Test Driven Development

History

- Test-driven development is related to the test-first programming concepts of extreme programming, begun in 1999
- Kent Beck, who is credited with having developed or 'rediscovered' the technique, stated in 2003 that TDD encourages simple designs and inspires confidence

When to Write Tests?

- Write tests after implementation
  - test-last AKA code-first
- Write tests before implementation
  - test-first
- Write a test whenever a bug is found
  - Do not fix the bug once found
  - First develop a test case that exposes the bug
  - Fix the code and checks that new test case passes

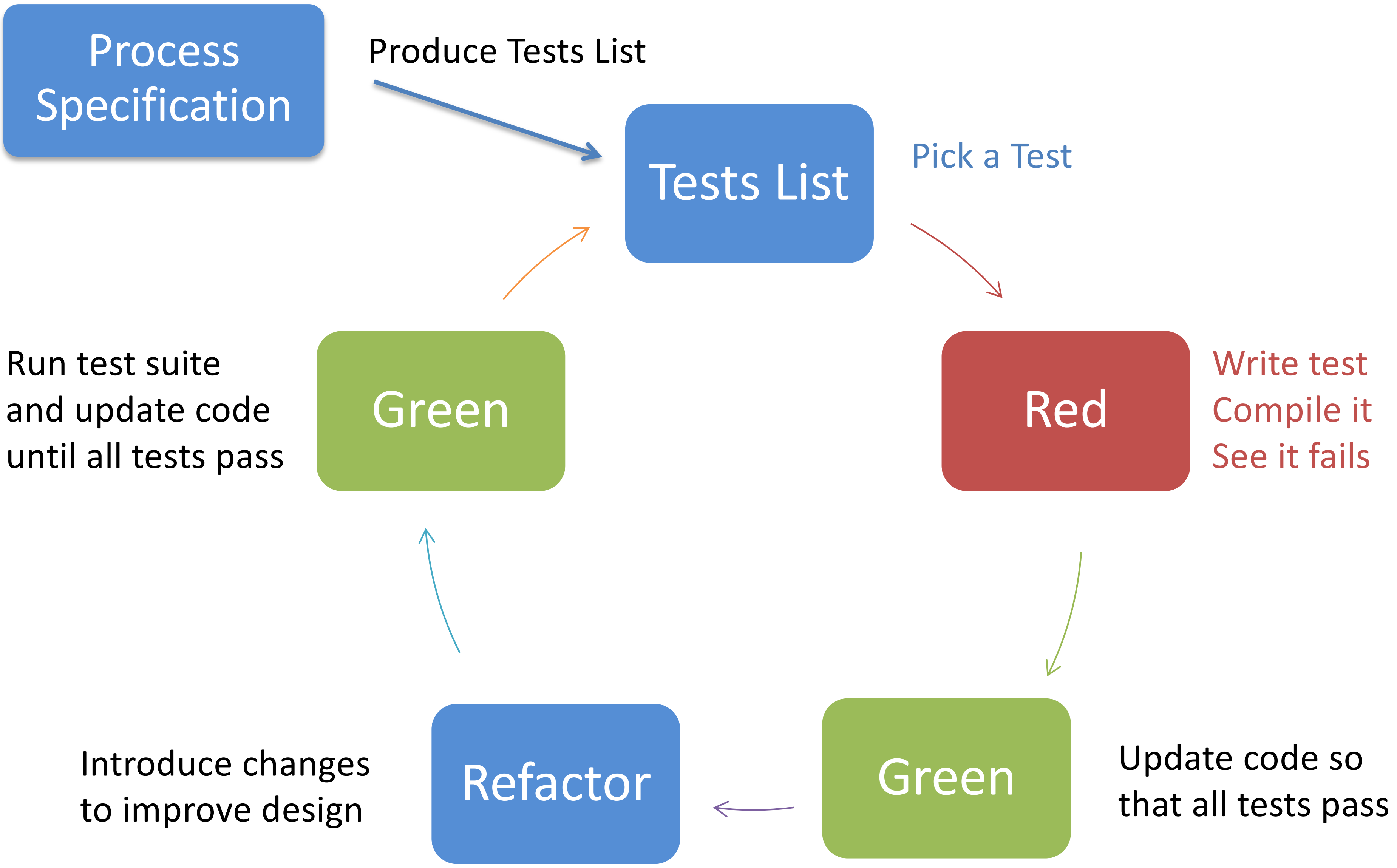
Test-last Approach

- Advantage
  - – Tests developed after functionality of SUT is *well understood*
- But
  - – Developer focus on testing the implementation (written some minutes before) instead of focus on behavior
    - Tests tightly coupled to implementation
    - Select the test cases with a higher probability of passing
  - – Why write tests if the code is running?

