REFACTORING

What is Refactoring

Refactoring is the art of improving the design of existing code. Refactoring provides us with ways to recognize problematic code and gives us recipes for improving it.

What is Not Refactoring

- Refactoring does **not** include just any changes in the system.
- Refactoring is **not** rewriting from scratch.
- Refactoring is **not** just any restructuring intended to improve.
- Refactoring is **not** fixing a problem in code when all tests are passing.

Reasons:- Easy to add new features/code

Design erodes very quickly. Developers add features and hack the design to accomplish short-term goals. Refactoring helps in maintaining the design.

Reasons:-
Improves readability and understanding

If you don't understand the code, maintenance becomes a nightmare. Refactoring helps in-

- Removing duplicates and dead code
- Giving proper names to methods and classes
- Removing unnecessary comments
- Making short methods and small classes
- Applying a proper design pattern to conserve the open/closed principle
- Making code clean and easy to change.

Reasons:-
Improves the design of the existing code

Refactoring helps us focus on emergent design and coding principles, some of these principles are listed below. If any piece of code violates these design principles, refactor the code.

CODING PRINCIPLES

REFACTORING

DRY

DRY stands for **Don't Repeat Yourself**. If a class has a duplicate code, it violates the DRY principle. If we find any bug in the duplicate code, we have to fix the same code in all places. This principle helps in removing code duplication—so bug fixing is easy.

OPEN/CLOSED

This principle states that a piece of code should be open for extension but closed for modification, that means the design should be done in such a way that a new functionality should be added with minimum changes in the existing code.

YAGNI

This principle refers to over engineering. The full form of YAGNI is **You Aren't Gonna Need It**. Add code for today's feature, not for tomorrow. Test Driven Development and refactoring helps us focus on emergent design.

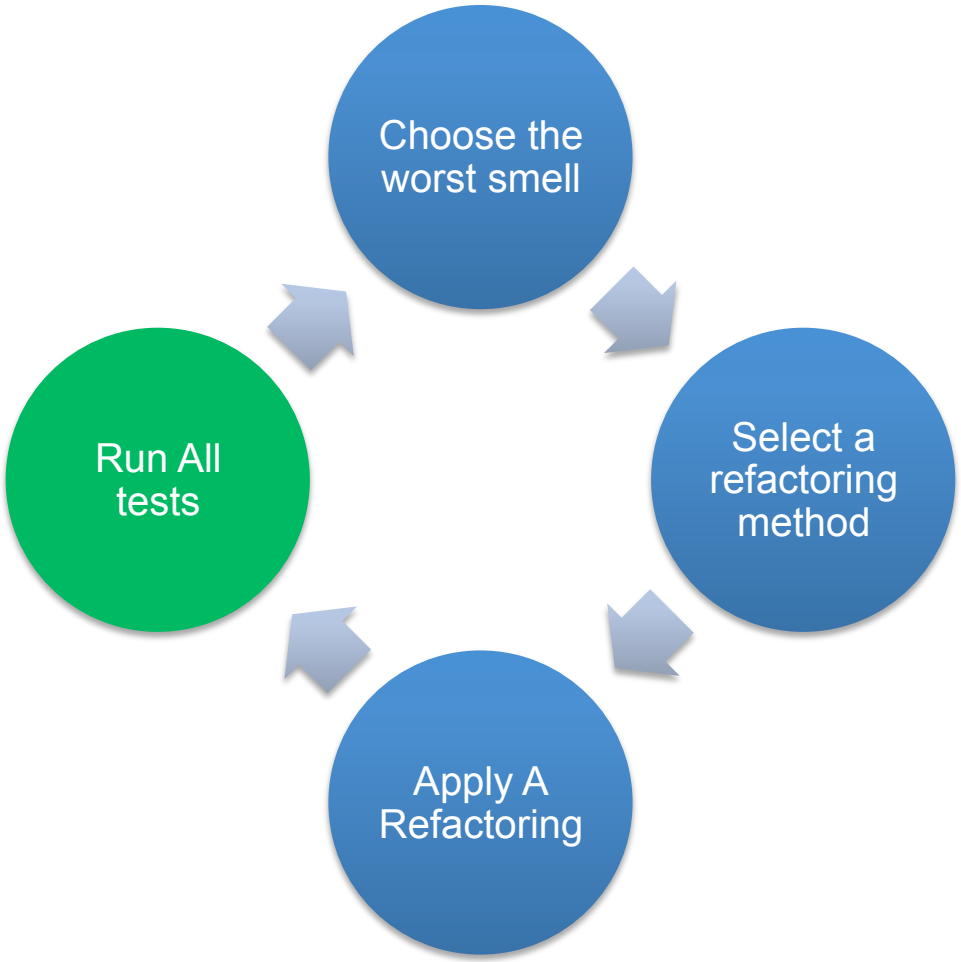
KISS

KISS, "**Keep It Simple Stupid**" is a principle that suggests favoring simplicity over complexity. Simple code is easier to read and consume.

There are many code smells that deal with detecting complexity and refactoring it to simpler code. These smells deal with specific areas of complexity such as class-level, method-level.

Simplicity can apply in many other places.

