Introduction to Software Testing
(*2nd edition*)
Chapter 4

Putting Testing First

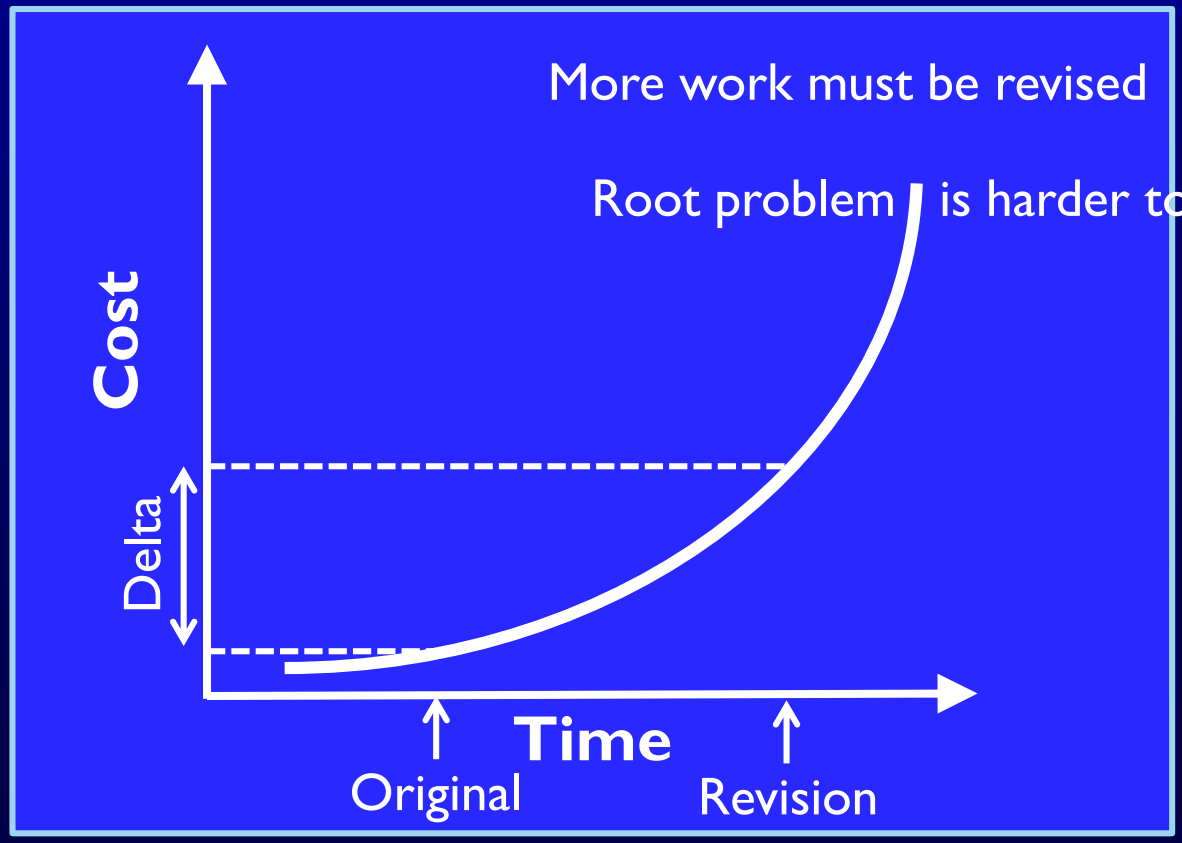
Paul Ammann & Jeff Offutt

http://www.cs.gmu.edu/~offutt/softwaretest/

August 2014

The Increased Emphasis on Testing

- Philosophy of traditional software development methods
  - Upfront analysis
  - Extensive modeling
  - Reveal problems as early as possible



Traditional Assumptions

1. Modeling and analysis can identify potential problems early in development

2. Savings implied by the cost-of-change curve justify the cost of modeling and analysis over the life of the project

- These are true if requirements are always complete and current
- But those annoying customers keep changing their minds!
  - Humans are naturally good at approximating
  - But pretty bad at perfecting
- These two assumptions have made software engineering frustrating and difficult for decades

Thus, agile methods …

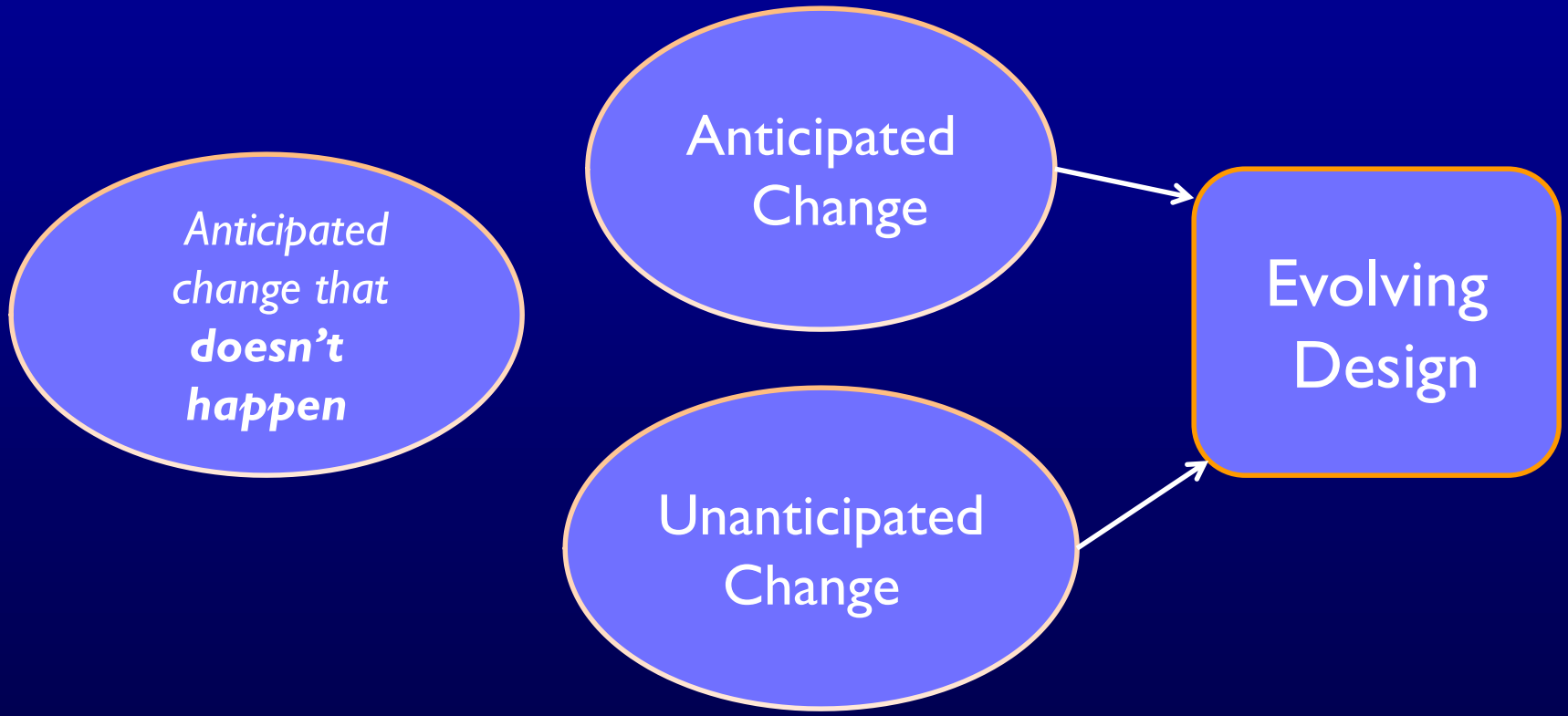
Why Be Agile ?

