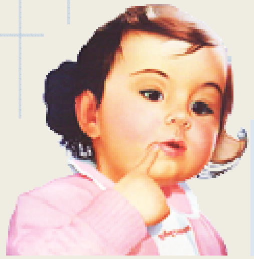
Copyright Notice

Welcome..

Prakash Badhe

Introduction To BDD with Cucumber in Java



Murphy’s Law

“If Anything that can go wrong, It will go wrong".

Software – What can go wrong?

IMPLEMENTATION DEFECT

- Program does not do what it is intended
- Fixed in development/QA

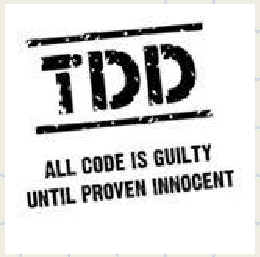
REQUIREMENTS DEFECT

- Program does what is intended
- Unstated, misunderstood or tacit requirements

TDD Introduction

Test Driven Development (TDD)

- Problems with TDD?
  - o Desired "behavior" of an system is specified in terms of code, not always the right way to capture behavior
  - o Other stakeholders in Software lifecycle find it hard to be involved in the whole process



50-60%

of issues identified by testers can be chalked down as

Requirements Defects

100-200%

more expensive to fix than other defects because

the code is already written

Benefits of TDD

- ➢Unit test proves that the code actually works
- ➢It drives the design of the program
- ➢Refactoring allow to improve the design of the code
- ➢Low Level regression test suite
- ➢Test first approach reduces the cost of the bugs.

Issues in TDD

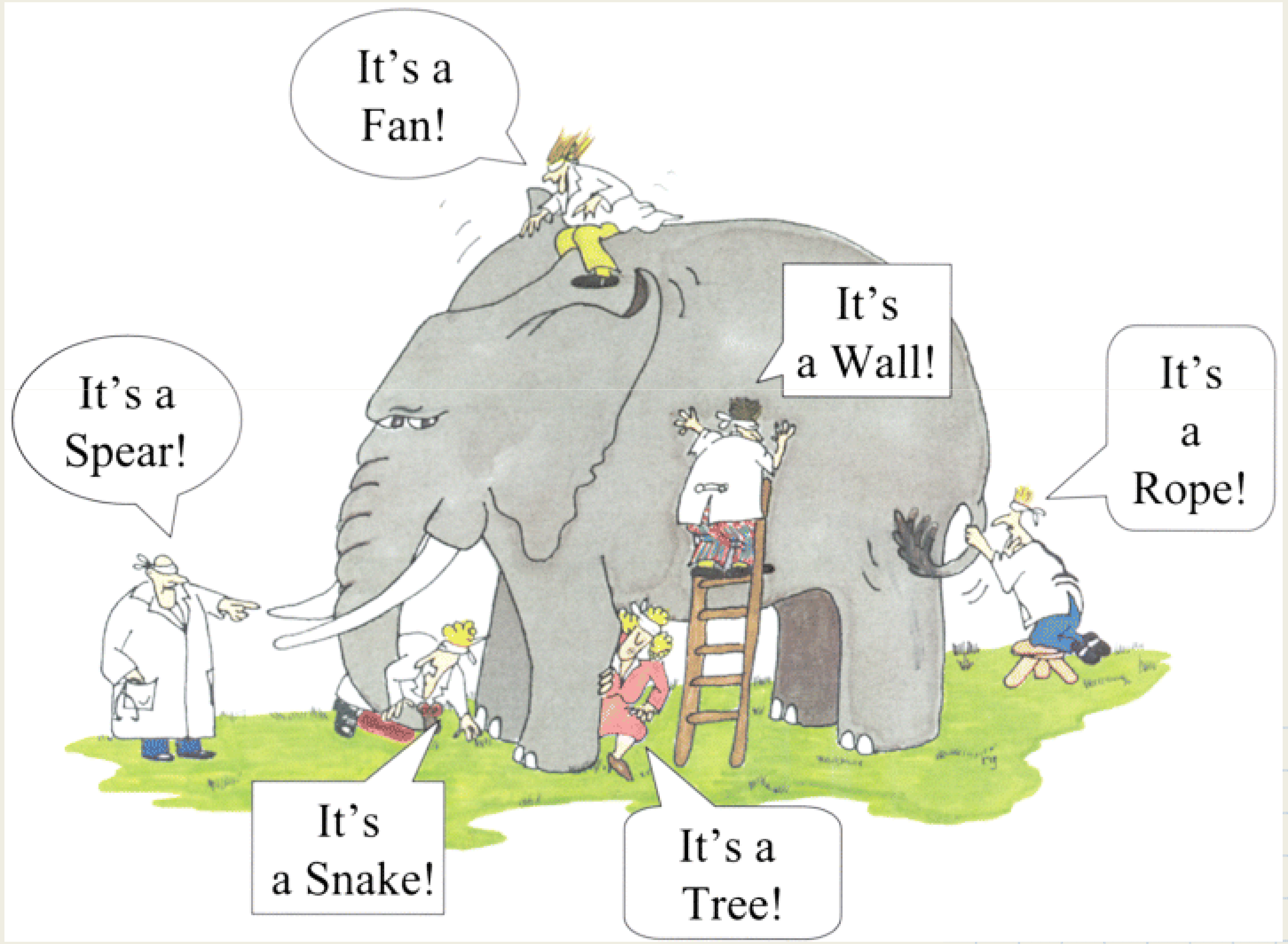
1. Developer can consider it as a waste of time
2. The test can be targeted on verification of classes and methods and not on what the code really should do
3. Test become part of the maintenance overhead of a project
4. Rewrite the test when requirements change
5. The tester or developers writes the unit test cases in specific programming language, not known b y the business users and
6. Hence reduces the collaboration.
7. Its white box testing. The internal logic must be known, to implement the test.