Refactoring Cycle



Pre-conditions

- All Tests are Up-to-date, If tests need to be updated do so prior to refactoring
- All Tests are **Green**

Post-conditions

- Refactored code should be as good as the pre-existing code or better
- All Tests are **Green**

SMELLS IN CODE

REFACTORING

Simple Smells

- Comments
- Long Method
- Large Class
- Long Parameter List

Smells in Name

- Type Embedded in Name
- Uncommunicative Name
- Inconsistent Names

Smells in Complexity

- **Dead Code**: A variable, parameter, field, code fragment, method, or class is not used anywhere (perhaps other than in tests).
- **Speculative Generality**: There are unused classes, methods, fields, parameters, and such. They may have no clients or only tests as clients.

Smells in Duplication

- Duplicated Code
- Alternative Classes with Different Interfaces

Smells in Conditional Logic

- Null Check
- Complicated Boolean Expression

REFACTORING TECHNIQUES

REFACTORING

Extract Method

```
void printSectionOnTop()
{
    printBanner();

    //print details
    System.out.println("name: " + name);
    System.out.println("amount: " + amount);
}
```

- You have a code fragment that can be grouped together.
- The more lines found in a method make it difficult to figure out what the method does.

Extract Method

Problem

```
void printSectionOnTop()
{
    printBanner();

    //print details
    System.out.println("name: " + name);
    System.out.println("amount: " + amount);
}
```

Treatment

```
void printSectionOnTop()
{
    printBanner();

    //print details
    printDetails(amount);
}


void printDetails(double amount)
{
    System.out.println("name: " + name);
    System.out.println("amount: " + amount);

}
```

Benefits:

- Improved readability.
- Gateway to finding other smells.
- An important step in many other refactoring techniques.

Replace Temp with Query

```
double calculateEarnings() {
    double earning = quantity * itemPrice;
    if (earning > 1000) {
        return earning * 0.95;
    }
    else {
        return earning * 0.98;
    }
}
```

- You place the result of an expression in a local variable for later use in your code.

Replace Temp with Query

Problem

```
double calculateEarnings() {
    double earning = quantity * itemPrice;
    if (earning > 1000) {
        return earning * 0.95;
    }
    else {
        return earning * 0.98;
    }
}
```

Treatment

```
double calculateEarningsAfterTax() {
    if (earning() > 1000) {
        return earning() * 0.95;
    }
    else {
        return earning() * 0.98;
    }
}
double earning() { return quantity * itemPrice; }
```

Benefits-

- Improved Code Readability
- Slimmer Code by reducing duplication

- This refactoring can lay the groundwork for applying Extract Method for a portion of a very long method.
- The same expression may sometimes be found in other methods as well, which is one reason to consider creating a common method.

Reduce Scope of Variable

```
//Reduce scope of local variable

void aFunction()
{
    int i = 7;

    // i is not used here

    if (someCondition)
    {
        // i is used only within this block
    }

    // i is not used here
}
```

- You have a local variable declared in a scope that is larger than where it is used

Reduce Scope of Variable

Problem

```
//Reduce scope of local variable

void aFunction()
{
    int i = 7;

    // i is not used here

    if (someCondition)
    {
        // i is used only within this block
    }

    // i is not used here
}
```

Treatment

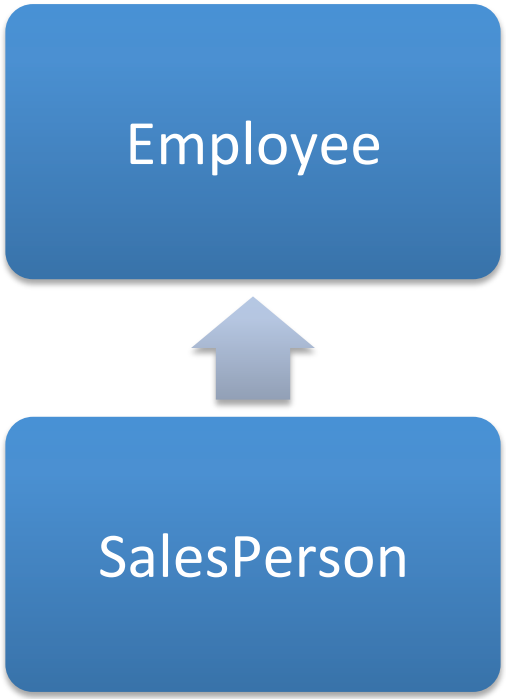
```
void aFunction()
{
    // i can not be used here

    if (someCondition)
    {
        int i = 7;

        // i is used only within this block
    }

    // i can not be used here
}
```

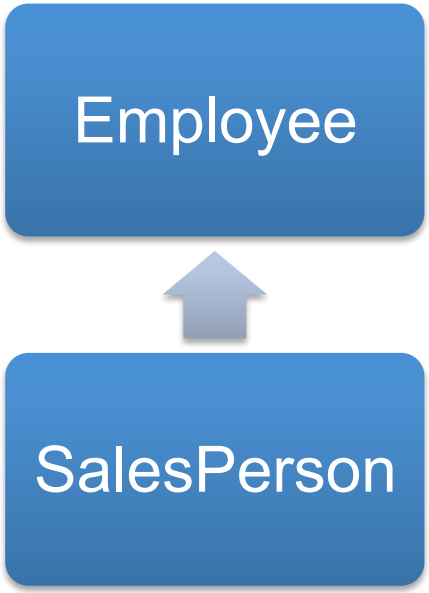
Collapse Hierarchy



- You have a class hierarchy in which a **subclass is practically the same as its superclass.**
- Programs may grow over time , subclass and superclass may become practically the same

Collapse Hierarchy

Problem



Treatment

Exercise: Refactoring

30 Minutes

Goal : Refactor the provided java code

Identify problems in the Java classes from here:
https://github.com/shekhar2010us/tdd/tree/master/src/main/java/shekhar/refactor/before

Solution is here
https://github.com/shekhar2010us/tdd/tree/master/src/main/java/shekhar/refactor/after

When Are we Done with refactoring?