- Agile methods start by recognizing that neither assumption is valid for many current software projects
  - Software engineers are not good at developing requirements
  - We do not anticipate many changes
  - Many of the changes we do anticipate are not needed
- Requirements (and other "non-executable artifacts") tend to go out of date very quickly
  - We seldom take time to update them
  - Many current software projects change continuously
- Agile methods expect software to start small and evolve over time
  - Embraces software evolution instead of fighting it

Supporting Evolutionary Design

Traditional design advice says to anticipate changes

Designers often anticipate changes that don't happen

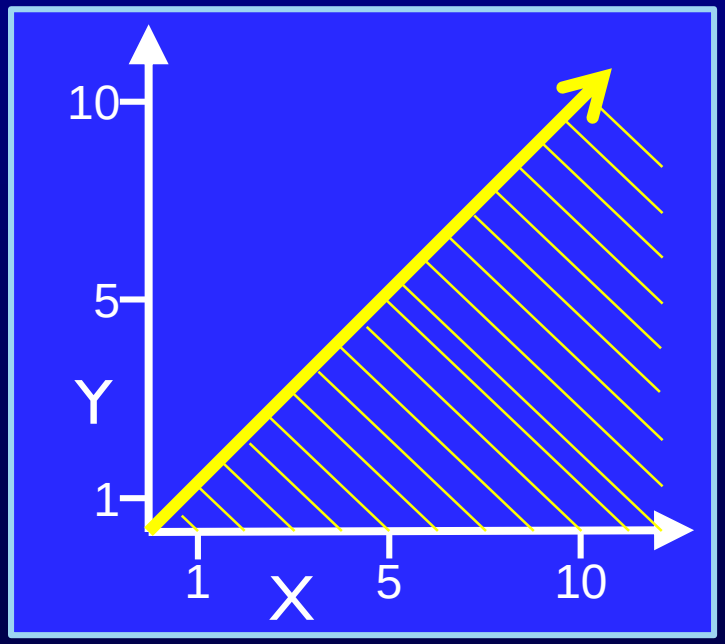


Both anticipated and unanticipated changes affect design

The Test Harness as Guardian (4.2)

What is Correctness ?

Traditional Correctness
(Universal)
$\forall$ x,y, x ≥ y



Agile Correctness
(Existential)

{ (1, 1) → T
(1, 0) → T
(0, 1) → F
(10, 5) → T
(10, 12) → F }

A Limited View of Correctness

- In traditional methods, we try to define all correct behavior completely, at the beginning
  - What is correctness?
  - Does "correctness" mean anything in large engineering products?
  - People are VERY BAD at completely defining correctness
- In agile methods, we redefine correctness to be relative to a specific set of tests
  - If the software behaves correctly on the tests, it is "correct"
  - Instead of defining all behaviors, we demonstrate some behaviors
  - Mathematicians may be disappointed at the lack of completeness

But software engineers ain't mathematicians!

Test Harnesses Verify Correctness

A *test harness* runs all automated tests efficiently and reports results to the developers

- Tests must be automated
  - Test automation is a prerequisite to test driven development (TDD)
- Every test must include a test oracle that can evaluate whether that test executed correctly
- The tests replace the requirements???
- Tests must be high quality and must run quickly
- We run tests every time we make a change to the software (regression tests!)

Continuous Integration

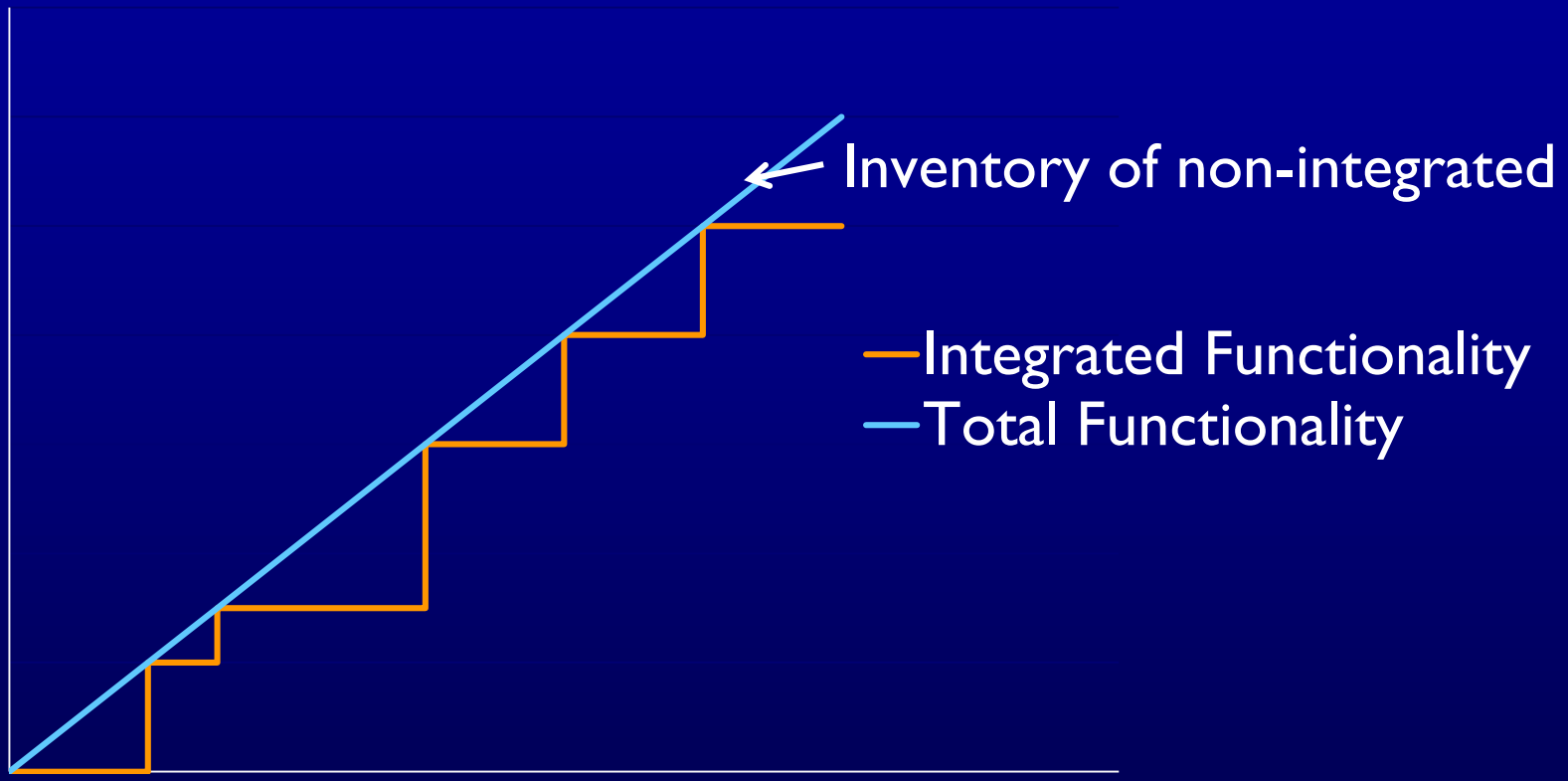
- Agile methods work best when the current version of the software can be run against all tests at any time

> A *continuous integration server* rebuilds the system, returns, and reverifies tests whenever *any* update is checked into the repository

- Mistakes are caught earlier
- Other developers are aware of changes early
- The rebuild and reverify must happen as soon as possible
  - Thus, tests need to execute quickly