TDD Extensions

1. Acceptance TDD : ATDD
2. Behavior Driven Development : BDD

Behavior Driven Development

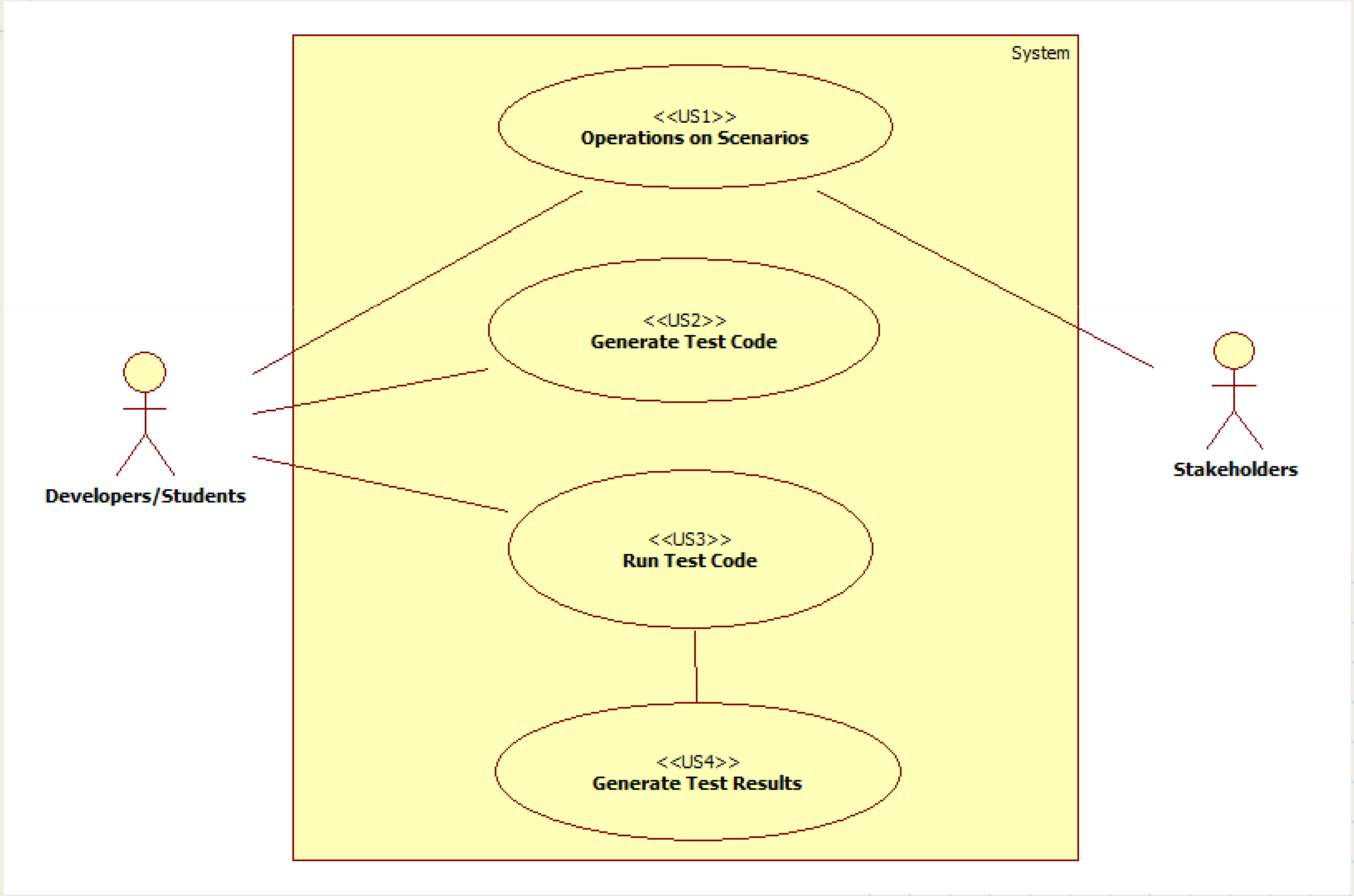
Different Perceptions



BDD Goals

- Developers, who want to practice TDD using a simplified framework
- To follow a test-first approach towards software development.
- Stakeholders like customers who may have not followed a test-first approach, can use the system to test application code.

Collaboration in BDD



Features of BDD

1. Shifting from thinking in “tests” to thinking in “behavior”
2. Collaboration between Business stakeholders, Business Analysts, QA Team and developers
3. Ubiquitous language, it is easy to describe.
4. Driven by Business Value.
5. Extends Test Driven Development (TDD) by utilizing natural language that non technical stakeholders can understand
6. BDD frameworks acting a “bridge” between Business users & Programming Language.
7. BDD is can be utilized for ***Unit level*** test cases and for ***UI level*** testing also.

BDD Test language

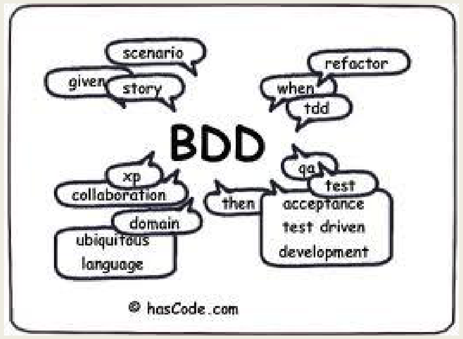
✓Tests are written in plain descriptive English type grammar
✓Tests are explained as behavior of application and are more user focused
✓Using examples to clarify requirements.
✓The BDD brings in the need to have a language which can define the test cases in an understandable format.

BDD Introduction

Behaviour Driven Development (BDD)

- BDD is TDD done right!!!
- BDD was conceived by Dan North in 2003
- BDD uses natural language to describe the "desired behavior" of the system, that can be understood by both the developer and the customer

- Demo of an existing BDD application using Cucumber which is a BDD framework .



