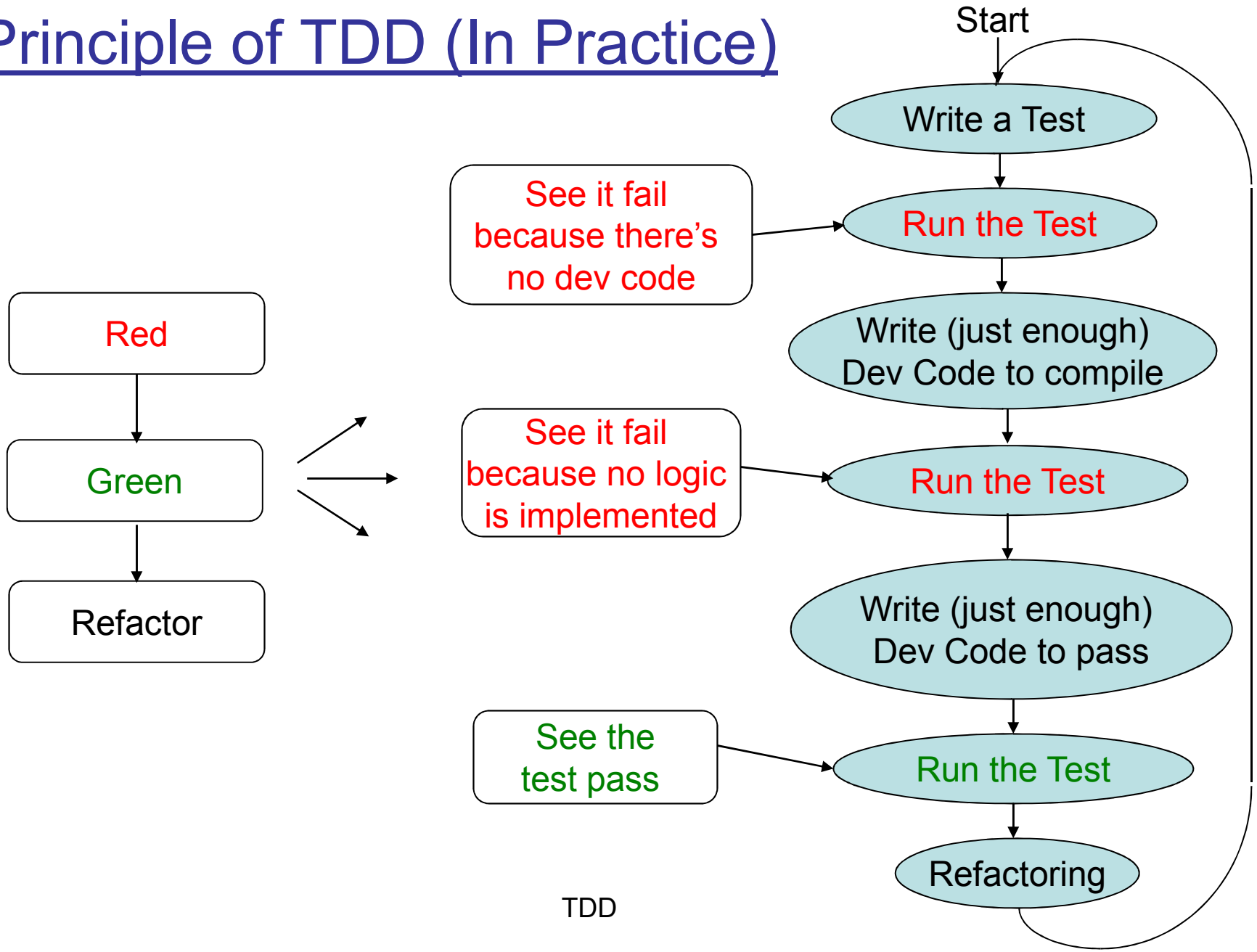
Principle of TDD (In Practice)



Principle of TDD (In Practice)



Principle of TDD (In Practice)

