

Using TDD and BDD to deliver
tested-customer value.

About me

- Michael Collier CSM, CSP, CSD
- Software Developer
- Agile Champion since 2008
- Musician & Entertainer
- Scrum Coach & Developer



Certified Scrum Professional




Certified Scrum Developer



Certified ScrumMaster



 @pappymcbeard
#agilepirate
www.musicalblades.com



Our Objectives

- Understand how the customer's behavior drives acceptance tests.
- Creating acceptance tests and unit tests from the acceptance criteria.
- Criteria for writing easy to understand unit tests.
- RED – GREEN – REFACTOR – REPEAT
- What is and is not refactoring.
- Pairing, it really is for everyone.

Manifesto for Agile Software Development

We are uncovering better ways of developing software by doing it and helping others do it.

Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

Principles behind the Agile Manifesto

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity--the art of maximizing the amount of work not done--is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Agile Development Processes

Lean Software Development

- Eliminating Waste
- Amplifying Learning
- Deciding as Late as Possible
- Delivering as Fast as Possible
- Empowering the Team
- Building Integrity In
- Seeing the Whole

Agile Development Processes

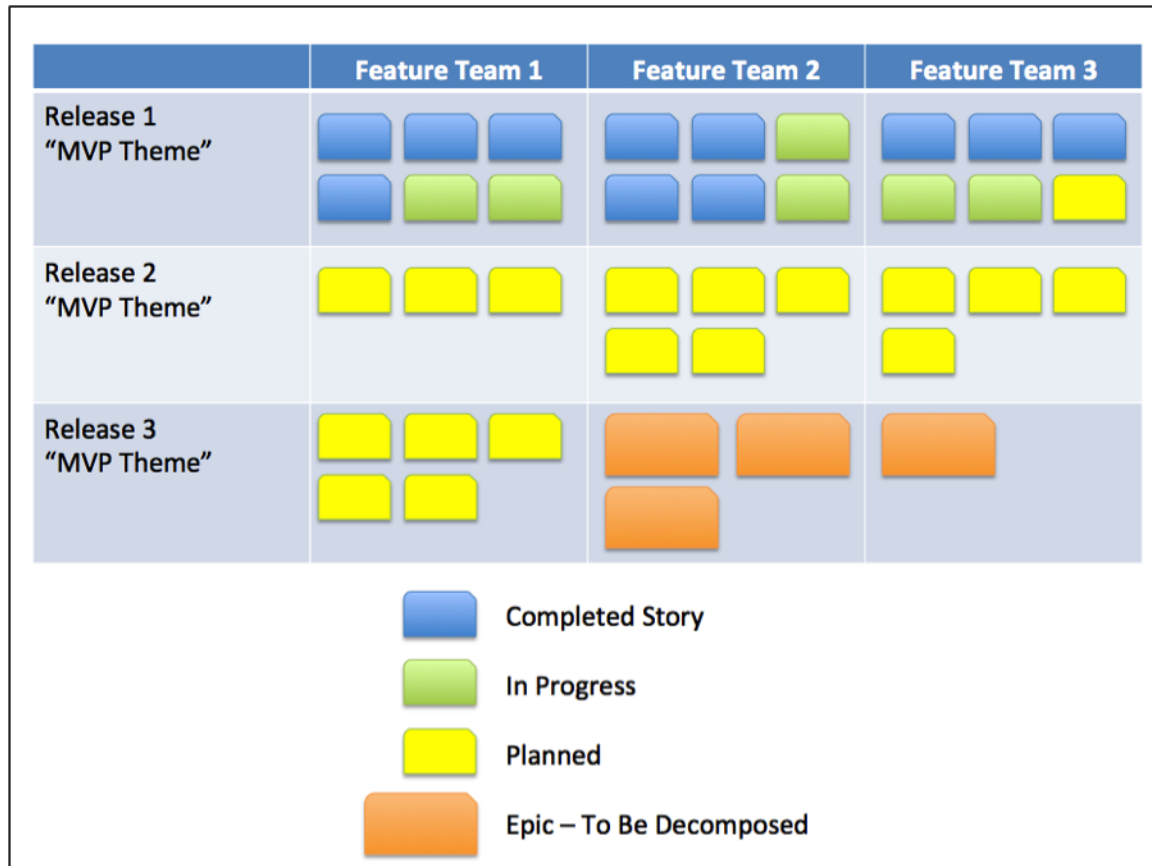
Extreme Programming (XP)