- Passes all the tests.
- Has no duplicated logic.
- States every intention important to the programmers.
- Has the fewest possible classes and methods.

“If you want to refactor, the essential precondition is having solid tests.”

Martin Fowler

Creating an Environment for Refactoring

- Docker comes handy for testing

Tests: Strong test support is an essential precondition to refactoring.

it's possible to make a mistake while applying refactoring. By Running test suites before and after refactoring, we can ensure that you change the design of your code, not its effects.

Pair Programming or Swarming: For Important refactoring decisions pair programming or swarming are very useful techniques. A team can often generate better ideas while working together.

PAIR PROGRAMMING

Introduction to Pair Programming

Pair programming is a technique in which two programmers jointly work on same problem, such as a design, an algorithm, or some code.

Experiments have demonstrated that **pairing improves design quality, reduces defects**, **enhances technical skills, improves team communications**, and is considered more enjoyable at statistically significant levels

Basic Rules

- Share Everything
- Be Comfortable
- Stop When You're Tired
- Debate with Your Partner

Results: Time

Results: Defects

Source: The Costs and Benefits of Pair Programming, Alistair Cockburn & Laurie Williams

Pairing Setup

- 2 Mirror Monitors
- Common Problem
- 1 Keyboard
- 2 Developers

Pairing Participants

Driver

- Myopic (near-sightedness)
- Task @ hand

Navigator

- Hyperopic (far-sightedness)
- Larger Context

Pair Rotation

- Never assign pairs, Pairs should be fluid
- When you become stuck or frustrated, switch Pairs for a fresh perspective
- Switch several times per day
- Rotate Driver & Navigator roles
- Take frequent breaks

Technique: Ping Pong Pairing – Model 1

A

1. Write a Test,
2. Run the Test

B

3. Write code
4. Run the test
5. Pass the Test

Technique: Ping Pong Pairing – Model 2

A

1. Write a Test,
2. Run the Test

8. Write code
9. Run the test
10. Pass the Test
11. Write a New Test
12. Run the test

B

3. Write code
4. Run the test
5. Pass the Test
6. Write a New Test
7. Run the test

Promiscuous Pairs

Pairs are switched very frequently (every 90-120 minutes) , swapping out the most experienced member of the pair each time

Benefits

- This can lead to a lot of knowledge sharing in the team

Cross-Functional Pairing

Instead of two software engineer working together,

one software engineer works with another member with different role.

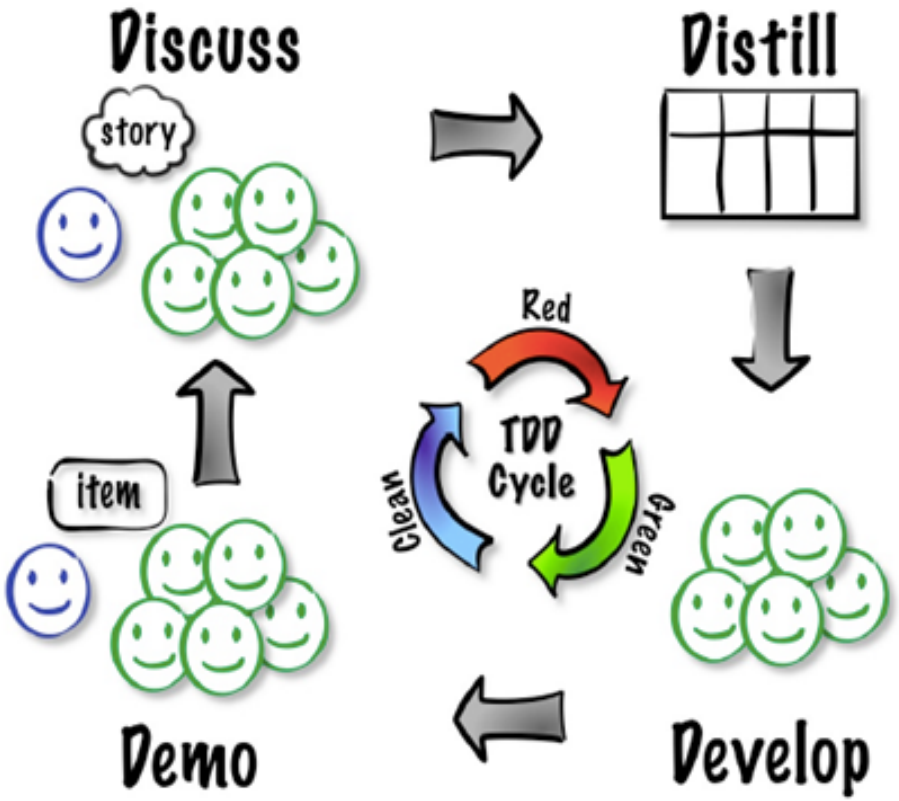
- Developer, Tester
- Developer, BA
- BA, Tester

ACCEPTANCE TEST DRIVEN DEVELOPMENT

Introduction to ATDD

Acceptance-test-driven development (**ATTD**) A technique in which the participants collaboratively discuss acceptance criteria, using examples, and then distill them into a set of concrete acceptance tests before development begins.