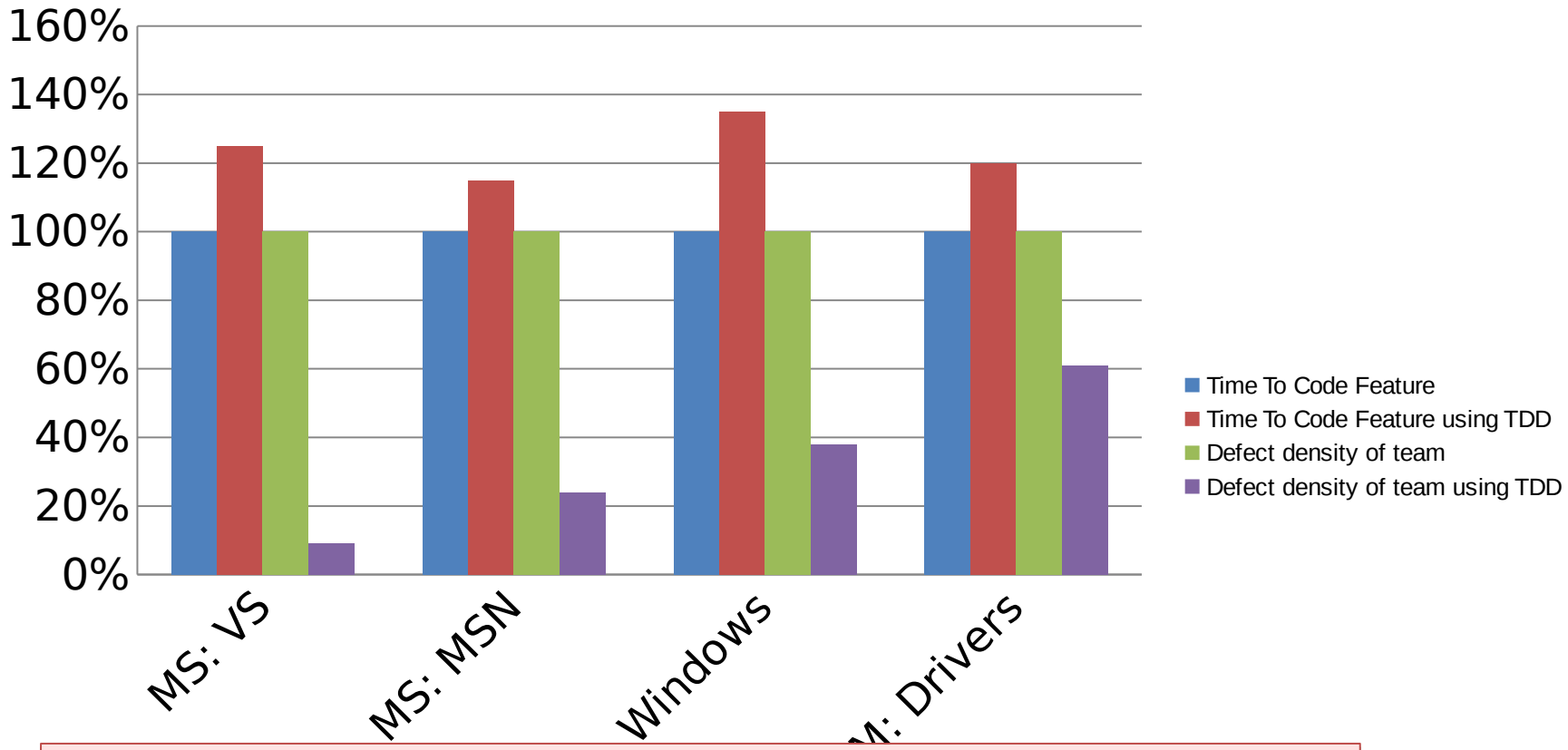


Why TDD

WHY / BENEFITS

- Confidence in change
  - Increase confidence in code
  - Fearlessly change your code
- Document requirements
- Discover usability issues early
- Regression testing = Stable software = Quality software

IS TDD A WASTE OF TIME (MICROSOFT RESEARCH)



Major quality improvement for minor time investment

ADVANTAGES OF TDD

- TDD shortens the programming feedback loop
- TDD promotes the development of high-quality code
- User requirements more easily understood
- Reduced interface misunderstandings
- TDD provides concrete evidence that your software works
- Reduced software defect rates
- Better Code
- Less Debug Time.

DISADVANTAGES OF TDD

- Programmers like to code, not to test
- Test writing is time consuming
- Test completeness is difficult to judge
- TDD may not always work

How to

EXAMPLE

- We want to develop a method that, given two Integers, returns an Integer that is the sum of parameters.

EXAMPLE (CONT.)

- Test

```
Integer i =
  new Integer(5);
Integer j =
  new Interger(2);
Object o = sum(i,j);
```

- Method

EXAMPLE (CONT.)

- Test

```
Integer i =
  new Integer(5);
Integer j =
  new Interger(2);
Object o = sum(i,j);
```

- Method

```
public static Object
  sum(Integer i,
      Integer j) {
  return new  Object();
}
```

EXAMPLE (CONT.)