How does BDD helps..?

- All stakeholders involved in the discussion drives out clear and concise specifications centred around **business value**.
- Write specifications in **natural language** so everyone is on the same page
- Turn the specification into **acceptance tests** that guarantees the software does as it is intended
- Use those tests as the software evolves to guarantee that it **continues to work as intended.**
- Specifications are **documentation** – each scenario describes how the system is being used

BDD Frameworks

- There are very few published studies on BDD, most of which take a relatively narrow view of BDD and only treat it as a specific technique of software development

- There are tools which use UML, XML schema etc. as input and generate an template for the source code.

- Various Frameworks that make BDD accessible
  - JBehave
  - Cucumber
  - JDave

Cucumber

Cucumber is a testing framework which supports ***Behavior Driven Development (BDD).***
It lets us define application behavior in plain meaningful English text using a simple grammar defined by a language called ***Gherkin***.
Cucumber itself is written in ***Ruby***, but it is used to “test” code written in *Ruby* or other languages including but not limited to *Java*, *C#* and *Python.*

Cucumber -- BDD

- Cucumber
  - Designed with Ruby
  - Born 2007-2008
- Cuke4Duke
  - First seen in 2008
  - Jruby
- Cucumber-JVM
  - Released April 2012
  - Java, Groovy, Clojure, Scala, Ioke, Jython, Rhino (JavaScript) and Jruby
- Cucumber for C# .Net
- Cucumber for JavaScript

Feature file in BDD

1. A ***Feature File*** is an entry point to the *Cucumber BDD* tests.
2. This is a file where users will describe the tests in Descriptive language (Like English).
3. A feature file can contain a scenario or can contain many scenarios in a single feature file but it usually contains a list of scenarios.
4. Each feature is represented as a plain text file.
5. Feature files must have the “.feature” extension

Scenario's in Cucumber BDD

- The scenarios are group of behaviour test situations.
- Scenarios are organized together into features
- Each feature can contain many scenarios.

Scenarios - Given When Then

Given an owner with a registered pet called “Sly“
When the owner registers a new pet called “Sly“
Then the user should be warned that the pet already exists

Step Definitions

- Step is the translation of feature scenarios to application test language.
- A single Method representing a Step from a Feature file
- The Method is annotated with information about the step it represents
- Cucumber generates “Code Snippet's” for Step Definition it can’t find

Step Annotations

import cucumber.annotation.en.*;

Each Gherkin step keyword has a Annotation

```
@When("^we want to stop traffic$")
public void methodName() { … }
```

JUnit Test Runner

```
import org.junit.runner.RunWith;
import cucumber.junit.Cucumber;

@RunWith(Cucumber.class)
public class TestRunner {
}
```

JUnit Test Runner 2

- Body of the class should be empty
  - No @Test annotated methods
  - No @Before or @After methods
- Arguments can be provided with an additional annotation