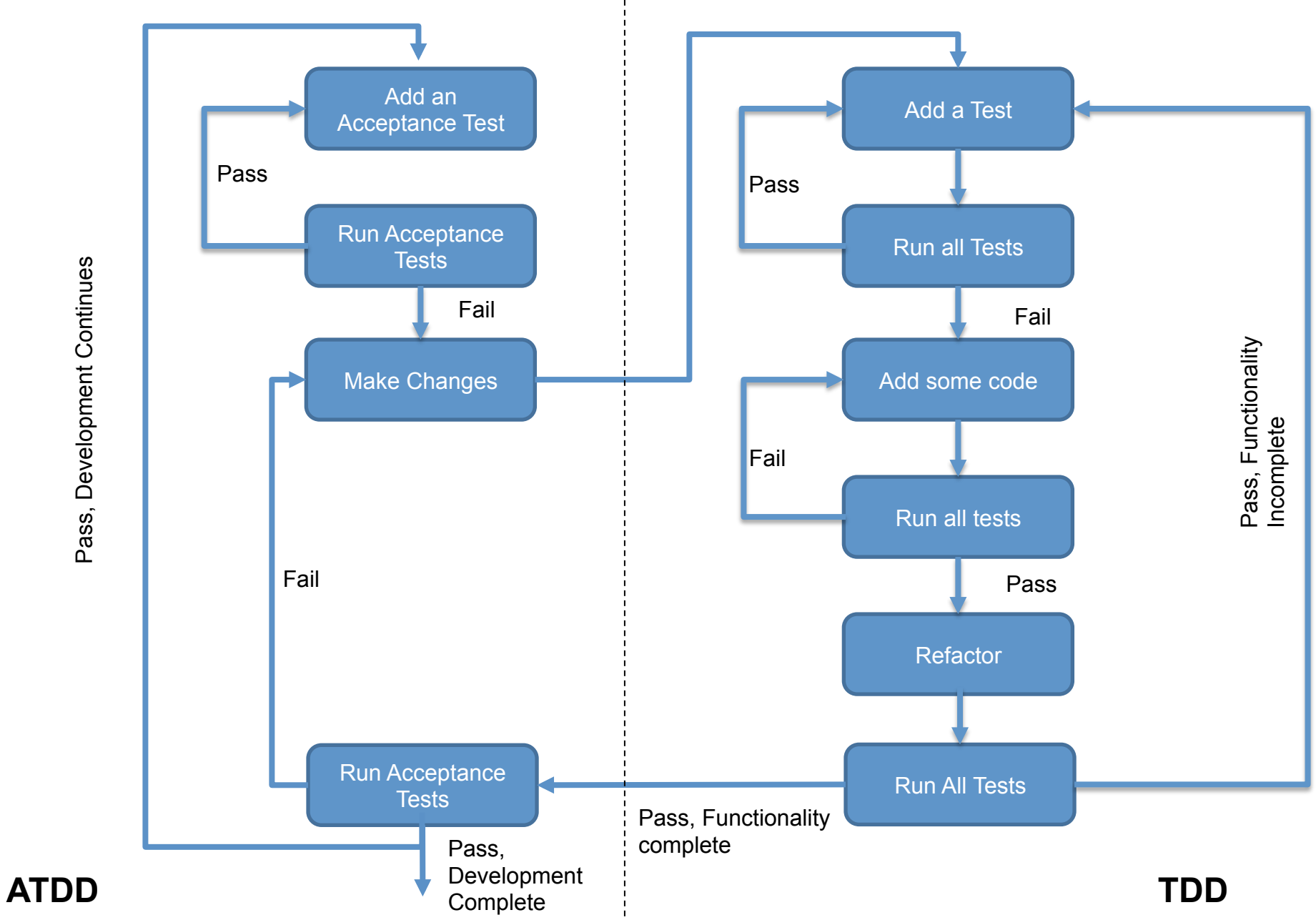
ATDD Cycle



ATDD cycle model developed by James Shore with changes suggested by Grigori Melnick, Brian Marick, and Elisabeth Hendrickson

TDD and ATDD

- TDD and acceptance TDD work together seamlessly.
- At feature/story level, we run our development process with ATDD.
- Inside the implementation step of each feature we employ TDD.
- They are by no means tightly coupled, but they are powerful in combination and they do fit together seamlessly.


Add an Acceptance Test
Pass
Run Acceptance Tests
Fail
Make Changes
Pass, Development Continues
Fail
Run Acceptance Tests
Pass, Development Complete
ATDD
Add a Test
Pass
Run all Tests
Fail
Add some code
Fail
Run all tests
Pass
Refactor
Run All Tests
Pass, Functionality complete
Pass, Functionality Incomplete
TDD

BEHAVIOR DRIVEN DEVELOPMENT

Introduction to BDD

Behavior Driven Development (BDD) is an extension and refinement of practices stemming from Test Driven Development (TDD) and Acceptance Test Driven Development (ATDD).