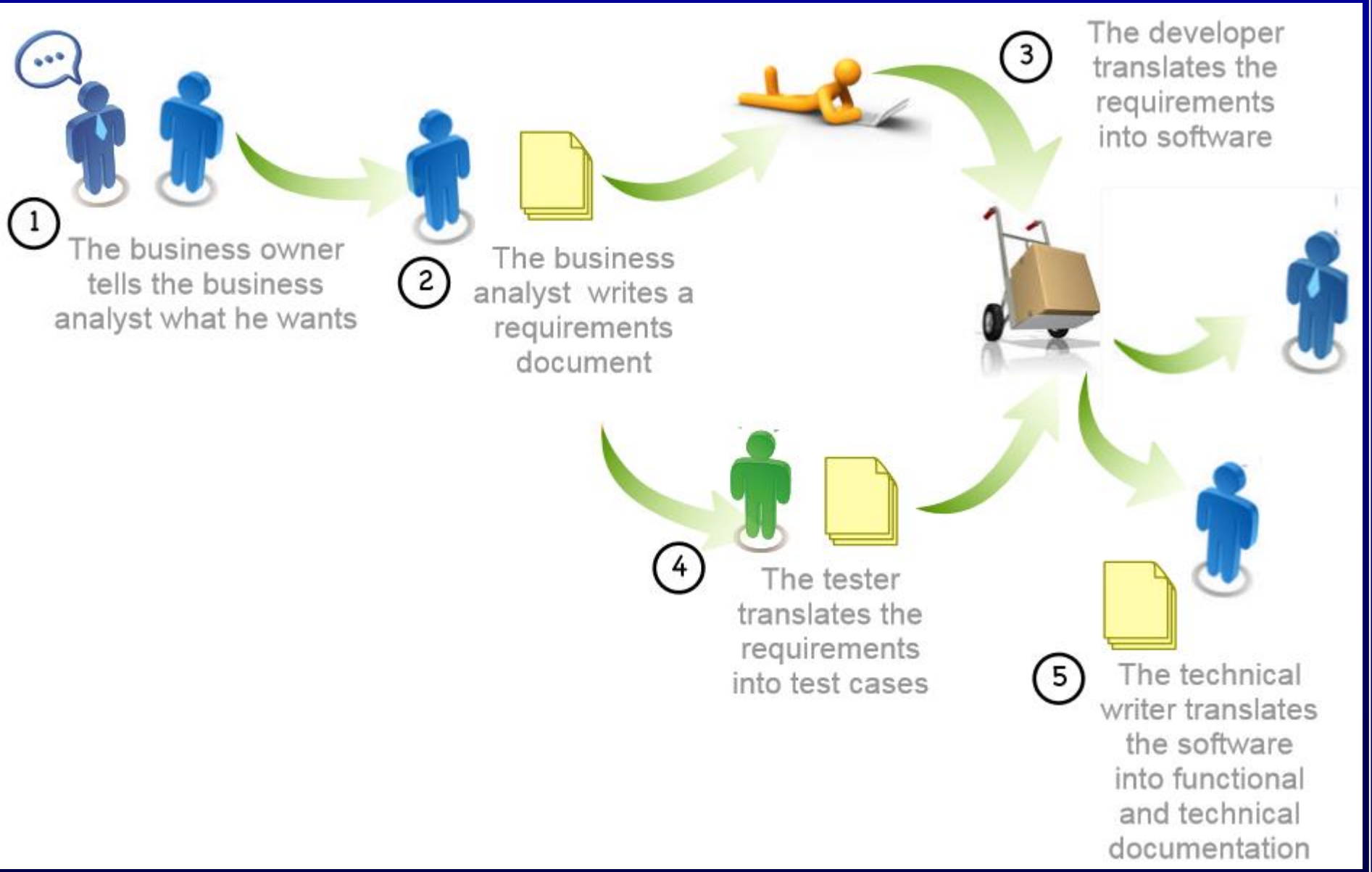
> A *continuous integration server* doesn't just run tests, it decides if a modified system is still correct

Continuous Integration Reduces Risk



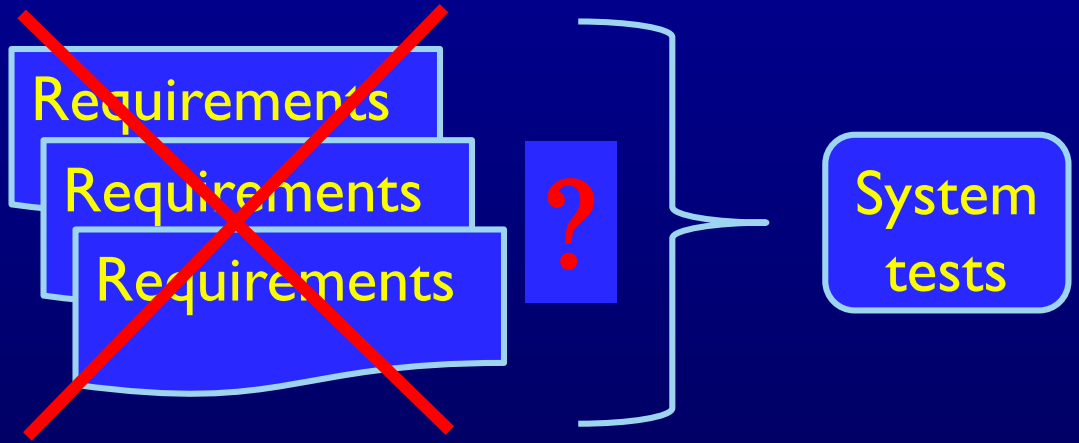
Non-integrated functionality is dangerous!

A Traditional Development Process



System Tests in Agile Methods

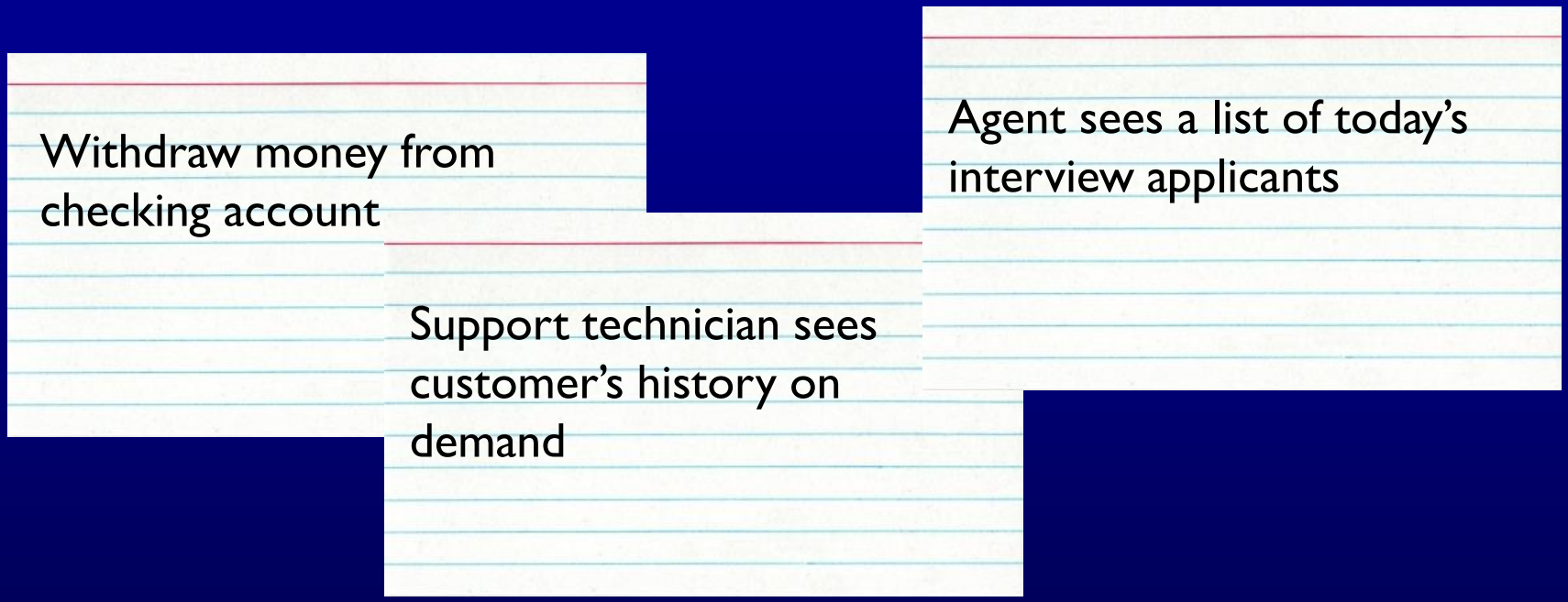
Traditional testers often design system tests from requirements



But … what if there are no traditional requirements documents ?

User Stories

A *user story* is a few sentences that captures what a user will do with the software



- In the language of the end user
- Usually small in scale with few details
- Not archived