- simplicity,
 - communication
 - feedback
 - courage
-
- | | |
|---|---|
| <ul style="list-style-type: none">• Planning Game• Small Releases• Customer Acceptance Tests• Simple Design• Pair Programming• Test-Driven Development | <ul style="list-style-type: none">• Refactoring• Continuous Integration• Collective Code Ownership• Coding Standards• Metaphor• Sustainable Pace |
|---|---|

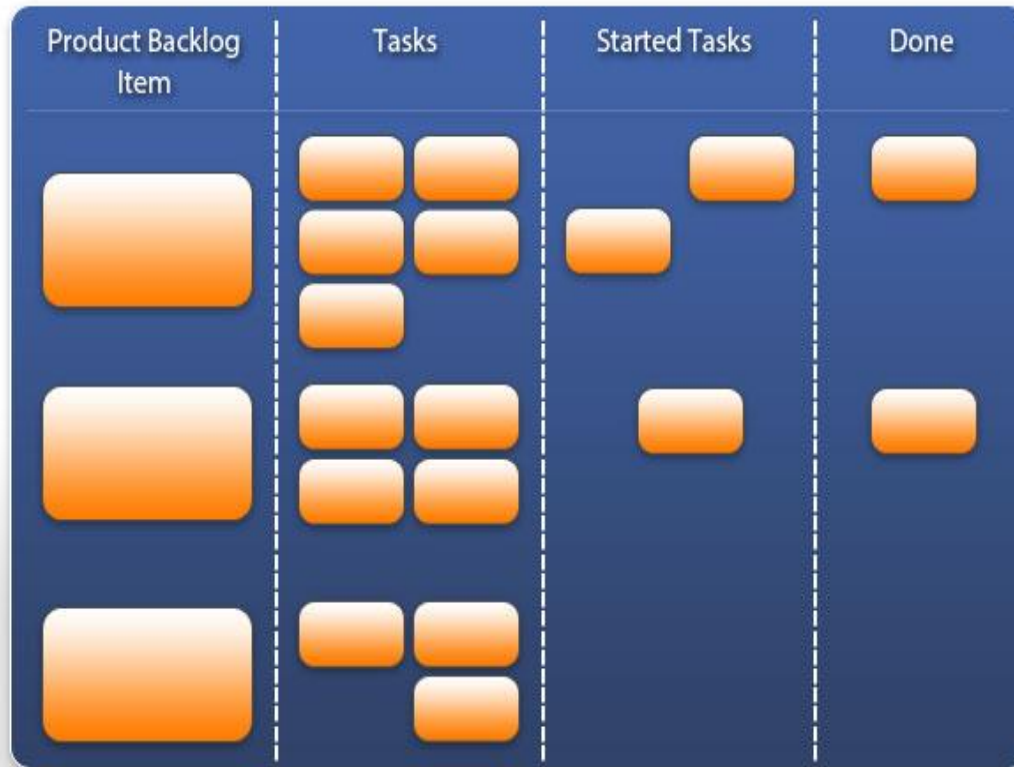
Story Map



MVP



Sprint Planned



User Story

Story: Account Holder withdraws cash

As an Account Holder

I want to withdraw cash from an ATM

So that I can get money when the bank is closed

Ready to start coding!?!



Get the full story



The Elephant



User Stories

A short, simple description of a feature told from the perspective of the person who desires the new capability, usually a user or customer of the system.

Michael Cohn

- User stories are focused very much around the user and what they are trying to achieve.
- A user story does not *necessarily* focus on why the business is in need of this new feature
- User stories maybe transient



Acceptance Criteria

- A set of statements, each with a clear pass/fail result
- The stuff that lets you know when a user story is functionally complete
- Executable
- Specify both functional and system requirements
- Can be ad-hoc and prone to vary in quality
- Often does not say enough to enable the creation of robust tests
- Constitutes our basic “Definition of Done”

Test Driven Development

- TDD is very good at detailed specification and validation.
- TDD **alone** can be a costly bottom up approach. You can end up with hundreds of low level tests.
- It may not provide a test that proves you created a feature that actually delivers the promised value.
- Not so good at thinking through bigger issues such as how people will use the system.
- Behavior driven development (BDD) attempts to address all of these issues and more.