Continued - Introduction to BDD

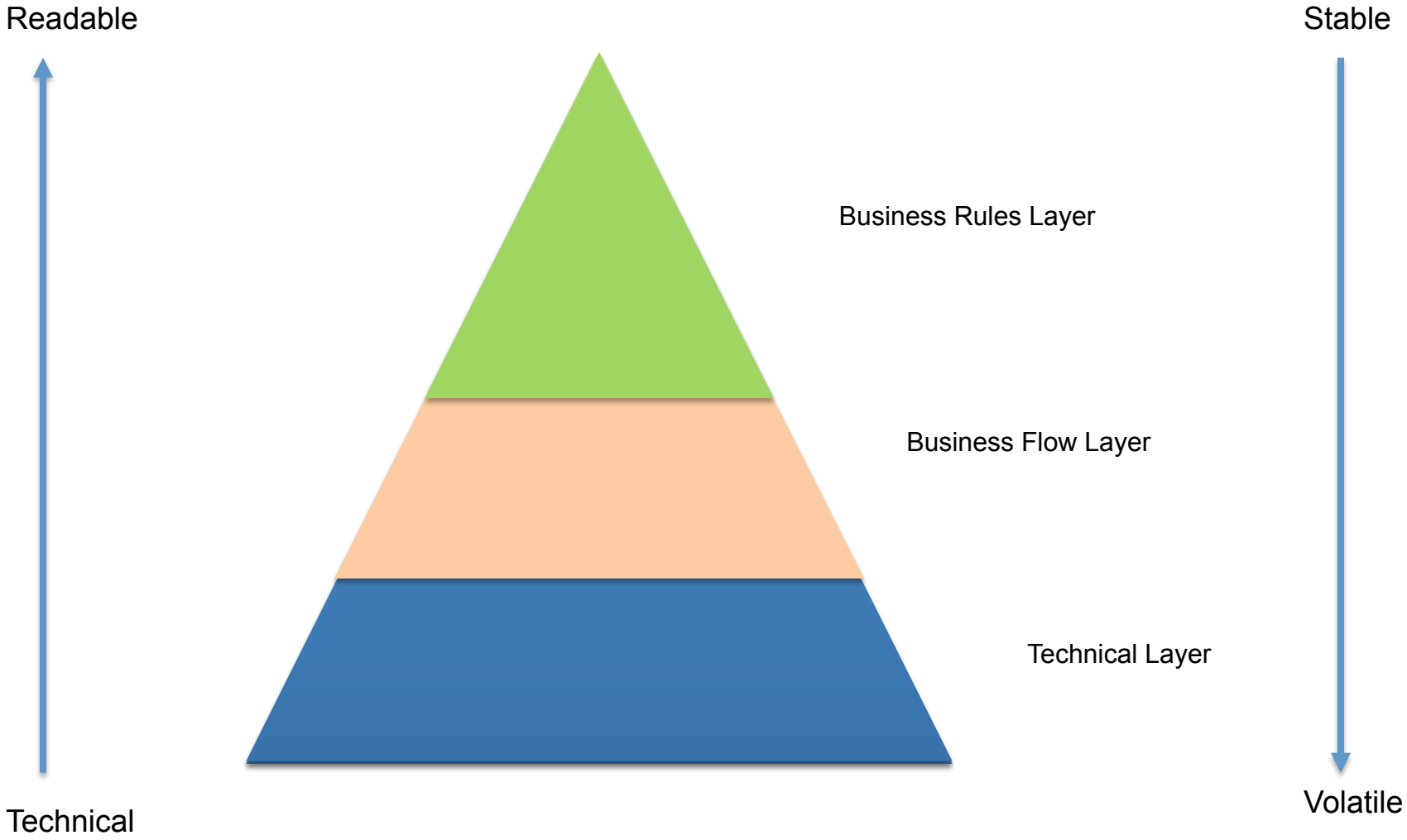
When working on a requirement, there can be a disconnect between:

- the product owner being able to define the desired outcomes
- the developer's understanding of what needs to be built, and
- The product owner's understanding of the technical challenges their requirements may present.

Behavior Driven Development (BDD) can help achieve all of the above and ultimately, helps a business and its technical team deliver software that fulfils business goals.

Continued - Introduction to BDD

- As organizations attempt the move to DevOps, one of the building blocks they need to incorporate is the use of a **common language**.
- The use of a common language is implemented in order to prevent teams and team members from misinterpreting what their actual deliverables (critical behaviors) are.
- This common language is then used to drive the development of the CI/CD/DevOps infrastructure and the automated application testing.


Readable
Stable
Business Rules Layer
Business Flow Layer
Technical Layer
Technical
Volatile

Business Rules Layer

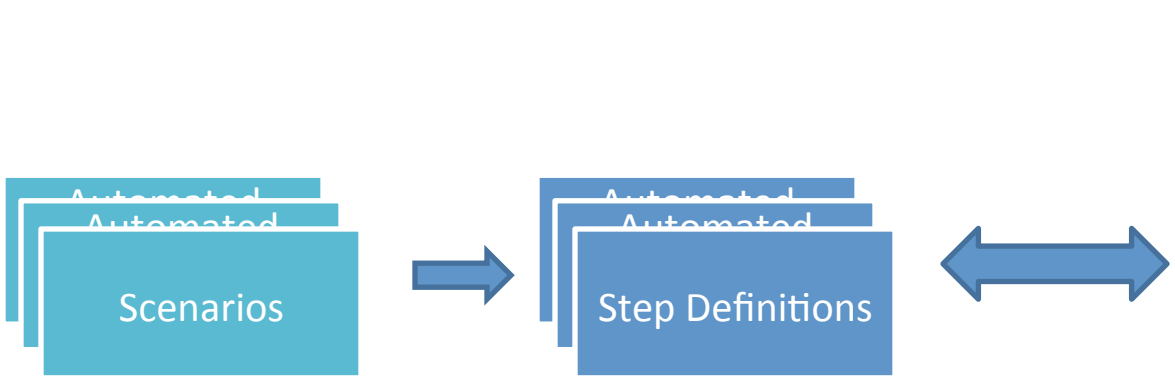
Business Rules layer describes the requirement under test in high-level business terms.
If you are using a BDD tool like Cucumber, business rules are typically written as a scenario in a feature file.

```
Scenario: Timeline Display order
  Given Timeline has atleast 10 updates
  When Timeline is opened for the first time by user
  Then  User should see 10 updates in descending order of time when they were posted
```