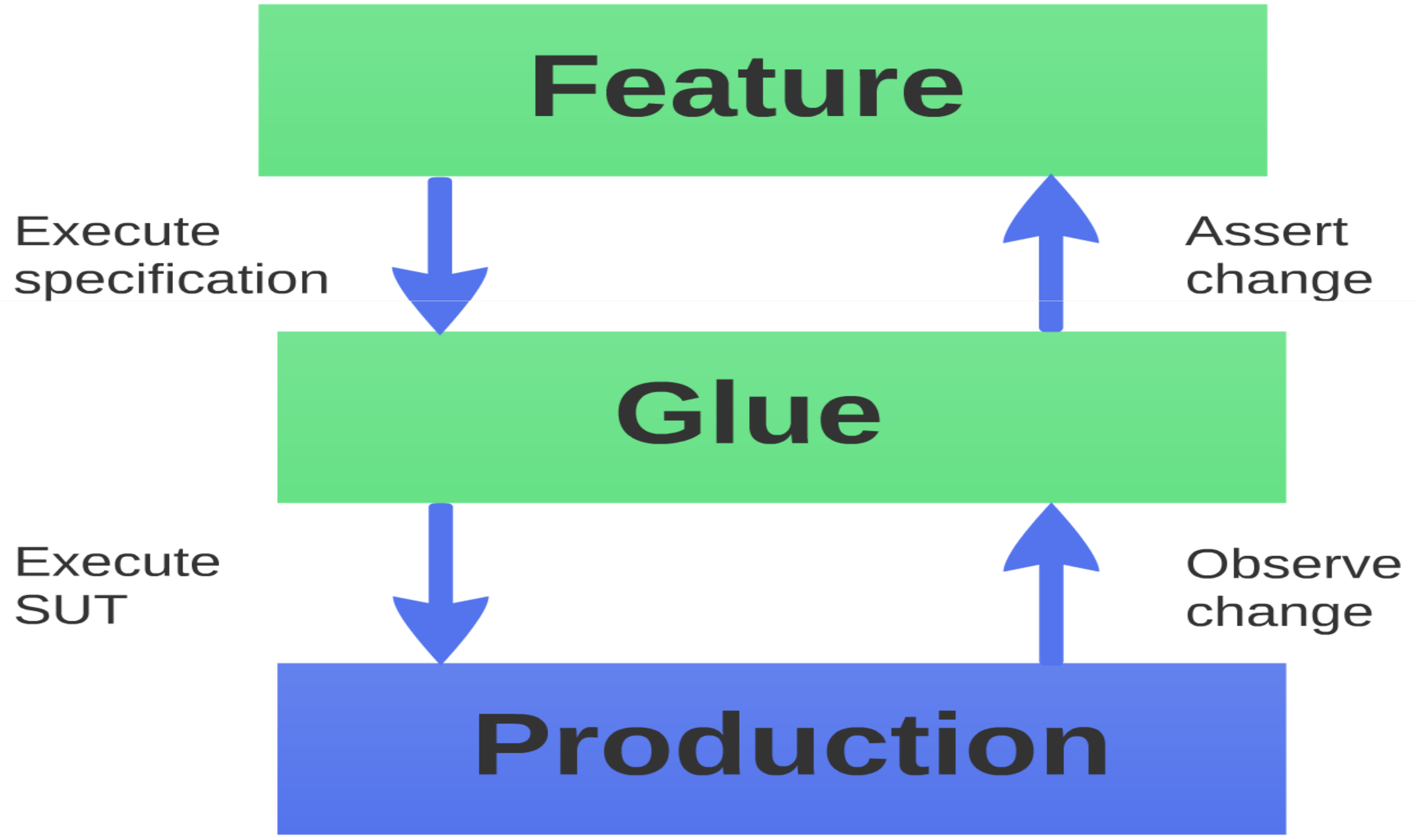
Passing Or Failing state

- Each step has it own result state
  - Passed
  - Pending
  - Skipped
  - Failed
- Individual step result state combined determine the Scenarios success or failure
- Steps can be failed on exceptions or JUnit Assertion failures