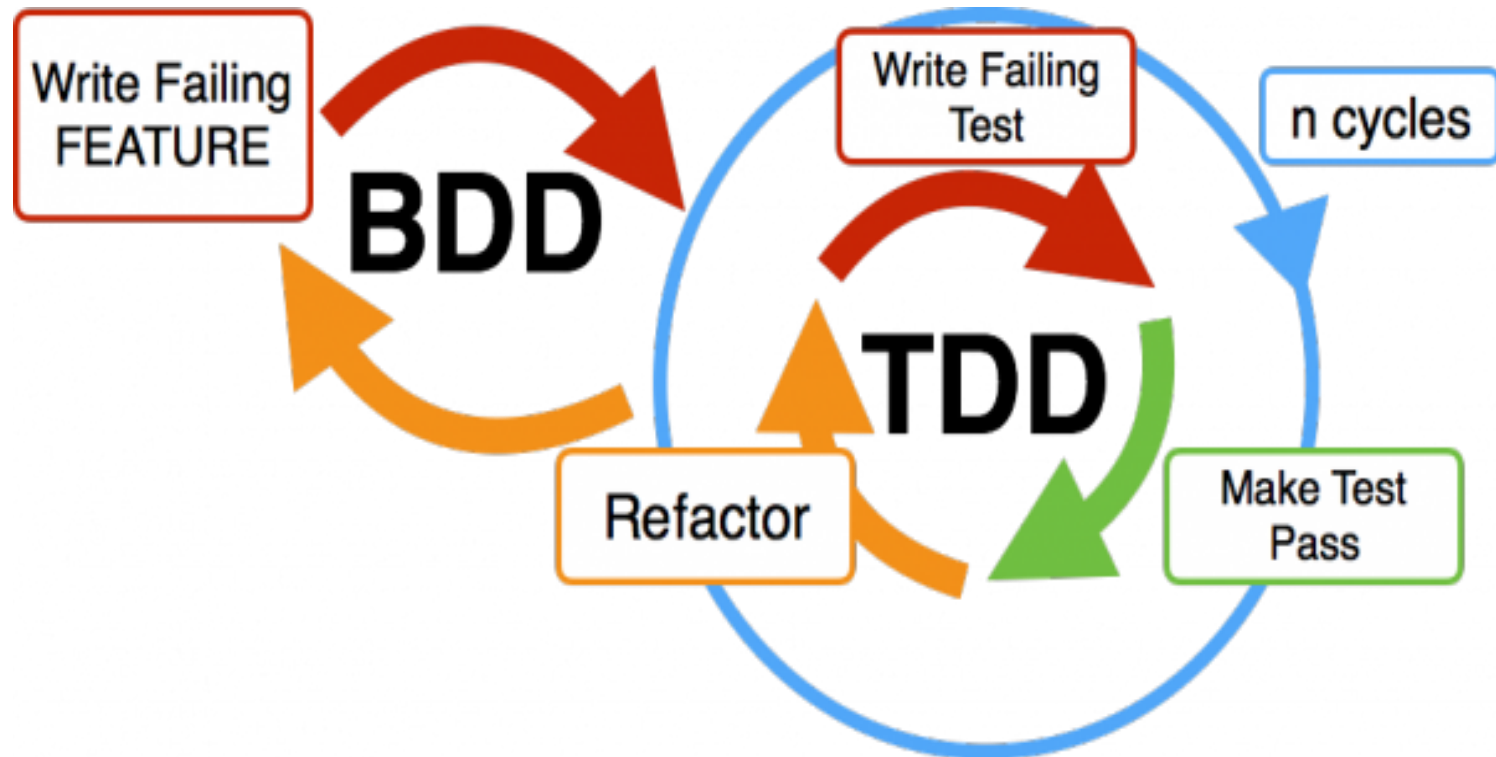
Product Owner Tests

- As the Customer how would you test this story?
- What would you do to validate that we have covered your expectations for this story?
- Are there scenarios that we might have missed in the initial discussions?
- Do we have a shared understanding of this story?

Product Owner processes

- How many times have you read a user story or requirement and not been able to make heads nor tails of it?
- How many times have you had a conversation with a customer or product owner and only found out later that you both had different versions of what you had “agreed” on?

Process for Writing Unit Tests



“If you can take the idea of defining required behavior for a simple unit test, then why can't you define required behavior for acceptance tests?”

“Distinct features of an application can be communicated effectively between all members of the project team, from product owner through to business analysts through to software developers through to test engineers.”