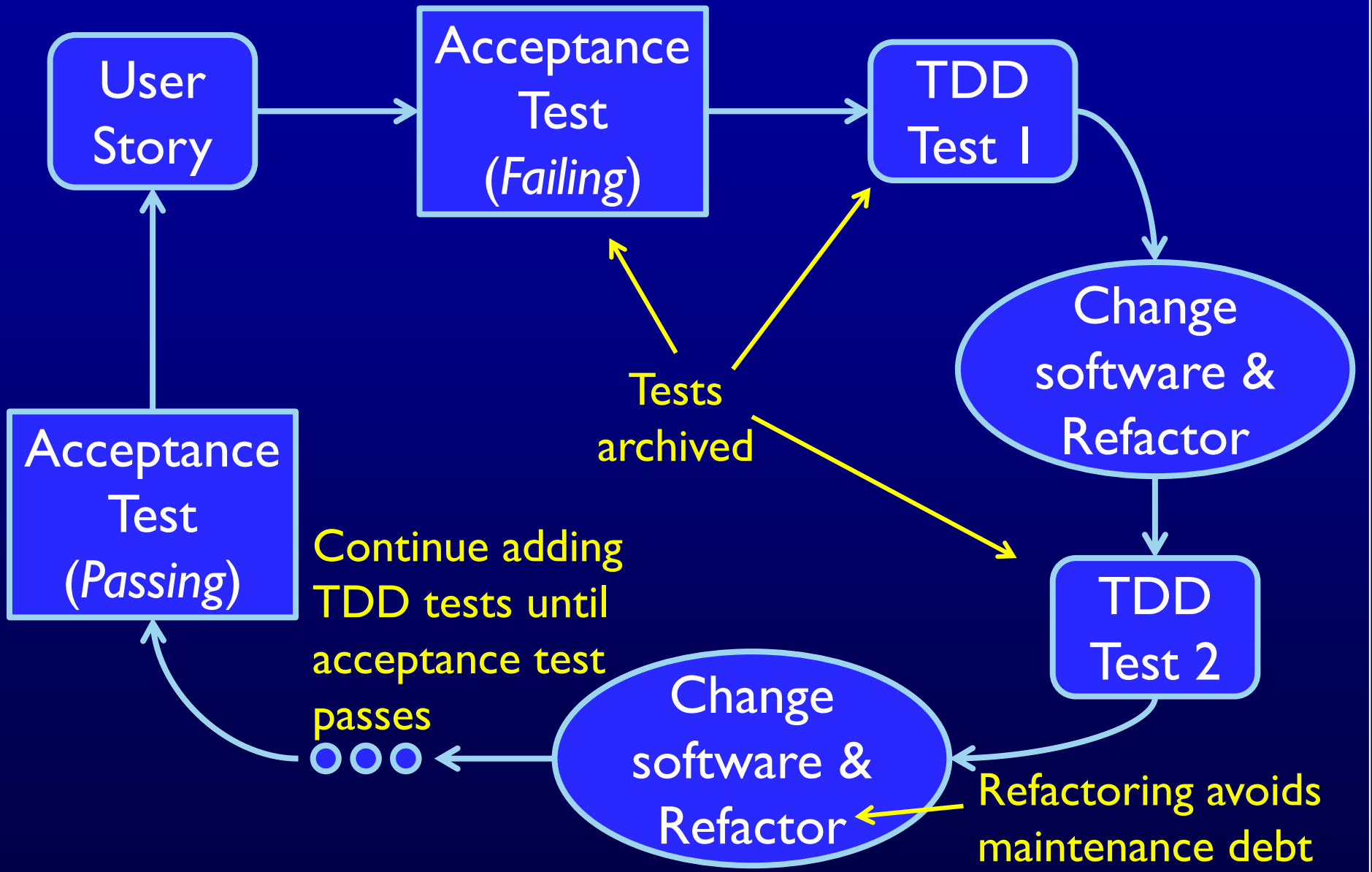
User Stories

- A user story is an informal, natural language description of one or more features of a software system. It is a promise for a conversation between the stakeholders (customers, users, developers, testers, etc.)
- User stories may follow one of several formats or templates
  - **As a** < type of user >, **I want** < some goal > **so that** < some reason >
  - **As a** <role> **I can** <capability>, **so that** <receive benefit>
- Examples:
  - **As a** website user,
    I want to able to search on the webpage,
    So that I can find necessary information
  - **As** a user, **I want** to be able to recover the password to my account, **so that** I will be able to access my account in case I forgot the password
  - **As a** user, **I want** to be able to request the cash from my account in ATM **so that** I will be able to receive the money from my account quickly and in different places

INVEST Criteria

- Good user stories always fit the INVEST set of criteria
  - **I**ndependent – they can be developed in any sequence and changes to one User Story don't affect the others.
  - **N**egotiable – it's up for the team to decide how to implement them; there is no rigidly fixed workflow.
  - **V**aluable – each User Story delivers a detached unit of value to end users.
  - **E**stimable – it's quite easy to guess how much time the development of a User Story will take.
  - **S**mall – it should go through the whole cycle (designing, coding, testing) during one sprint.
  - **T**estable – there should be clear acceptance criteria to check whether a User Story is implemented appropriately.

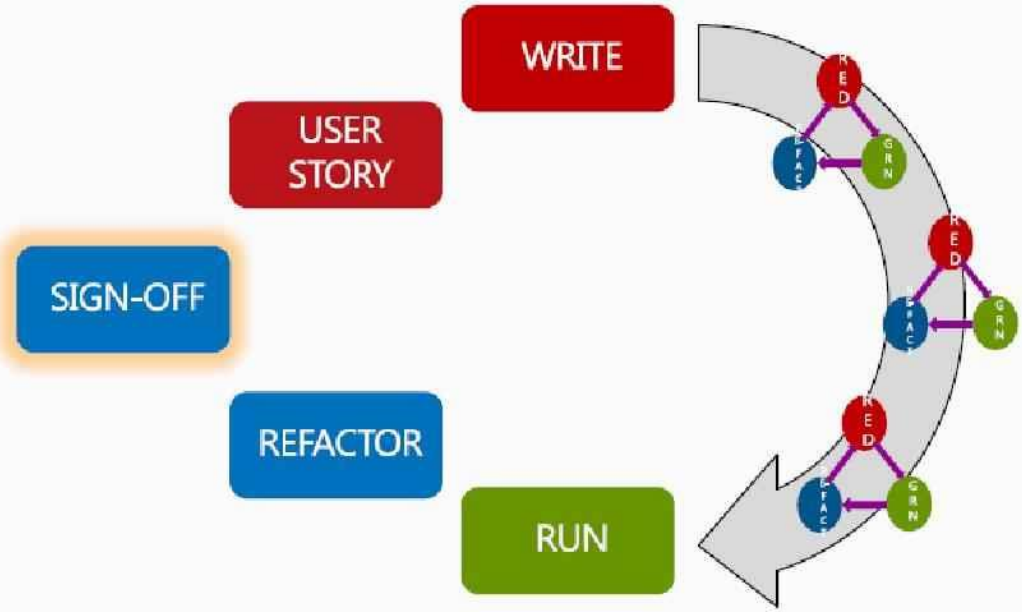
Acceptance Tests in Agile Methods

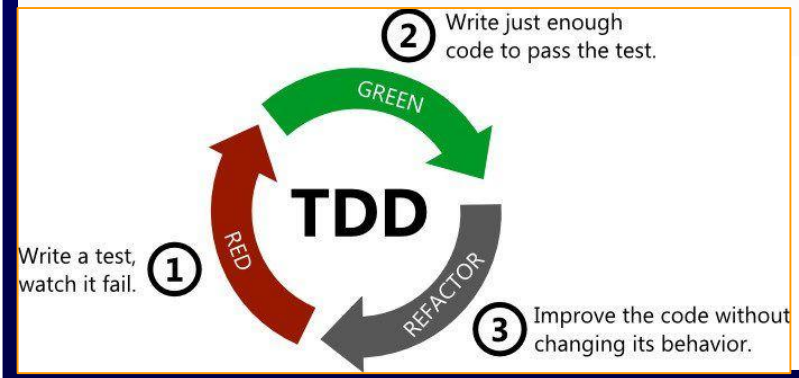


Acceptance Test Driven Development (ATDD)

(Acceptance Test Driven Development)