TDD Rhythm

- Test-first coding consists of a few very simple steps repeated over and over again
  - Incremental development
- Red, green, refactor
  - Write a test that fails
  - Make the code work
  - Eliminate redundancy
- And three simple rules:
  - **Never write code without a failing test**
  - Run all tests before new code
  - And after new code

TDD Rhythm - 2


Process Specification
Produce Tests List
Tests List
Pick a Test
Red
Write test
Compile it
See it fails
Green
Update code so that all tests pass
Refactor
Introduce changes to improve design
Green
Run test suite and update code until all tests pass

Pick up a test

- Next test to write?
- Can follow some heuristics:
  - **The Low-Hanging Fruit**
    - *Start with something really simple. Implement an obvious test case.*
  - **The Most Informative One**
  - **First The Typical Case, Then Corner Cases**
  - **Listen To Your Experience**

Red – Write a test that fails

- Test **must** represent an expected functionality of the code
- By writing the test first:
  - Know that the functionality really does not work
  - That the new test verifies behavior of the new functionality
    - High probability, it is not a proof
  - Once it is implemented, you will know it, red->green
- Change in coding habits
  - Great design opportunity

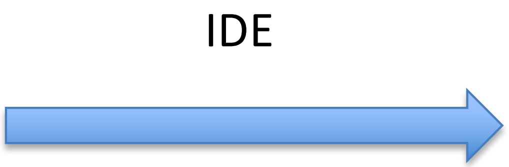
Red – Write a test that fails - 2

- By writing test first:
  - Make you think like a client of the SUT
  - Make you concentrate on what is really required
- Writing a test first will make you think in terms of the API of the SUT
  - More chances of testing behavior and not implementation details
  - Make tests more maintainable
- Make the test compile
  - May need to add some code before compiling

Should I follow it blindly?

- Simple rule: write a test and see it fail!

```
Client cl = new Client();
cl.setAge(20);
assertEquals(cl.getAge(), 20)
```



IDE

```
public class Client {
   private int age;
   public void setAge(int age) {
     this.age = age;
   }
   public int getAge() {
     return age;
   }
 }
```

- Test does not fail
- In trivial cases
  - Skip this restriction but make sure the test was executed

Should I follow it blindly - 2?

- In cases where the generated code is more complex
  - Do not take for granted that generated code works
  - How to be sure that test exercises just the required functionality?
- In this case:
  - Break the code to see test fail
  - Run the test and see it fail
  - Revert the change and rerun the test

Green - Write the simplest thing that works

- Just code the smallest and simplest amount of code that satisfies the test
  - Do not think (too much) about possible enhancements
  - Do not try to fulfill those requirements of tests that have not yet been implemented
  - **Concentrate on the task at hand**
  - "Simple" and "silly" are not synonyms!

Green - The simplest thing

- Make it run
  - Quickly getting that bar to go to green dominates everything else
- This shift in aesthetics is hard for some experienced software engineers
  - They only know how to follow the rules of good engineering
  - Quick green excuses all sins
  - This is not just about accepting sin; it is about being **sinful**
  - Write sinful code!

REFACTOR - Improve the code

- Adding simple functionalities one by one degrades design of code
- In previous step
  - Focused on the task at hand
  - Wrote smallest amount of code to pass the test
- Now
  - Look at the broader picture and refactor all code
    - Avoid redundancy
    - Change code from place
    - …
- Tests play a crucial role here

Clean code now

- The Refactoring Step is when we produce clean code
  - It’s when you apply patterns (see Joshua Kerievsky Refactoring to Patterns)
  - It is when you remove duplication

- You do not write new unit tests here
  - You are not introducing public classes

Clean code when?