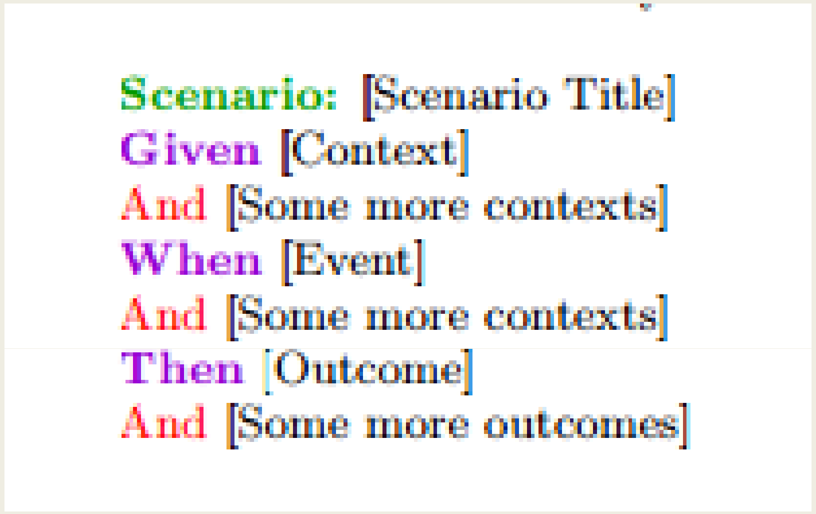
Problem Statement

- In existing BDD frameworks, developer needs to manually write the glue code which maps "behaviors" to the implementation code
- This mapping involves writing code for interpreting behaviors in terms of test-cases
- Writing this glue code is tedious and error prone, if the written test code does not depict the specified behavior
- **Can this intermediate step of writing glue code be automated?**

Cucumber Process



Scenario: specification


Scenario: [Scenario Title]
Given [Context]
And [Some more contexts]
When [Event]
And [Some more contexts]
Then [Outcome]
And [Some more outcomes]

- **GIVEN**, an initial context
- **WHEN**, occurrence of an event
- **THEN**, expected outcome
- Parameters in Quotes, **" "**
- Connectives - **And** and **Not** (only in then condition)