Introduction to behavior driven development

- Dan North

Given, When, Then

Scenario:

What is different from other scenarios for this story

Given

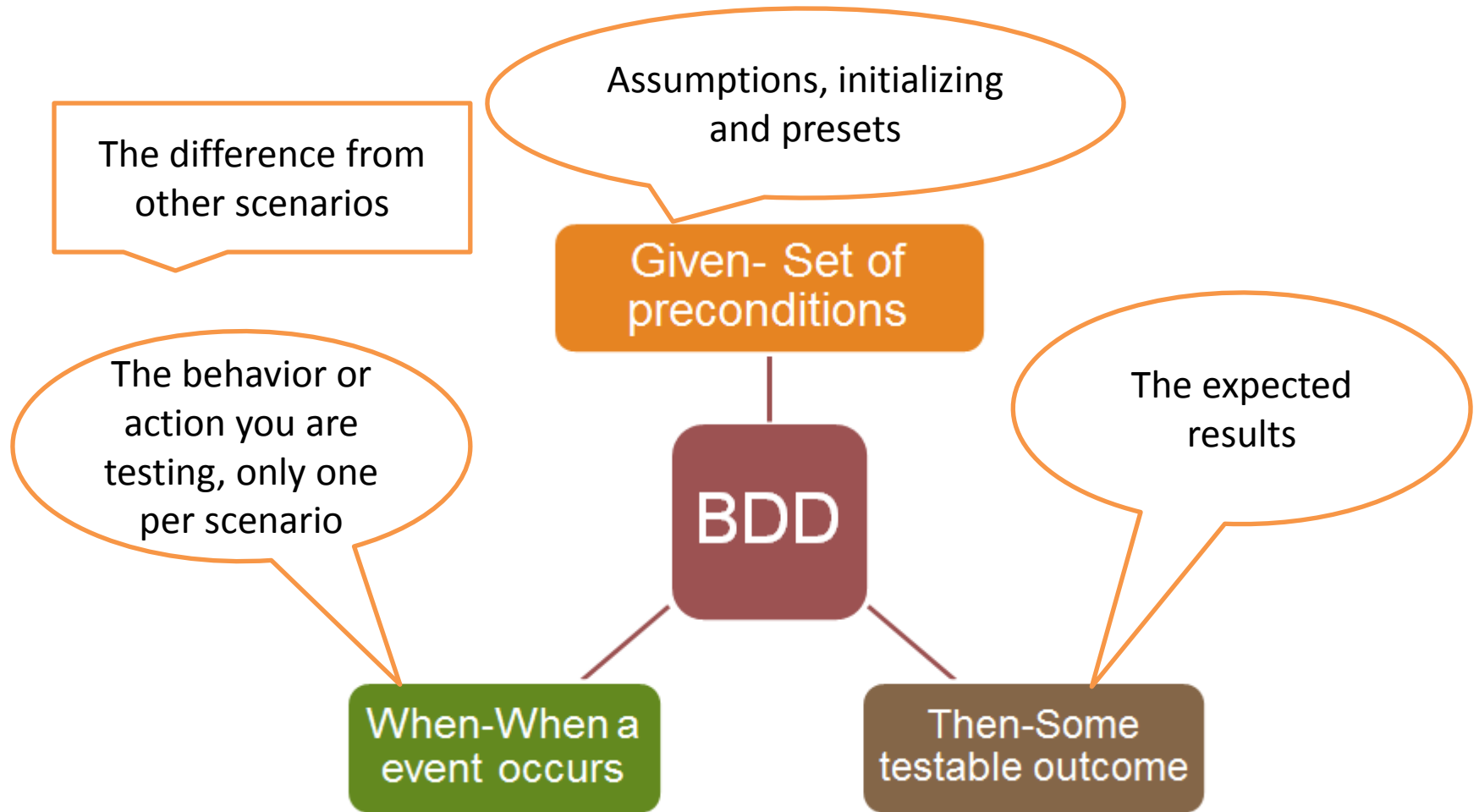
These are any and all assumptions that you're making in this test. Oftentimes, this will end up being just instantiating whatever objects will be needed for the rest of the test joined by **And**.

When

This is the action that's going to take place. Most **When** methods are actually just one line in length (i.e. make a call to a particular method and store the result).

Then

This is where the test determines if the result of the Given and When sections meets what was expected in the test. All of the assertions are done in this method itself.