- “Dependency is the key problem in software development at all scales.“ - Kent Beck, TDD By Example
- We need to eliminate dependency between our tests and our code
  - Tests should not depend on details, because then changing implementation breaks tests. Tests should depend on contracts or public interfaces
  - This allows us to refactor implementations without changing tests
  - Don’t bake implementation details into tests!
- Test behavior not implementation

REFACTOR - Improve the tests

- Should you refactor the tests?
- Yes, they are a valuable asset
  - The quality of the code depends on your tests
- However
  - No tests for tests
  - It is possible to introduce unwanted changes when refactoring the tests
    - Usually, this has a low probability

REFACTOR - Documentation

- Goal: code should be self-documenting
- Make code easier to read by adding comments
  - Do that in refactoring step
  - Keep comments short
    - The business purpose of the classes and methods
    - Any important design decision

Benefits of TDD - 1

- All code is exercised
  - At least one test case for each functionality
  - Keeps unused code out of application
  - Finds bugs earlier
  - Improves software quality

OUR GOAL IS TO WRITE BUG-FREE SOFTWARE. I'LL PAY A TEN-DOLLAR BONUS FOR EVERY BUG YOU FIND AND FIX.
YAHOO!
WE'RE RICH
YES!!! YES!!! YES!!!
I HOPE THIS DRIVES THE RIGHT BEHAVIOR.
I'M GONNA WRITE ME A NEW MINIVAN THIS AFTER-NOON!
S. Adams E-mail: SCOTTADAMS@AOL.COM
11/13 © 1995 United Feature Syndicate, Inc.(NYC)

Benefits of TDD - 2

- The code is written only to satisfy the tests
  - **YAGNI** principle
- Writing the smallest amount of code to make the test pass
  - Leads to simple solutions
  - **KISS** principle
- Refactoring phase make the code clean and readable
  - **DRY** principle
- Don´t lose time after interruption

Example – Race results service

- Race results service
  - Receive information about races
  - Notify interested parties about the result of races
- Requirements:
  - It should allow clients to subscribe
  - It should allow clients to unsubscribe
  - Every time a new message comes, it should be sent to all subscribers.

Example – Test cases

- Build test list
  - If client is subscribed, it should receive each new message once (and only once)
  - If multiple clients are subscribed, each of them should receive each new message
  - Consecutive subscribe requests issued by the same client will be ignored (nothing happens)
  - If the client unsubscribes, then it should be the case that no more messages are sent to it

Example – Apply TDD

- Build test list
  - If client is subscribed, it should receive each new message once (and only once)
  - If multiple clients are subscribed, each of them should receive each new message
  - Consecutive subscribe requests issued by the same client will be ignored (nothing happens)
  - If the client unsubscribes, then it should be the case that no more messages are sent to it

1. Pick up a test
2. Start TDD iteration with Red phase

1st Test: A subscriber receives message -- Red Phase --

```
@Test public class RaceResultsServiceTest {
  public void subscribedClientShouldReceiveMessage() {
    // Arrange
    RaceResultsService srv = new RaceResultsService();
    Message message;
    Subscriber client;
    client = mock(Subscriber.class); // Spy test double
    message = mock(Message.class);   // Dummy test double
    srv.addSubscriber(client);
    // Act
    srv.send(message);
    // Assert
    verify(client).receive(message);
  }
}
```

How to initialize ?

How to check behavior?

1st Test: Empty implementations

```
public interface Subscriber{
    void receive(Message message);
}

public interface Message {
}

public class RaceResultsService {
    public void addSubscriber(Subscriber s) {}
    public void send(Message message) {}
}
```

1st Test: Compile and run

- Compile, run tests
- Test fails. Yes! :

```
Wanted but not invoked:
client.receive(
    Mock for Message, hashCode: 986944742
);
-> at race.RaceResultsServiceTest.subscribedClientShould
ReceiveMessage(RaceResultsServiceTest.java:29)
Actually, there were zero interactions with this mock.
```

- RED phase of TDD
  - Done!

1^{st} Test : Implementation – Green Phase --

```
public class RaceResultsService {
  private Subscriber subscriber;

  public void addSubscriber(Subscriber s) {
    subscriber = s;
  }

  public void send(Message message) {
    subscriber.receive(message);
  }
 }
```