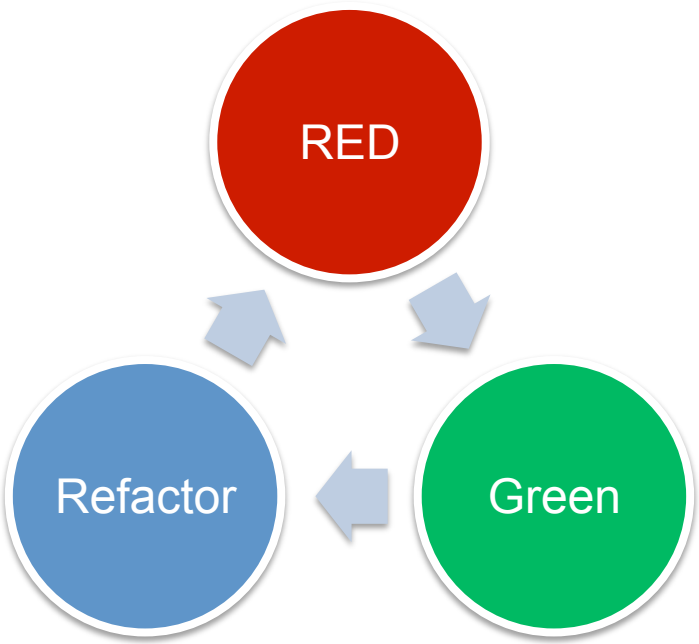
Technical Layer

Technical layer represents how the user interacts with the system at a detailed level—how they navigate to the registration page, what they enter when they get there, how you identify these fields on the HTML page, and so forth.

TDD and BDD Together



Business Layer



Technical Layer

Cucumber

- https://examples.javacodegeeks.com/core-java/junit/junit-cucumber-example/

FIRST PROPERTIES OF GOOD TEST

UNIT TESTING PRINCIPLES

FIRST Principles of Unit Testing

Follow FIRST principles to avoid common unit testing failure modes:

- [F]ast
- [I]solated
- [R]epeatable
- [S]elf-validating
- [T]imely

[F]IRST: [F]ast!

- On a typical java projects you would regularly see thousands of unit test cases
- If an average test takes 1 second, you'll wait over eight minutes each time to run 2,500 unit tests.
- **Keep your tests fast!**
- Minimize dependencies on code that executes slowly.

F[I]RST: [I]solate Your Tests

- Good unit tests focus on a small chunk of code to verify.
- The code you're testing might interact with other code that reads from a database.
- Tests that ultimately depend on a database require you to ensure that the database has the right data.
- Other developers are often running their tests at the same time, there is a high chance of their changes breaking .
- **Good unit tests also don't depend on other unit tests.**
- **Tests should not be dependent on order of test execution.**

FI[R]ST: Good Tests Should Be [R]epeatable

- A repeatable test is one that produces the same results each time you run it.
- Without repeatability tests may give false negative results.
- Tests must be able to be run repeatedly without intervention.
- **Tests must not depend upon any assumed initial state.**
- **Repeatable tests do not depend on external services or resources that might not always be available.**
- Tests should run whether or not the network is up, and whether or not they are in the development server's network environment.

FIR[S]T: [S]elf-Validating

- Manually verifying the results of tests is a time-consuming process, Tests should be designed to be self validating.
- If you use Eclipse or IntelliJ IDEA, consider incorporating a tool like Infinitest. As you make changes to your system, Infinitest identifies and runs (in the background) any tests that are potentially impacted.
- On an even larger scale, you can use a continuous integration (CI) tool such as Jenkins or TeamCity. A CI tool watches your source repository and kicks off a build/test process when it recognizes changes.

FIRS[T]: [T]imely

- You can write unit tests at virtually any time. But its best to focus on writing unit tests in a timely fashion.
- You should have guidelines or strict rules around unit testing.
- Some teams use review processes or even automated tools to reject code without sufficient tests.

The RIGHT-BICEP

ADVANCED UNIT TESTING

Right-BICEP

Right-BICEP provides you with the strength needed to ask the right questions about what to test.

Right	Are the results right?
B	Are all the boundary conditions correct?
I	Can you check inverse relationships?
C	Can you cross-check results using other means?
E	Can you force error conditions to happen?
P	Are performance characteristics within bounds?

[Right]-BICEP: Are the Results Right?

- Your tests should first and foremost validate that the code produces expected results.
- Derive tests from the requirements, user stories, domain knowledge, acceptance criteria…
- If the requirements are incomplete
  - Make your assumptions (but it's dangerous, so ASK!!)
  - Make the code do what you think it should
  - Get user feedbacks and fine tune your assumptions

Right-[B]ICEP: Boundary Conditions

Boundary conditions you might want to think about include:

- Inconsistent input values, file filename with special characters
- Badly formatted data, such as an email like “will@gmail”
- Calculation that can result in numeric overflow.
- Empty or missing values, 0 or null.
- Values far in excess of reasonable expectations, such as a person’s age of 550 years.
- Duplicates in lists that shouldn’t have duplicates.
- Ordered lists that aren’t actually ordered.