Story: Account Holder withdraws cash
As an Account Holder
I want to withdraw cash from an ATM
So that I can get money when the bank is closed

Scenario 1: Account has sufficient funds

Given

the account balance is \ \$100

And the card is valid

And the machine contains enough money

When

the Account Holder requests \ \$20

Then

the ATM should dispense \ \$20

And the account balance should be \ \$80

And the card should be returned

Scenario 2: Account has insufficient funds

Given

the account balance is \ \$10

And the card is valid

And the machine contains enough money

When

the Account Holder requests \ \$20

Then

the ATM should not dispense any money

And the ATM should say there are insufficient funds

And the account balance should be \ \$20

And the card should be returned

Scenario 3: Card has been disabled

Given

the card is disabled

When

the Account Holder requests \ \$20

Then

the ATM should retain the card

And the ATM should say the card has been retained

Unit Tests

- Unit Tests are **FIRST**

- Fast
- Isolated
- Repeatable
- Self-Verifying
- Timely

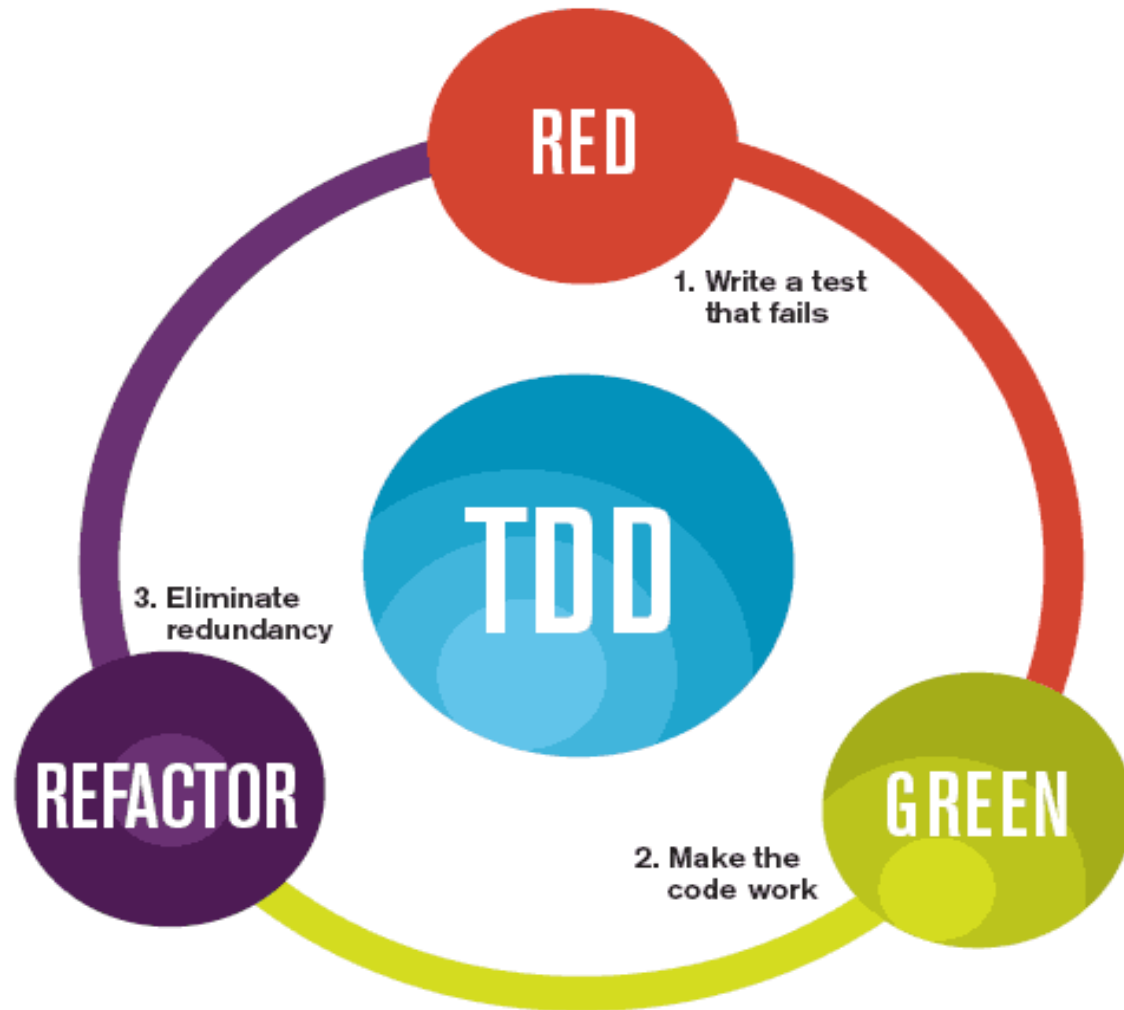
- This is not a Unit Test:
 - It talks to a database
 - It communicates across a network
 - It touches the file system
 - You have to do special things to your environment
 - These would be considered integration tests