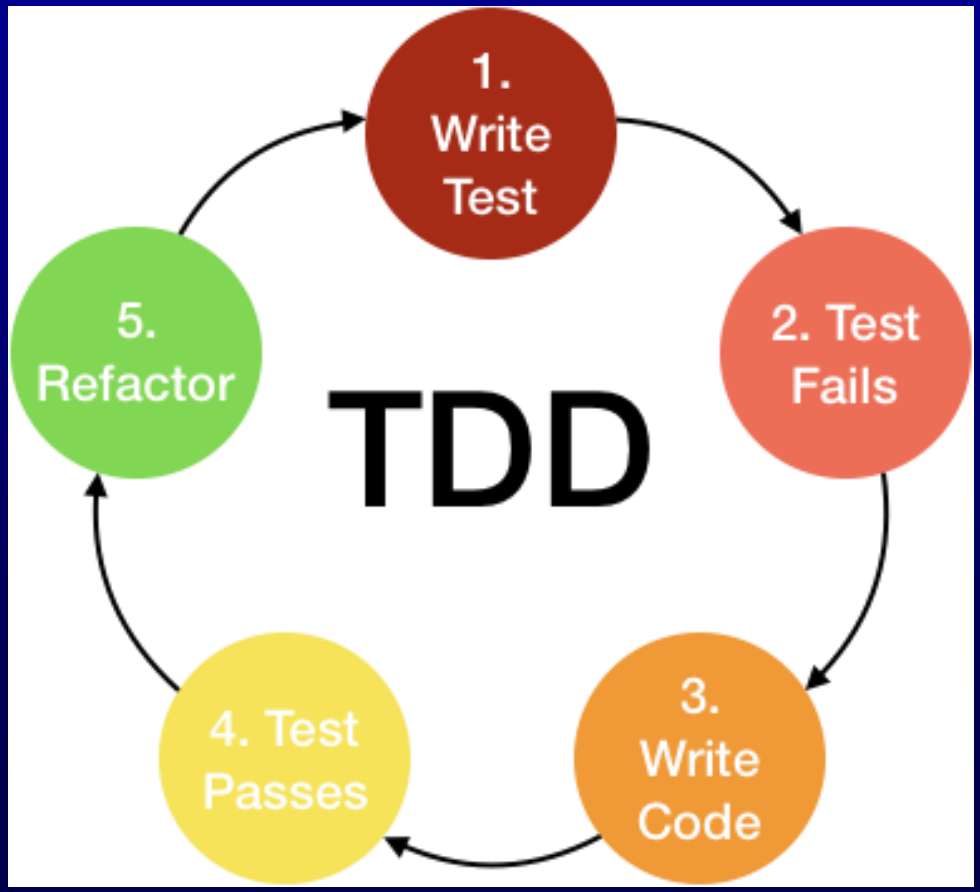
- Select User story
- Write Acceptance Test
- Implement User Story
- Run Acceptance Test
- (Refactor)
- Get Sign-Off

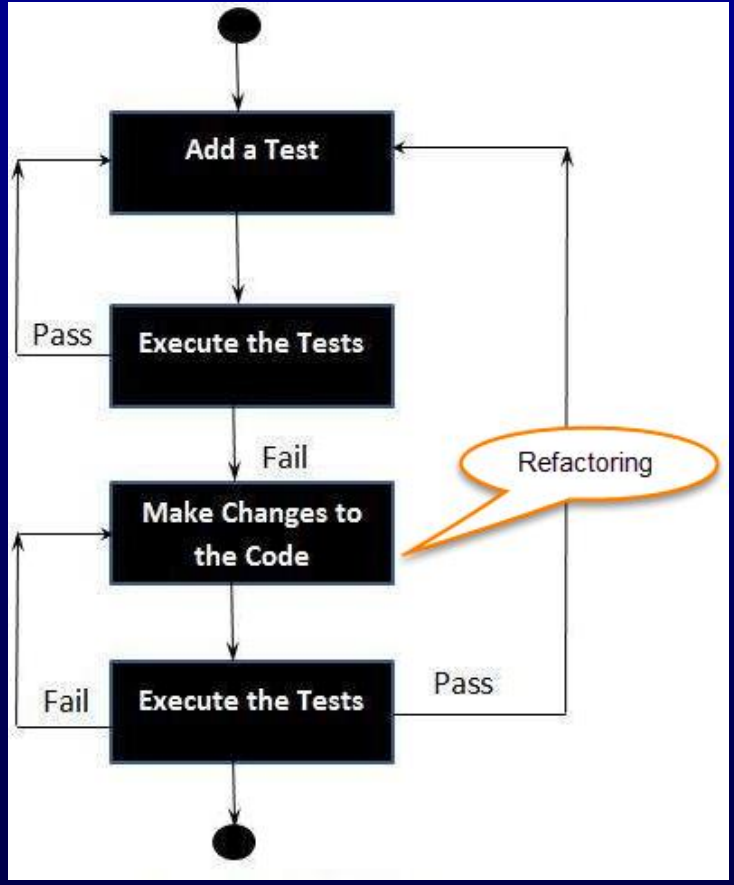




Test-Driven Development (TDD)

- TDD is a software **development** process that requirements are turned into very specific test cases, then the code is improved so that the tests pass, and repeat





Adding Tests to Existing Systems

- Most of today's software is legacy
  - No legacy tests
  - Legacy requirements hopelessly outdated
  - Designs, if they were ever written down, lost
- Companies sometimes choose not to change software out of fear of failure

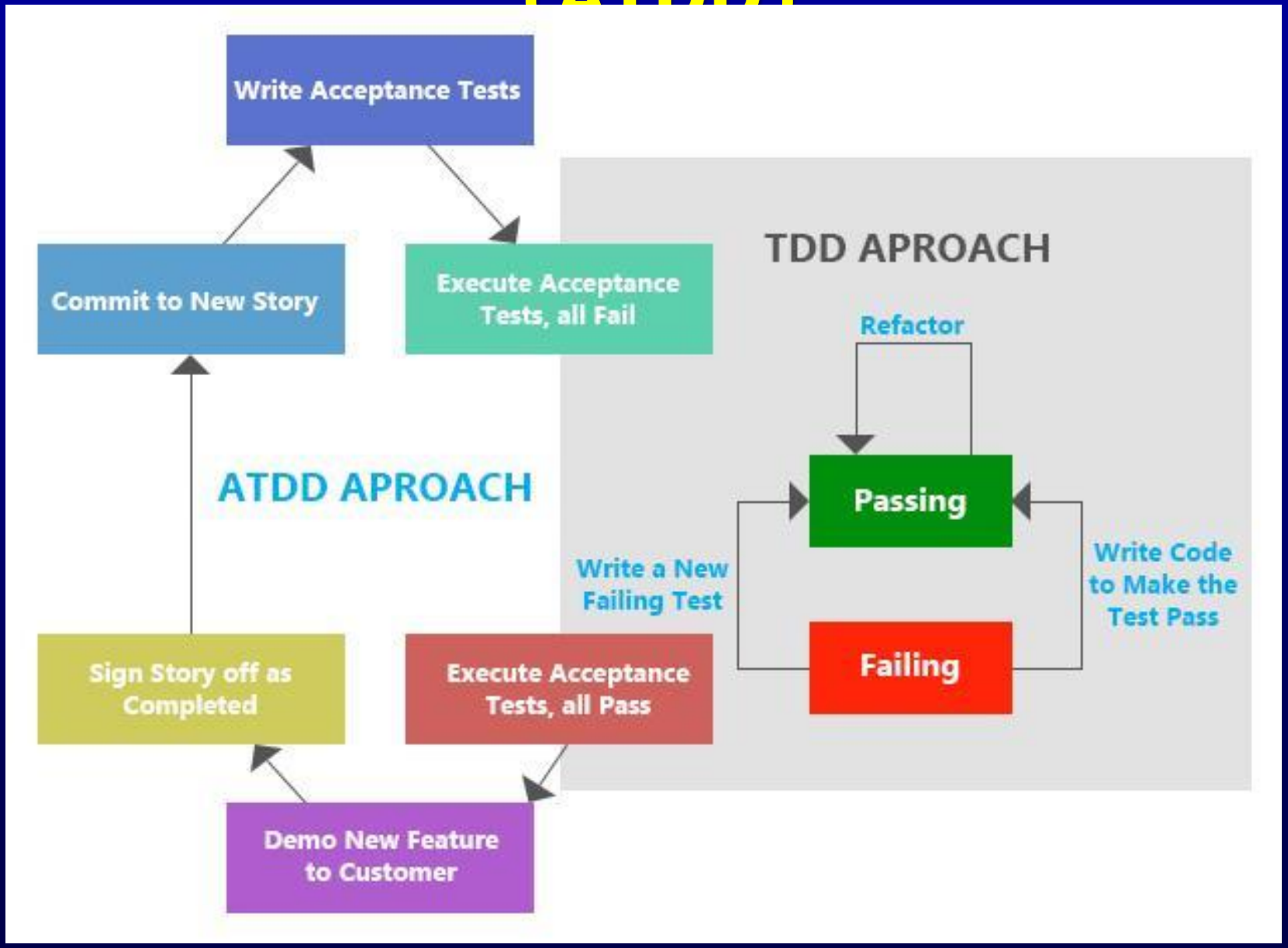
How to apply TDD to legacy software with no tests?