Right-B[I]CEP: Checking Inverse Relationships

Some method can be checked by applying their logical inverse

Examples:

- Check a method that calculate square root by squaring the result
- Check that some data was successfully inserted in to the DB by searching for it AND taking count pre and post insert operation

Right-BI[C]EP: Cross-Checking Using Other Means

Do you have other means to check the results?

- String concat: check the length of the result string and not only concatenated string
- Checking database entry from business logic and not only through direct queries

Right-BIC[E]P: Forcing Error Conditions

Errors happen, even when we think they can't possibly.
You want to test that your code handles all of these real-world problems in a graceful or reasonable manner.
To do so, you need to write tests that force errors to occur.

- Out of memory
  - **libvirt**
  - long[] li = new long[INTEGER.MAX_VALUE];
  - Set low -Xmx values
- Out of disk space
- Network availability and errors
- System load
  - **loadtest** "loadtest [-n requests] [-c concurrency] URL"

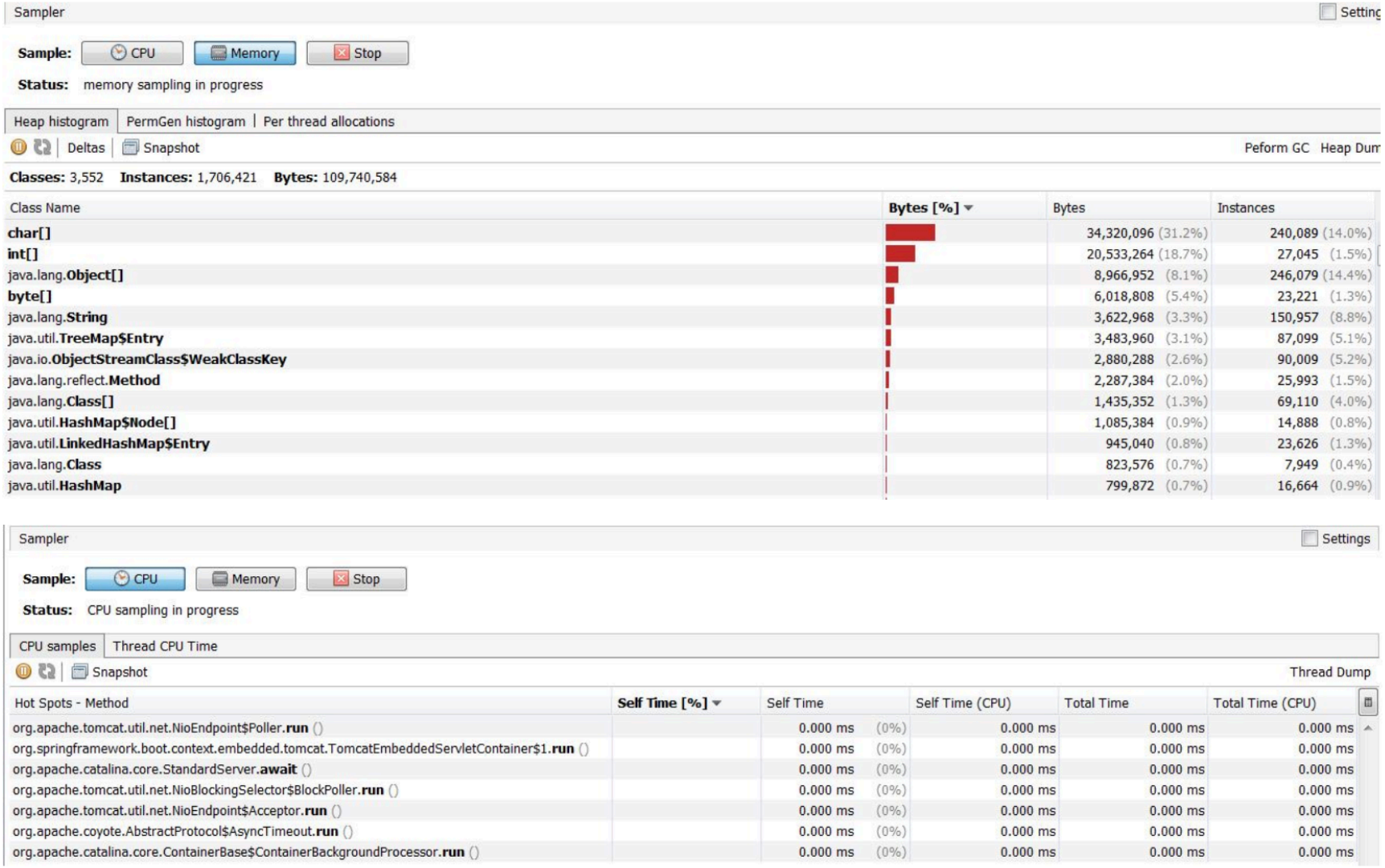
Right-BICE[P]: Performance Characteristics

Design unit tests to help you know where performance problems lie and whether your proposed changes make enough of a difference.