Source: The Art of Agile Development

Unit Test Naming

- Should be expressive.
- Easy to understand.
- Contain what is being tested and expected results.
- Full sentences work great.
- The team should agree on convention.

Process for Writing Unit Tests



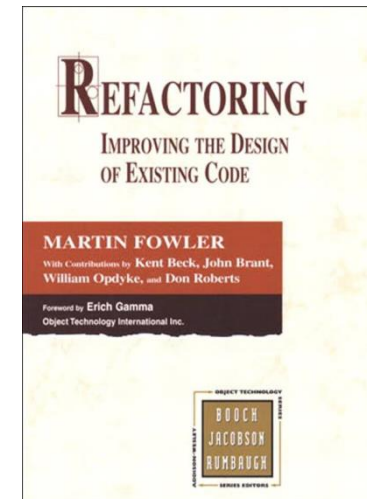
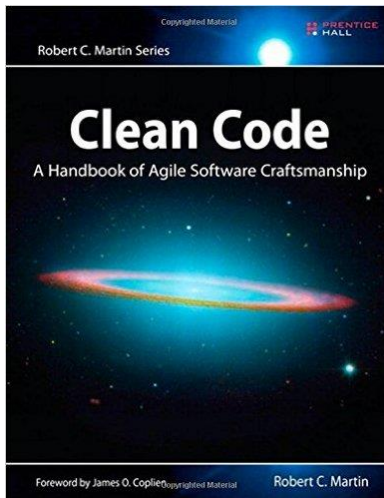
The mantra of Test-Driven Development (TDD) is “red, green, refactor.”

Refactoring

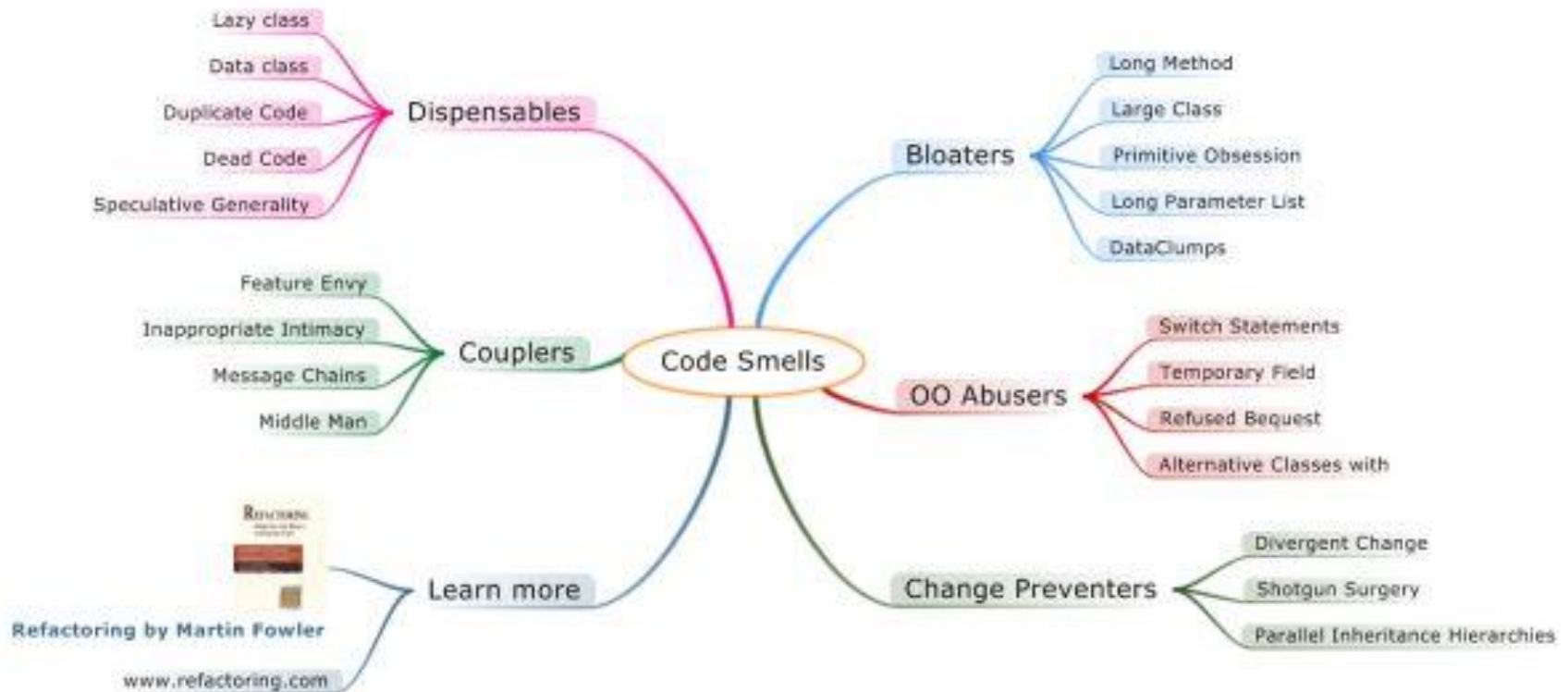
- **Refactoring is:**
 - The process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.
 - A method to make it easier to understand and cheaper to modify code.
- **Refactoring is not:**
 - Debugging
 - Adding features
 - Changing observable behavior
 - Performance improving