- Test

```
Integer i =
  new Integer(5);
Integer j =
  new Interger(2);
Object o = sum(i,j);
if (o instanceof
    Integer)
  return true;
else
  return false;
```

- Method

```
public static Object
  sum(Integer i,
      Integer j) {
  return new  Object();
}
```

EXAMPLE (CONT.)

- Test

```
Integer i =
  new Integer(5);
Integer j =
  new Interger(2);
Object o = sum(i,j);
if (o instanceof
    Integer)
  return true;
else
  return false;
```

- Method

```
public static Integer
  sum(Integer i,
     Integer j) {
  return new
  Integer();
}
```

EXAMPLE (CONT.)

- Test

```
Integer i =
  new Integer(5);
Integer j =
  new Interger(2);
Object o = sum(i,j);
if ((o instanceof
      Integer) &&
     ((new Integer(7))
      .equals(o))
  return true;
else
  return false;
```

- Method

```
public static Integer
  sum(Integer i,
      Integer j) {
  return new
  Integer();
}
```

EXAMPLE (CONT.)

- Test

```
Integer i =
  new Integer(5);
Integer j =
  new Interger(2);
Object o = sum(i,j);
if ((o instanceof
     Integer) &&
    ((new Integer(7))
     .equals(o))
  return true;
else
  return false;
```

- Method

```
public static Integer
  sum(Integer i,
      Integer j) {
  return new  Integer(
       i.intValue() +
        j.intValue());
}
```

REMEMBER - THERE ARE OTHER KINDS OF TESTS

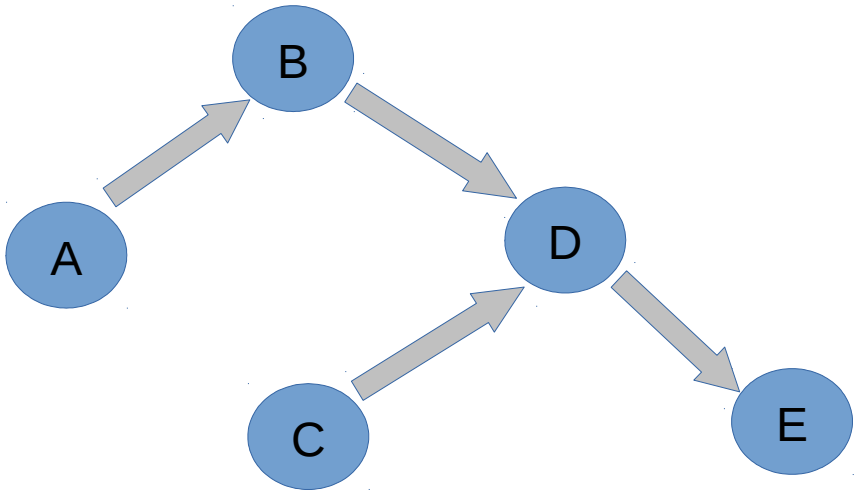
- Unit test (Unit)
- Integration test (Collaboration)
- User interface test (Frontend)
- Regression test (Continuous Integration)
- …, System, Performance, Stress, Usability, …

The only tests relevant to TDD is Black-box Unit Testing

- White-box test

HOW TO DO TDD

- … on my component A?
- Unit Test A, but what about B, C, D…?



HOW TO DO TDD

- … on my component A?

This, and the previous slide,
are all thoughts on implementing tests on existing code.

This is all White-box Unit- or Integration Testing and has therefore nothing to do with TDD.

In TDD you write a Black-box Unit test that fails, first,
and worry about the code implementation later.

SINGLE MOST IMPORTANT THING
WHEN LEARNING TDD

Do not
write the code in your head
before you write the test

SINGLE MOST IMPORTANT THING WHEN LEARNING TDD

- When you first start at doing TDD you "know" what the design should be. You "know" what code you want to write. So you write a test that will let you write that bit of code.
- When you do this you aren't really doing TDD – since you are writing the code first (even if the code is only in your head)
- It takes some time to (and some poking by clever folk) to realize that you need to focus on the test. Write the test for the behavior you want - then write the minimal code needed to make it pass - then let the design emerge through refactoring. Repeat until done.