Scenario: example

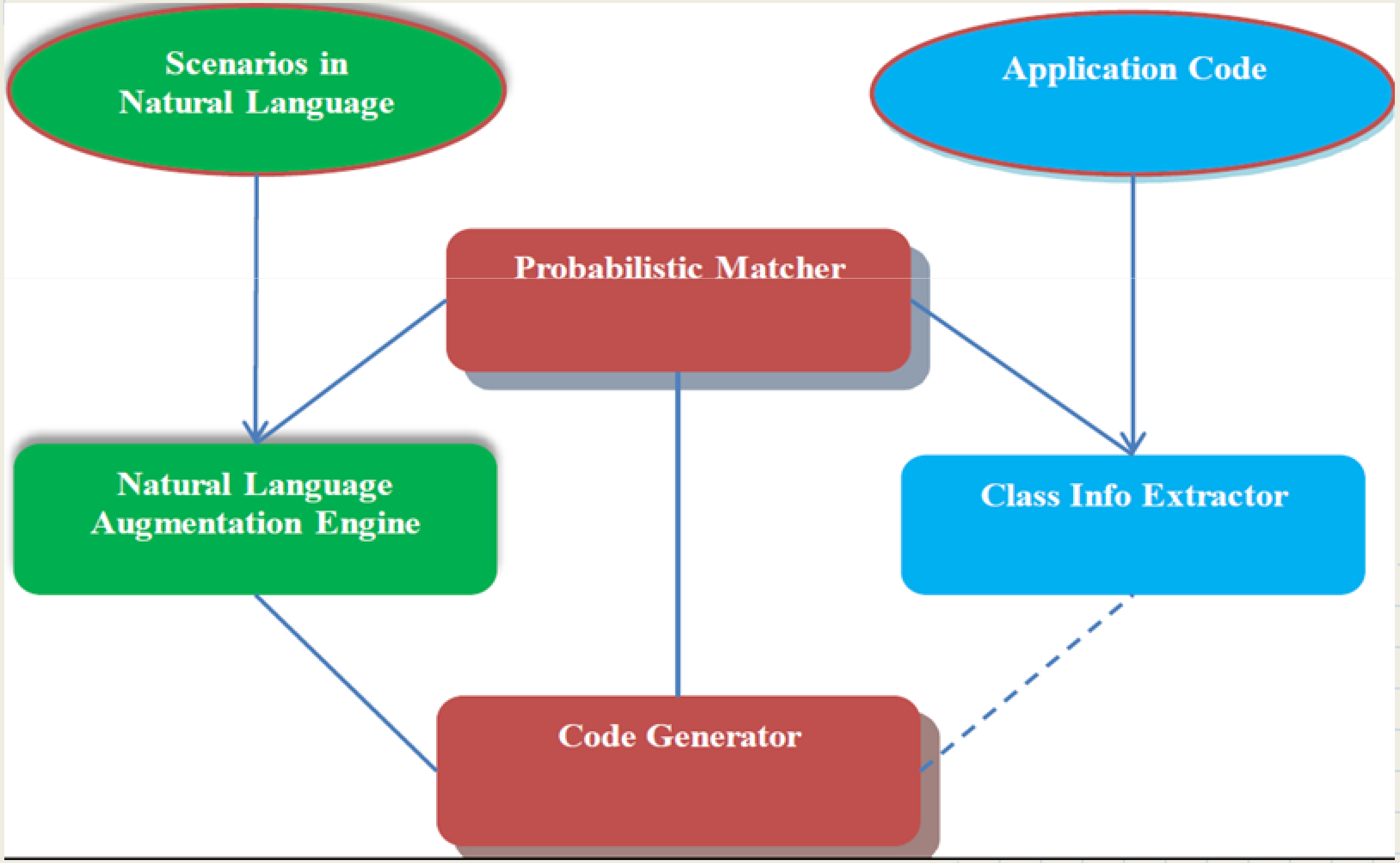
Scenario 1: Account Deposit

Given a bank account with a balance of "**100**" $

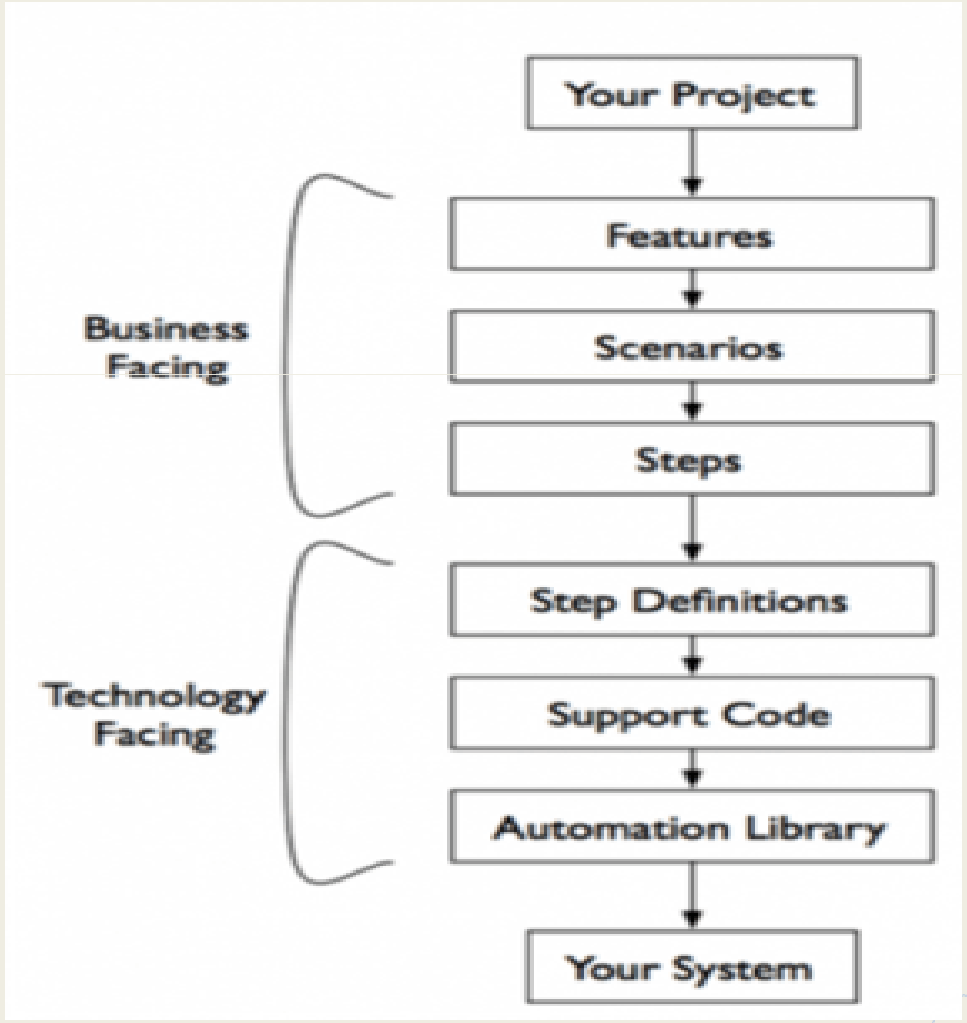
When a customer deposits "**20**" $

Then the balance of the account should be "**120**" $

Architecture



Cucumber Flow



Regex for the Match

Cucumber uses Regular Expressions to match steps to definitions.

```
@When("^we want to stop traffic$")
```

^	Matches the starting position
$	Matches the ending position

Capturing Arguments

```
@Given("^an owner with a pet called
\"Sly\"$")
public void an_owner_with_a_pet_called() {
      …
}

@Given("^an owner with a pet called
\"([^\"]*)\"$")
public void an_owner_with_a_pet_called(String
name) {
      …
}
```

Capturing Arguments 2

- The pair of brackets is a capture group
- Everything captured is passed to method parameter
- Cucumber will automatically convert to the method parameter type
- Equal number of method parameters and capture groups

Type Conversion