Benefits of Refactoring

- Improved software design
- Reduced code size
- Confusing code is restructured into simpler code
- Code is easier to maintain



Code smells (refactor opportunities)



- `@Test(expected = NullPointerException.class) public void test_null_height_input() { }`
- `@Test(expected = NullPointerException.class) public void test_null_weight_input() { }`

Given, When, Then

Scenario:

Should add to numbers correctly

Given

That I have a calculator


When

I add two numbers

Then

Then I should get the correct answer.

The Calculator Tests






```
private Calculator _calc;
private int _resultOfAddTwoNumbersTest;

[TestCase(1,2)]
public void ShouldAddNumbersCorrectly(int firstNumber, int secondNumber)
{
    Given_That_I_Have_A_Calculator();
    When_I_Try_To_Add_Two_Numbers(firstNumber, secondNumber);
    Then_I_Should_Get_The_Correct_Answer(firstNumber + secondNumber);
}

private void Given_That_I_Have_A_Calculator()
{
    _calc = new Calculator();
}

private void When_I_Try_To_Add_Two_Numbers(int first, int second)
{
    _resultOfAddTwoNumbersTest = _calc.AddNumbers(first, second);
}

private void Then_I_Should_Get_The_Correct_Answer(int expectedResult)
{
    Assert.IsNotNull(_resultOfAddTwoNumbersTest);
    Assert.AreEqual(_resultOfAddTwoNumbersTest, expectedResult);
}
```



The Calculator class

```
public class Calculator
{
    //Constructor possibly here...

    public int AddNumbers(int first, int second)
    {
        return first + second;
    }

    //Other Calculator methods...
}
```

User Story

Story: Account Holder orders a checkbook

As an Account Holder

I want to order a checkbook

So that I can write checks

Given, When, Then

Scenario:

Valid Address

Given

The account is in credit

And the user has been authenticated

And the user's address is available

When

The user clicks on 'order a Check book'

Then

Send check book to user

Given, When, Then

Scenario:

No address available

Given

The account number is 12345

And the user has been authenticated

And the user's address is UNKNOWN

.When

The user clicks on 'order a check book'