1st Test: TDD cycle

1. Compile, run the tests
   - All tests pass (GREEN)
   - GREEN phase is done

2. Do Refactor Phase
   - Source code
     - Comment code
   - Test code
3. Compile
4. Run, and if GREEN go to next test

1. Nothing to refactor in this case
2. Maybe add Javadoc comments

2nd Test: Send a message to multiple subscribers – Red Phase

- Implement test case

```
public void notifyAllSubscribedClients() {
    // Arrange
    RaceResultsService srv = new RaceResultsService();
    Message message = mock(Message.class);
    Subscriber clientA, client;
    clientA = mock(Subscriber.class , “clientA”);
    clientB = mock(Subscriber.class , “clientB”);
    srv.addSubscriber(clientA);
    srv.addSubscriber(clientB);
    // Act
    srv.send(message);
    // Assert
   verify(clientA).receive(message);
   verify(clientB).receive(message);
}
```

2nd Test: TDD Cycle

- Compile
- Run the tests: RED Phase
  - The new test fails since only one of the clients receives the message

```
Wanted but not invoked:
clientA.receive(
    Mock for Message, hashCode: 1982958205
);
-> at
race.RaceResultsServiceTest.messageShouldBeSentToAllSubscribedClients
(RaceResultsServiceTest.java:40)
Actually, there were zero interactions with this mock.
```

  - RED phase is finished
- **Can** go to next phase
  - Code to pass the test

2nd Test: Code for passing test – Green Phase

```
public class RaceResultsService {
  private Collection<Subscriber> subscribers =
                     new ArrayList<>();

  public void addSubscriber(Subscriber s) {
    subscribers.add(s);
  }

  public void send(Message message) {
     for(Subscriber subscriber : subscribers)
       subscriber.receive(message);
  }
 }
```

2nd Test: TDD cycle

1. Compile, run the tests and if GREEN,
2. Refactor Phase
   - Source code; not needed
   - Test code: Yes
3. Compile
4. Run, and if GREEN go to next test

```
@Test public class RaceResultsServiceTest {
   private  Subscriber client;
   private Message message;
   private RaceResultsService srv;

   @BeforeMethod private void setUp() {
      srv = new RaceResultsService();
      client = mock(Subscriber.class, “clientA”);
      message = mock(Message.class);
   }

   public void subscribedClientShouldReceiveMessage() {
       srv.addSubscriber(client);
       srv.send(message);
       verify(client).receive(message);
  }
  public void messageShouldBeSentToAllSubscribers() {
      Subscriber clientB = mock(Subscriber.class, “client”);
      srv.addSubscriber(client);
      srv.addSubscriber(clientB);
       srv.send(message);
       verify(client).receive(message);
       verify(clientB).receive(message);
   }
}
```

3rd Test: **Subscribe more than once --** **Red Phase**

- Implement test case

```
public void subscribeTwice() {
    // Arrange
    srv.addSubscriber(client);
    srv.addSubscriber(client);
    // Act
    srv.send(message);
    // Assert
    verify(client).receive(message);
}
```

3rd Test: TDD Cycle

- Compile
- Run tests – RED Phase

```
clientA.receive(
   Mock for Message, hashCode: 492252770
);
Wanted 1 time:
-> at race.RaceResultsServiceTest.subscribeTwice(RaceResultsServiceTest.java:51)
But was 2 times:
-> at race.RaceResultsService.send(RaceResultsService.java:28)
-> at race.RaceResultsService.send(RaceResultsService.java:28)
```

- RED phase is finished
- Go to next phase:
  - Code to pass the test

3rd Test: TDD cycle – Green Phase

- Implement to pass the test

```
public class RaceResultsService {
    private Collection<Client> clients =
                                new HashSet<Client>();
    // remains equal
}
```

1. Compile, run the tests and if GREEN,
2. Refactor Phase
   - Nothing to do in this case
3. Compile
4. Run, and if GREEN go to next test

4th Test: **Unsubscribed client stops receiving messages – Red Phase**

- Implement test case:

```
public void unsubscribedClientShouldNotReceiveMessages() {
    // Arrange
    clientB = mock(Subscriber.class, “clientB”);
    srv.addSubscriber(client);
    srv.addSubscriber(clientB);
    srv.removeSubscriber(client);
     // Act
    srv.send(message);
     // Assert
     verify(client, never()).receive(message);
     verify(clientB).receive(message);
}
```

4th Test: TDD Cycle

- Compile
  - Compilation error
  - Correct code: Add empty *removeSubscriber* method
- Run tests – RED Phase

```
clientA.receive(
    Mock for Message, hashCode: 981487964
);
Never wanted here:
-> at race.RaceResultsServiceTest.unsubscribedClientShouldNotReceiveMessages
(RaceResultsServiceTest.java:61)
But invoked here:
-> at race.RaceResultsService.send(RaceResultsService.java:28) with arguments:
[Mock for Message, hashCode: 981487964]
```