- Create an entire new test set? — too expensive!
- Give up? — a mixed project is unmanageable

Incremental TDD

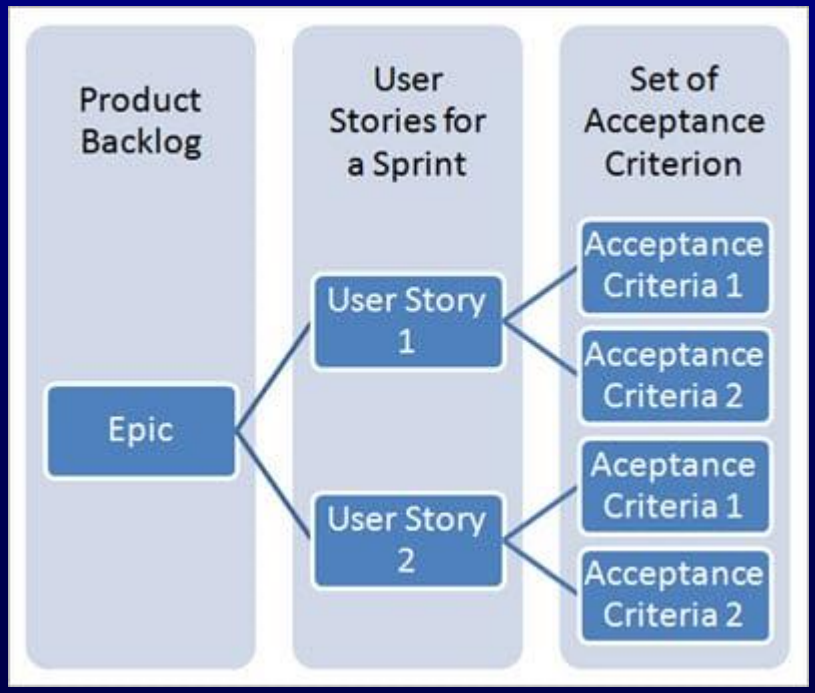
- When a change is made, add TDD tests for just that change
  - Refactor
- As the project proceeds, the collection of TDD tests continues to grow
- Eventually the software will have strong TDD tests

Acceptance Test Driven Development (ATDD)



Acceptance Criteria (AC)

- An acceptance criteria is a checklist or a set of satisfaction conditions that are used to confirm when a Story is completed
  - Concisely written criteria help development teams avoid ambiguity about a client's demands and prevent miscommunication
  - The acceptance criteria can become developer tasks
  - Written by three amigos: business analyst (product owner), developers, testers (QA) - Specification Workshop



User story: *As a user, I want to be able to register online, so that I can start shopping online.*

Acceptance criteria:

- ☐ User can only submit a form by filling in all required fields
- ☐ The email user provided must not be a free email
- ☐ Submission from same IP can only be made three times within 30 minutes
- ☐ User can only submit a form by filling in all required fields
- ☐ User will receive a notification email after successfully registration

How to Write Acceptance Criteria

- There are several types of acceptance criteria. The most popular are rules-oriented (in the form of a list), scenario-oriented (in the form of scenarios that illustrate each criterion), and custom formats
- The common template for describing acceptance criteria using a scenario-oriented approach is the *Given/When/Then* (GWT) format that is derived from behavior-driven development (BDD)
  - ***Given*** some precondition
  - ***When*** I do some action
  - ***Then*** I expect some result
- Each acceptance criteria written in this format has the following statements:
  - Scenario – the name for the behavior that will be described
  - Given – the beginning state of the scenario
  - When – specific action that the user makes
  - Then – the outcome of the action in "When"
  - And – used to continue any of three previous statements

Scenario-Oriented Acceptance Criteria

- User Story
  - As a logged-out user
  - I want to be able to sign in to a website
  - So that I can access my personal profile

- Scenario: System user signs in with valid credentials
  - Given: I'm a logged-out system user
  - *And:* I'm on the Sign-In page
  - When: I fill in the "Username" and "Password" fields with my authentication credentials
  - *And:* I click the Sign-In button
  - Then: the system signs me in

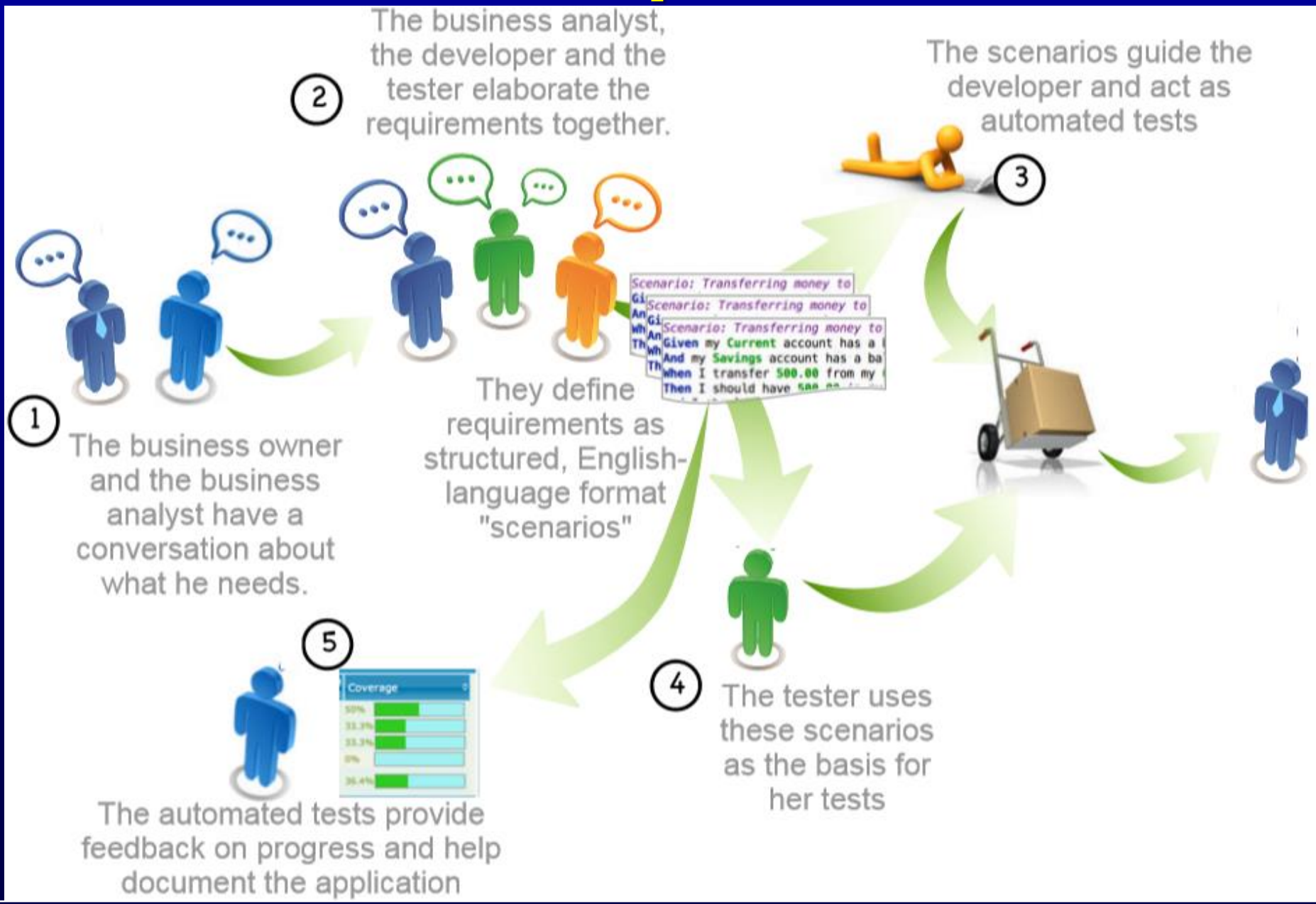
Rule-Oriented Acceptance Criteria

- User story
  - As a user, I want to use a search field to type a city, name, or street, so that I could find matching hotel options
- Basic search interface acceptance criteria
  - The search field is placed on the top bar
  - Search starts once the user clicks "Search"
  - The field contains a placeholder with a grey-colored text: "Where are you going?"
  - The placeholder disappears once the user starts typing
  - Search is performed if a user types in a city, hotel name, street, or all combined
  - Search is in English, French, German, and Ukrainian
  - The user can't type more than 200 symbols
  - The search doesn't support special symbols (characters). If the user has typed a special symbol, show the warning message: "Search input cannot contain special symbols."