Then

Reply NO_ADDRESS to the user's request

Coding the tests

```
private Customer _cust;  
private Account _acct;  
private bool _resultOfAddressCheck;  
  
[TestMethod]  
public void ShouldReturnTrueIfNoAddressForCustomer()  
{  
    Given_That_I_Have_A_Customer();  
    Given_That_I_Have_Valid_Account_Number("12345");  
    Given_That_I_Have_An_Authenticated_User(1);  
    When_Ordering_Checkbook();  
    Then_I_Should_Respond_If_Address_Is_Not_Present(true);  
}
```

```

[TestMethod]
public void Given_That_I_Have_A_Customer()
{
    throw new NotImplementedException();
    _cust = new Customer();
}

[TestMethod]
public bool Given_That_I_Have_Valid_Account_Number(String accountid)
{
    throw new NotImplementedException();
    return String.IsNullOrEmpty(accountid);
    _acct = new Account();
    return _acct.IsValidAccountNumber(accountid);
}

[TestMethod]
public bool Given_That_I_Have_An_Authenticated_User(long userid)
{
    return _cust.IsAuthenticated(userid);
}

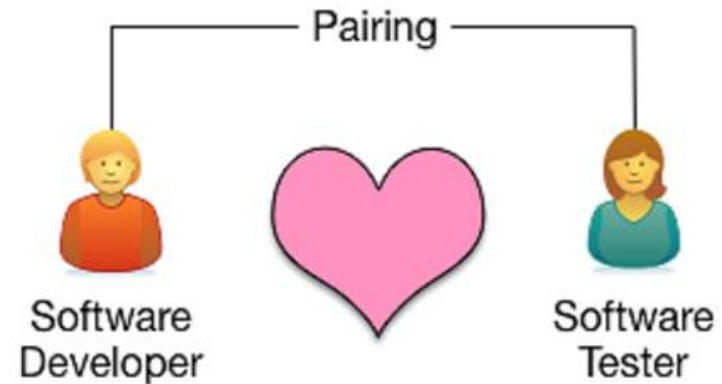
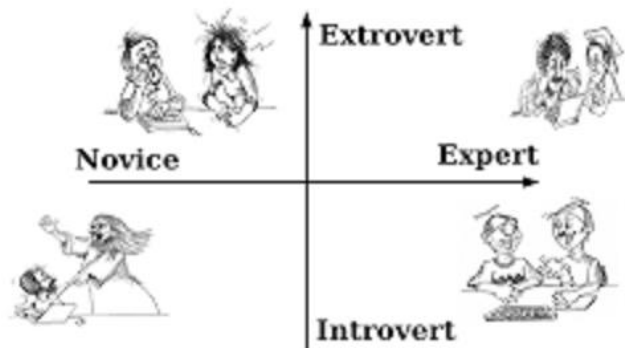
[TestMethod]
public void When_Ordering_Checkbook()
{
    _resultOfAddressCheck = String.IsNullOrEmpty(_cust.Address);
}

[TestMethod]
public void Then_I_Should_Respond_If_Address_Is_Present_Or_Not(bool expectedResult)
{
    Assert.AreEqual(_resultOfAddressCheck, expectedResult);
}

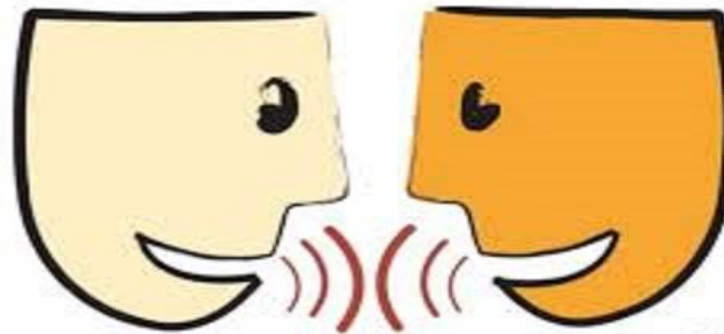
```

Collaborate through Pairing

DIFFERENT TYPES OF PAIRS



PAIR UP WITH PRODUCT OWNER



@NPradeepa

Build Quality, Fully Tested Product

- Are there zero “bugs” per sprint in the business logic of completed stories?
- Are all developers confident making changes to the code?
- Do developers have less than one debug session per week that exceeds 10 minutes?
- Are developers nearly always confident that the code they’ve written recently does what is intended?
- Is the Product Owner accepting the DONE story each Sprint?

Great User Stories combined with BDD and TDD drives customer value and satisfaction.

What have we learned

- Understand the product through collaborative shared understanding.
- Trust and collaboration is essential!
- Ask questions to get great User stories with clear acceptance criteria.
- Tie all Unit tests back to the acceptance criteria of the user story
- RED – GREEN – REFACTOR – REPEAT
- Refactoring is not rework.
- Quality code now – or fix technical debt later – you decide

Sources

- **The Art of Agile Development**
 - James Shore
- **Cleaner code**
 - Robert (Uncle Bob) Martin
- **Refactoring: Improving the Design of Existing Code**
 - Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts
- <http://ronjeffries.com/>
- <https://dannorth.net/>

Questions