Where to begin

UNBOUNDED STACK EXAMPLE

- Requirement: FILO / LIFO messaging system
- Brainstorm a list of tests for the requirement:
  - Create a *Stack* and verify that *IsEmpty* is true.
  - *Push* a single object on the *Stack* and verify that *IsEmpty* is false.
  - *Push* a single object, *Pop* the object, and verify that *IsEmpty* is true.
  - *Push* a single object, remembering what it is; *Pop* the object, and verify that the two objects are equal.
  - *Push* three objects, remembering what they are; *Pop* each one, and verify that they are removed in the correct order.
  - *Pop* a *Stack* that has no elements.
  - *Push* a single object and then call *Top*. Verify that *IsEmpty* is false.
  - *Push* a single object, remembering what it is; and then call *Top*. Verify that the object that is returned is the same as the one that was pushed.
  - Call *Top* on a *Stack* with no elements.

UNBOUNDED STACK EXAMPLE

- Choosing the First Test?
  - The simplest.
  - The essence.

Answers:

- If you need to write code that is untested, choose a simpler test.

- If the essence approach takes to much time to implement, choose a simpler test.

UNBOUNDED STACK EXAMPLE

- Anticipating future tests, or not?

Answers:

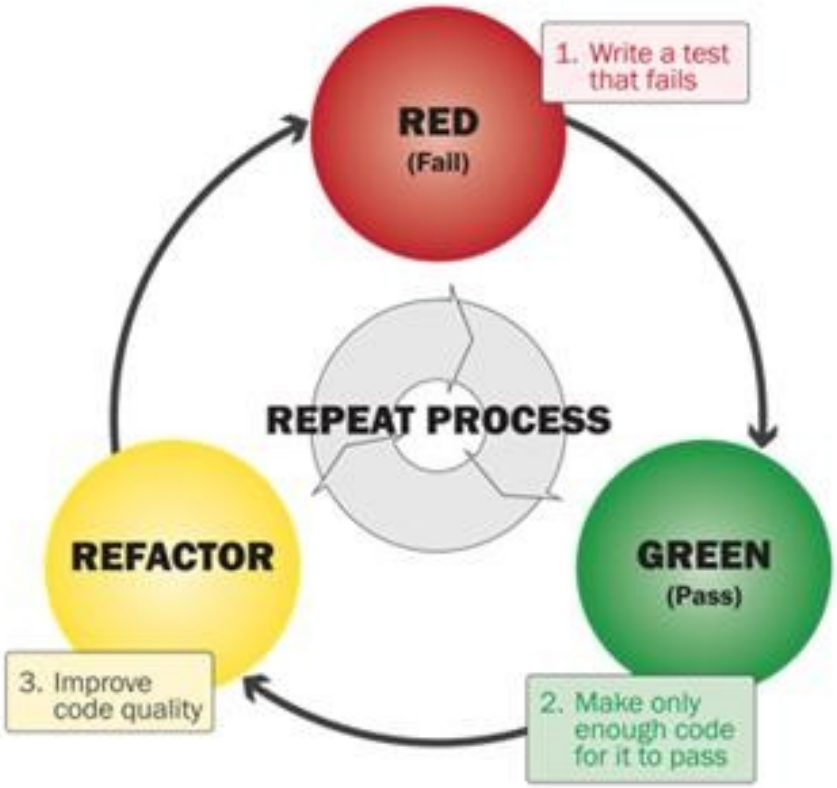
- In the beginning, focus on the test you are writing, and do not think of the other tests.
- As you become familiar with the technique and the task, you con increase the size of the steps.
- But remember still, no written code must be untested.

WHERE TO GO FROM HERE

- You don’t have to start big
- Start new tasks with TDD
- Add Tests to code that you need to change or maintain – but only to small parts
- Proof of concept

If it's worth building, it's worth testing.
If it's not worth testing,
why are you wasting your time working on it?

SUMMARY


1. Write a test that fails
RED
(Fail)
REPEAT PROCESS
GREEN
(Pass)
2. Make only enough code for it to pass
REFACTOR
3. Improve code quality

LINKS

- Books
  - Test-Driven Development in Microsoft® .NET (http://www.amazon.co.uk/gp/product/0735619484)
  - Test-Driven Development by Kent Beck (C++) (http://www.amazon.co.uk/gp/product/0321146530)

- Other links:
  - http://en.wikipedia.org/wiki/Test-driven_development
  - http://www.testdriven.com/
  - http://www.mockobjects.com/ - Online TDD book: http://www.martinfowler.com/articles/mocksArentStubs.html http://www.mockobjects.com/book/index.html
  - http://dannorth.net/introducing-bdd
  - http://behaviour-driven.org/Introduction