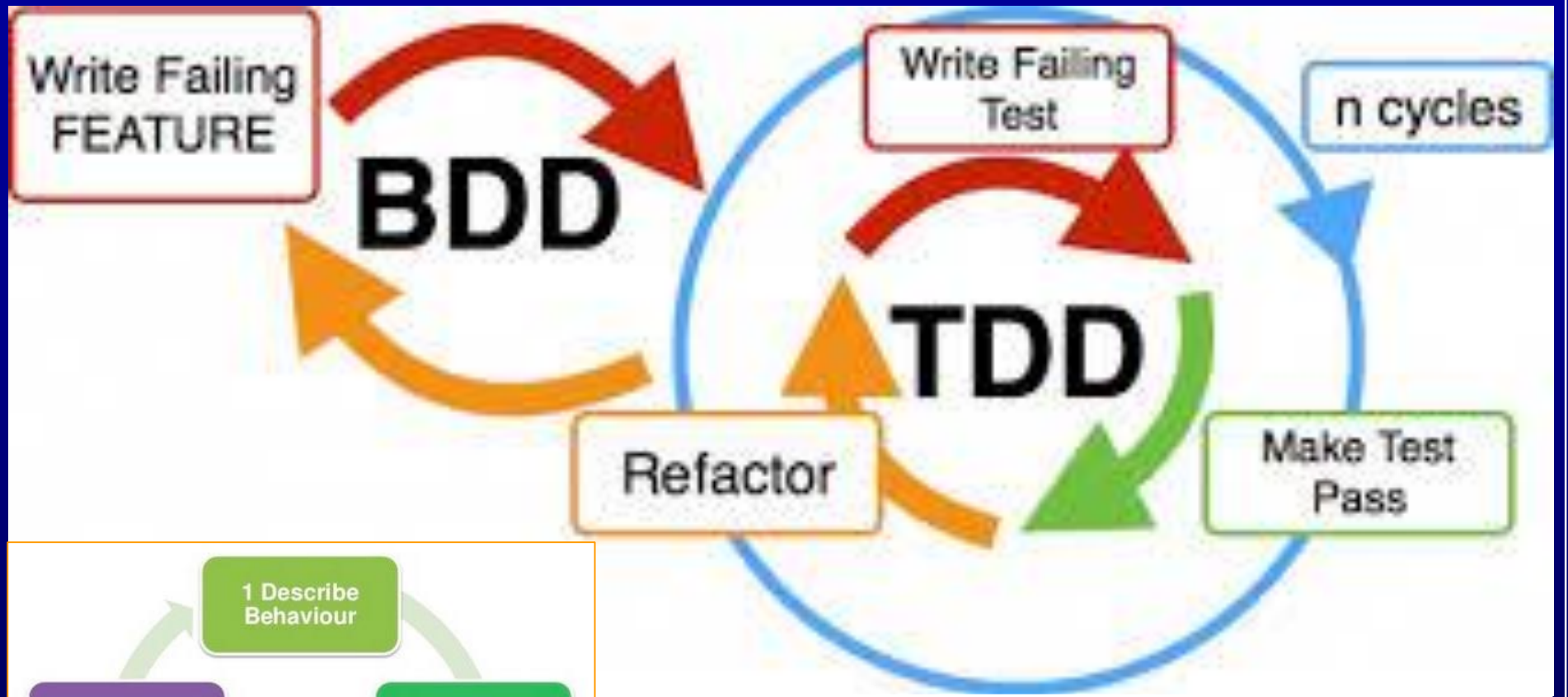
Behavior Driven Development (BDD)

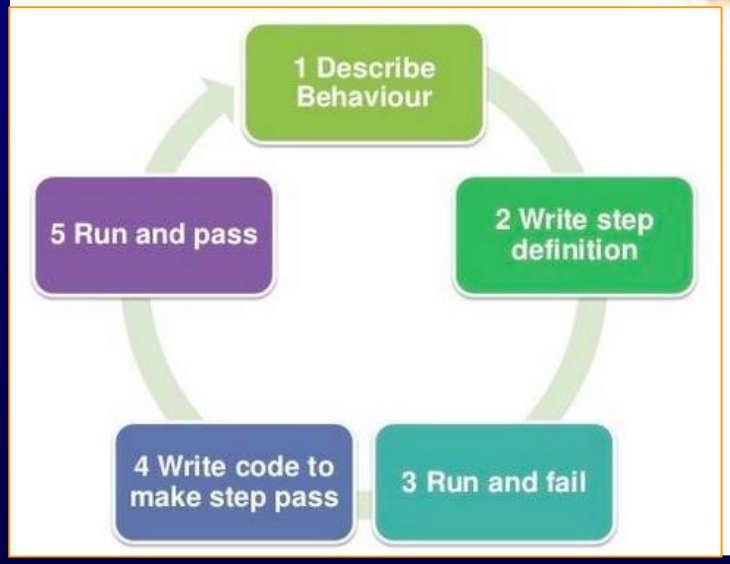
- **Behavior Driven Development (BDD)** is a synthesis and refinement of practices stemming from Test Driven Development (TDD) and Acceptance Test Driven Development (ATDD). BDD augments TDD and ATDD with the following tactics
  - Apply the "Five Why's" principle to each proposed user story, so that its purpose is clearly related to business outcomes
  - thinking "from the outside in", in other words implement only those behaviors which contribute most directly to these business outcomes, so as to minimize waste
  - describe behaviors in a single notation which is directly accessible to domain experts, testers and developers, so as to improve communication
  - apply these techniques all the way down to the lowest levels of abstraction of the software, paying particular attention to the distribution of behavior, so that evolution remains cheap
- BDD is using **examples** at multiple levels to create a **shared understanding** and surface uncertainty to deliver **software that matter** (by Dan North)