Some tips to consider:

- Run performance tests separately from your fast unit tests.
- Log and monitor execution times – in different machines/scenario
- Base all performance-optimization attempts on real data, not speculation.
- Monitor resource utilization for each code and for each variable.
  - **JVisualVM**

JVisualVM


Sampler
Setting
Sample: CPU Memory Stop
Status: memory sampling in progress
Heap histogram | PermGen histogram | Per thread allocations
Deltas | Snapshot
Peform GC Heap Dum
Classes: 3,552 Instances: 1,706,421 Bytes: 109,740,584
Class Name | Bytes [%] | Bytes | Instances
char[] 34,320,096 (31.2%) 240,089 (14.0%)
int[] 20,533,264 (18.7%) 27,045 (1.5%)
java.lang.Object[] 8,966,952 (8.1%) 246,079 (14.4%)
byte[] 6,018,808 (5.4%) 23,221 (1.3%)
java.lang.String 3,622,968 (3.3%) 150,957 (8.8%)
java.util.TreeMap$Entry 3,483,960 (3.1%) 87,099 (5.1%)
java.io.ObjectStreamClass$WeakClassKey 2,880,288 (2.6%) 90,009 (5.2%)
java.lang.reflect.Method 2,287,384 (2.0%) 25,993 (1.5%)
java.lang.Class[] 1,435,352 (1.3%) 69,110 (4.0%)
java.util.HashMap$Node[] 1,085,384 (0.9%) 14,888 (0.8%)
java.util.LinkedHashMap$Entry 945,040 (0.8%) 23,626 (1.3%)
java.lang.Class 823,576 (0.7%) 7,949 (0.4%)
java.util.HashMap 799,872 (0.7%) 16,664 (0.9%)

Sampler
Settings
Sample: CPU Memory Stop
Status: CPU sampling in progress
CPU samples | Thread CPU Time
Snapshot
Thread Dump
Hot Spots - Method | Self Time [%] | Self Time | Self Time (CPU) | Total Time | Total Time (CPU)
org.apache.tomcat.util.net.NioEndpoint$Poller.run () 0.000 ms (0%) 0.000 ms 0.000 ms 0.000 ms
org.springframework.boot.context.embedded.tomcat.TomcatEmbeddedServletContainer$1.run () 0.000 ms (0%) 0.000 ms 0.000 ms 0.000 ms
org.apache.catalina.core.StandardServer.await () 0.000 ms (0%) 0.000 ms 0.000 ms 0.000 ms
org.apache.tomcat.util.net.NioBlockingSelector$BlockPoller.run () 0.000 ms (0%) 0.000 ms 0.000 ms 0.000 ms
org.apache.tomcat.util.net.NioEndpoint$Acceptor.run () 0.000 ms (0%) 0.000 ms 0.000 ms 0.000 ms
org.apache.coyote.AbstractProtocol$AsyncTimeout.run () 0.000 ms (0%) 0.000 ms 0.000 ms 0.000 ms
org.apache.catalina.core.ContainerBase$ContainerBackgroundProcessor.run () 0.000 ms (0%) 0.000 ms 0.000 ms 0.000 ms

PERFORMANCE TESTING

Performance Testing Metrics

- CPU utilization
- Memory utilization
- Network utilization
- Operating system limitations
- Disk Usage

Bottlenecks

- Long load time
- Poor response time
- Poor scalability

Types of Performance Testing

- Load Testing
  - Checks the applications ability to perform under anticipated user loads.
  - Capacity planning
- Stress Testing
  - Where does system breaks, under extreme workloads
- Spike Testing
  - Test the reaction under sudden Spike
- Volume Testing
  - Large volume of data in database
- Scalability Testing
  - Whether scaling up solves the load/volume problem

Tools

- WebLoad – Mostly for load testing
- Apache Jmeter – load testing of web applications
- Loadtest - **loadtest** "loadtest [-n requests] [-c concurrency] URL"
- NeoLoad – performance testing platform
- HP LoadRunner – simulate users and perform performance testing

DATABASE TESTING

Database Testing

- Checking the schema, tables, triggers, etc. of the database under test
- Creating complex queries to load/stress test the database and check its responsiveness
- Checks data integrity and consistency

Types of Database Testing

Structural Testing

- **Schema testing – Mapping between back end and front end are same...** Scenario - if the developers want to change a table structure or delete it, the tester would want to ensure that all the Stored Procedures and Views that use that table are compatible with the particular change. Tools: DBUnit in Ant, SQL Server
- **Database Table, columns testing** – Data types, Field lengths, primary key, foreign keys, indexes
- **Server validation** – authorization, endpoint correctness
- Procedure/Trigger Testing – coding convention, covered all loops, update operations on data. Tools - LINQ

Types of Database Testing

Functional Testing - specified by the requirement specification needs to ensure most of those transactions and operations as performed by the end users are consistent with the requirement specifications.

- **Checking data integrity and consistency**
- Whether the field is mandatory while allowing NULL values on that field.
- Whether the length of each field is of sufficient size?
- Whether all similar fields have same names across tables?
- Whether there are any computed fields present in the Database?

Types of Database Testing

Load Testing – Why?

- User Transaction can potentially impact the performance of other transactions
- Include at least one editing transaction and one non-editing transaction during test, so that performance can be differentiated
- More important transactions should be definitely included
  - **Stress testing**

  Tools – JMeter, LoadRunner

Fitnesse

- Acceptance testing platform
- Wiki web server -- Write test in Wiki, executing which calls custom Fixtures (bridge between Wiki pages and SUT)
- Fixtures are written in Java, C#, etc.
- Execute a piece of business logic & pass the results back to Wiki
  - Two systems : SLIM and FIT. FIT is older is no longer actively developed
- fitnesse.jar – running this starts a Wiki web server
  - Add a Suite, add a Test, Write fixture and Test cases

https://dzone.com/refcardz/getting-started-fitnesse?chapter=1

Open Discussion
Q and A

Thank you!