- If the captured string is not in the correct format for type conversion an error will be thrown
- Use regular expressions to capture the correct format string for the type

Note: Escape regular expression characters with backslash

Typical Capture Groups

`(.+)` Good for capturing strings and dates

`\"([^\"]*)\"` Captures a string within Double quotes

`(\\d)` Capture decimal numbers only
`(\\d+\\.\\d+)` Capture floating point numbers
`(true|false)` Captures a Boolean value

Date Formats

```
@Given ("^today is (.+)$")
public void today_is(Date date) {
        ….
 }
```

Dates are recognized if it’s a format from the DateFormat class

Date Formats

SHORT is completely numeric, such as 12.13.52 or 3:30pm
MEDIUM is longer, such as Jan 12, 1952
LONG is longer, such as January 12, 1952 or 3:30:32pm
FULL is pretty completely specified, such as Tuesday, April 12, 1952 AD or 3:30:42pm PST.

Date Format Usage

```
import cucumber.DateFormat;

@Given ("^today is (.+)$")
public void today_is(
     @DateFormat ("yyyy-MM-dd") Date date) {
       ....
 }
```

Tag use cases

- Grouping Scenarios together
  - Identify slow running test that shouldn’t be run frequently
  - Identify tests that require a particular library or server

Running Scenarios with a Tag

```
@Cucumber.Options(
    tags = {"@WebDriver"}, ... )
```

Running Scenarios without Tag

Cucumber can exclude Scenarios with a particular tag by inserting the tilde character before the tag

For the following command will run all Scenarios without the WebDriver tag

```
@Cucumber.Options(
        tags = {"~@WebDriver"}, ... )
```

Tag expressions – Logical OR

Separate a list of tags by commas for a Logical OR tag expression

```
@Cucumber.Options(
        tags = {"@WebDriver,@email"}, … )
```

Tag expressions – Logical AND

Specifying multiple tag arguments creates a logical AND between each tag expression.

```
@Cucumber.Options(
        tags = {“@WebDriver”, ”@email”}, …
)
```

Tag expressions – AND OR

Run scenarios that are tagged with @WebDriver or @email but not @slow

```
@Cucumber.Options(
        tags = {"@WebDriver,@email", "~@slow"}, … )
```