- RED phase is finished
- Go to next phase

4th Test: TDD Cycle – Green Phase

1. Code RaceResults with simplest code
   - Update removeSubscriber

```
public void removeSubscriber(Client sub) {
  subscribers.remove(sub);
}
```

   - Run tests -> GREEN
2. Go to Refactor phase
   - Nothing to do
3. Go to next test
   - No more tests. Finished

Conclusions so far

- More Test code than Production code
- The Interface is what Really Matters
  - Develop code in parallel
- Interactions can be tested
  - And cost is small
- Test Dependencies are there
  - Whether you like It or not
  - Ideally, test cases should be independently
  - Not always possible

Use Test Doubles or Not?

- Testing without test doubles is possible. But
  - Imposes a class development order
  - Test code may depend on some business logic
  - …
- But using test doubles may be overkill
  - It implies a dependency!
- When do not use test doubles:
  - The collaborator is very simple
    - Without any logic or with a very simple logic
    - DTO, Value Objects

TDD Best practices

- Naming Conventions
- Processes
- Development practices
- Tools

Naming conventions

- Help to organize test better

1. Separate implementation from test code
2. Place test classes in the same package as implementation
3. Name test classes in a similar fashion as classes they test
4. Use descriptive names for test methods
    1. Comments in test code does not appear when tests fail

Processes

- Write the test before writing the implementation code
  - Functionality coverage
  - Focus on requirements
- Only write new code when test is failing
- Rerun all tests every time implementation code changes
- All tests should pass before new test is written
  - Focus on a small unit of work
  - Implementation code is (almost) working as expected
- Refactor only after all tests are passing

Development practices

- Write the simplest code to pass the test
- Use setup and tear-down methods
- Do not introduce dependencies between tests
- Tests should run fast
- Use test doubles
- Minimize assertions in each test
  - Test a **single** condition

Minimize assertions in each test – Good examples

```
@Test
public final void whenOneNumberIsUsedThenReturnValueIsThatSameNumber() {
  assertEquals(3, StringCalculator.add("3"));
}

@Test
public final void whenNegativeNumbersAreUsedThenRuntimeExceptionIsThrown() {
  RuntimeException exception = null;
  try {
    StringCalculator.add("3,-6,15,-18,46,33");
  } catch (RuntimeException e) {
    exception = e;
  }
  assertNotNull("Exception was not thrown", exception);
  assertEquals("Negatives not allowed: [-6, -18]", exception.getMessage());
}
```

Minimize assertions in each test – Bad example

```
@Test
public final void whenAddIsUsedThenItWorks() {
  assertEquals(0, StringCalculator.add(""));
  assertEquals(3, StringCalculator.add("3"));
  assertEquals(3+6, StringCalculator.add("3,6"));
  assertEquals(3+6+15+18+46,StringCalculator.add("3,6,15,18,46"));
  assertEquals(3+6+15, StringCalculator.add("3,6n15"));
  assertEquals(3+6+15, StringCalculator.add("//;n3;6;15"));
  assertEquals(3+1000+6, StringCalculator.add("3,1000,1001,6,1234"));
}
```

Tools

- Code coverage

- Continuous integration

Testing first clarifies the task

- A test is a small, self-contained action
- It becomes an example to help understand what the code needs to do
- Also acts as a checkpoint; if you don’t understand the problem well enough to write a test case, you aren’t ready to write the code
- Might grapple on how to write the test, but the time also helps you write the code

Fixing broken test cases

- You modify code or introduce a bug and as a result, tests don’t run correctly. What now?
- Goal is to make the tests pass
- Two distinct cases:
  1. Might require refactoring the test cases themselves to match new code signatures
  2. Might require searching through the code to find out why the test case fails

Tests Suites and Sanity

- Test suites psychologically help the team's frame of mind
  - Successful passing of tests strokes your inner programmer
  - Stronger boost when you see a new/better/more efficient way to write your code, and can see that all of the tests still pass

Shortcomings

- Tests become part of the maintenance overhead of a project
  - A refactoring phase goal: write tests that are easy to maintain
- Over-testing can consume time both to write the excessive tests, and later, to rewrite the tests when requirements change