A BBD Development Process

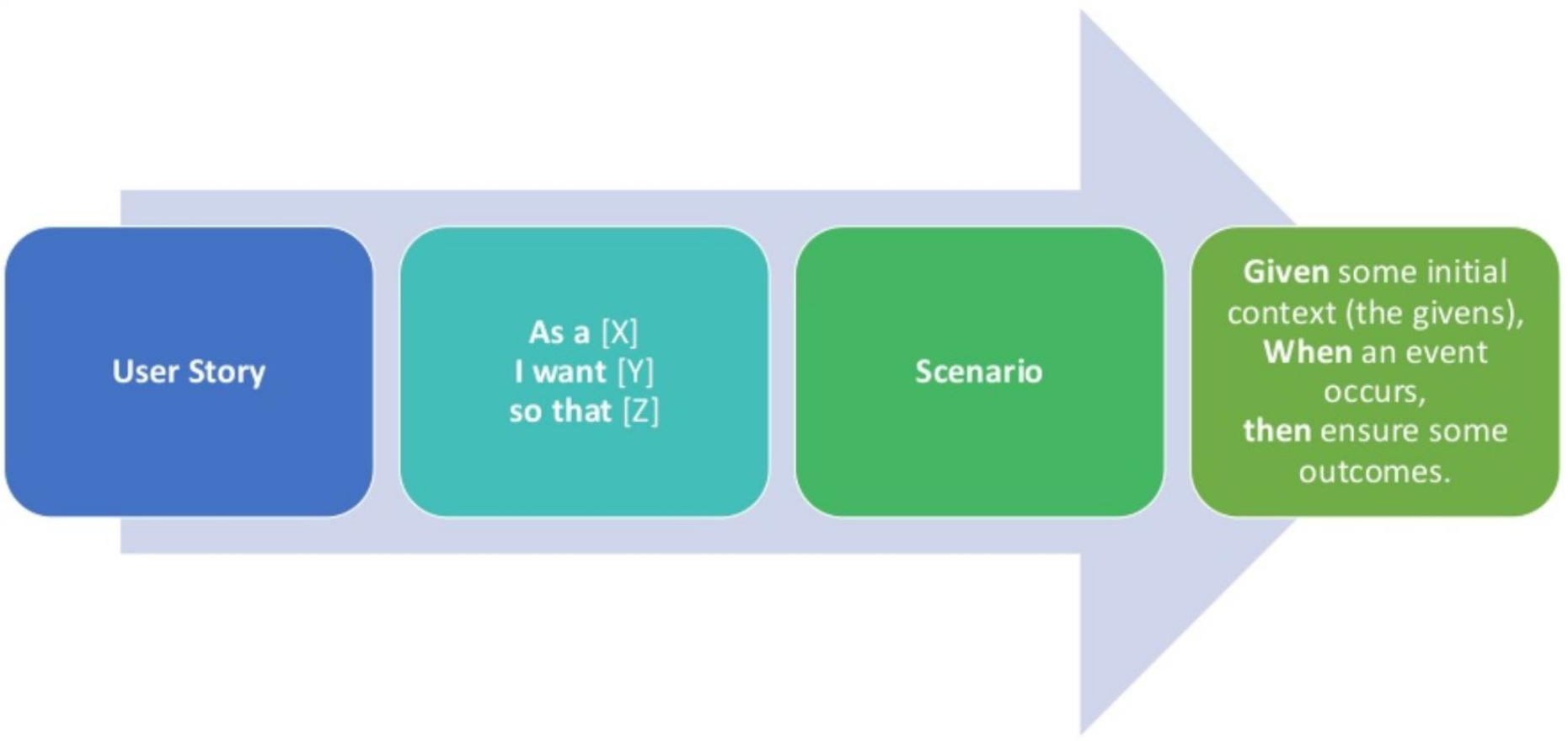


BBD + TDD

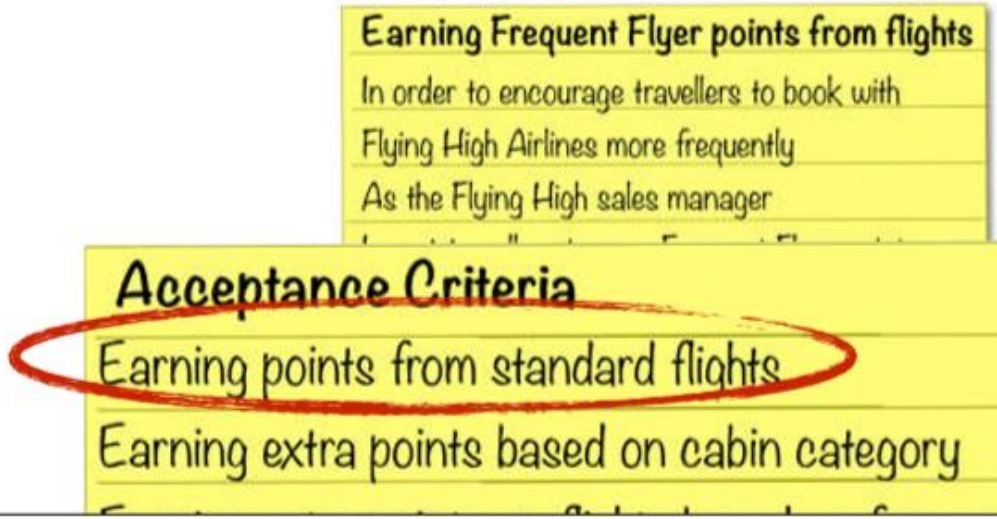

Write Failing FEATURE
BDD
Write Failing Test
TDD
n cycles
Refactor
Make Test Pass


1 Describe Behaviour
2 Write step definition
3 Run and fail
4 Write code to make step pass
5 Run and pass

Stages of Applying BDD



Acceptance Criteria, Scenarios, Examples




```
Scenario: Earning standard points from an Economy flight
Given the flying distance between Sydney and Melbourne is 878 km
And I am a standard Frequent Flyer member
When I fly from Sydney to Melbourne
Then I should earn 439 points
```

```
Scenario: Earning standard points from an Economy flight
Given the flying distance between <departure> and <destination> is <distance> km
And I am a standard Frequent Flyer member
When I fly from <departure> to <destination>
Then I should earn <points> points
Examples:
| departure | destination | distance | points | comments                                    |
| Sydney    | Melbourne   | 878      | 439    | 1 point per 2 kms for domestic flights      |
| Sydney    | Perth       | 4100     | 2050   |                                             |
| Sydney    | Hong Kong   | 7500     | 5000   | 1 point per 1.5 kms for international flights |
```

A Simple Feature File

```
Feature: Google Searching

  As a web surfer, I want to search Google, so that I can learn new things.


  Scenario: Simple Google search

    Given a web browser is on the Google page

    When the search phrase "panda" is entered

    Then results for "panda" are shown

    And the related results include "Panda Express"

    But the related results do not include "pandemonium"

      | Panda Express |

      | giant panda   |

      | panda videos  |
```

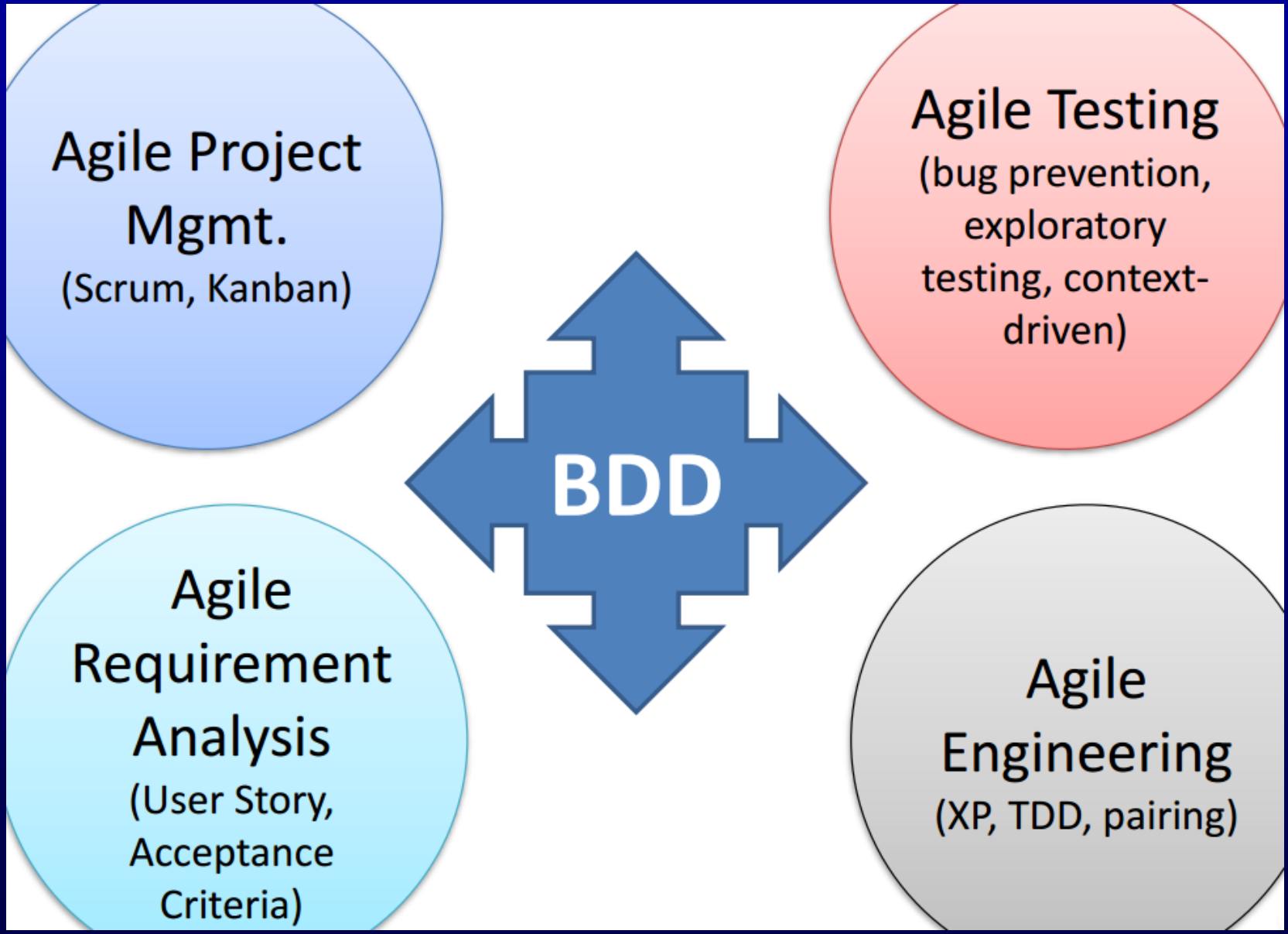
Scenario Outlines

```
Feature: Google Searching

  As a web surfer, I want to search Google, so that I can learn new things.


  Scenario Outline: Simple Google searches

    Given a web browser is on the Google page

    When the search phrase "<phrase>" is entered

    Then results for "<phrase>" are shown

    And the related results include "<related>"


    Examples: Animals

      | phrase   | related       |

      | panda    | Panda Express |

      | elephant | Elephant Man  |
```

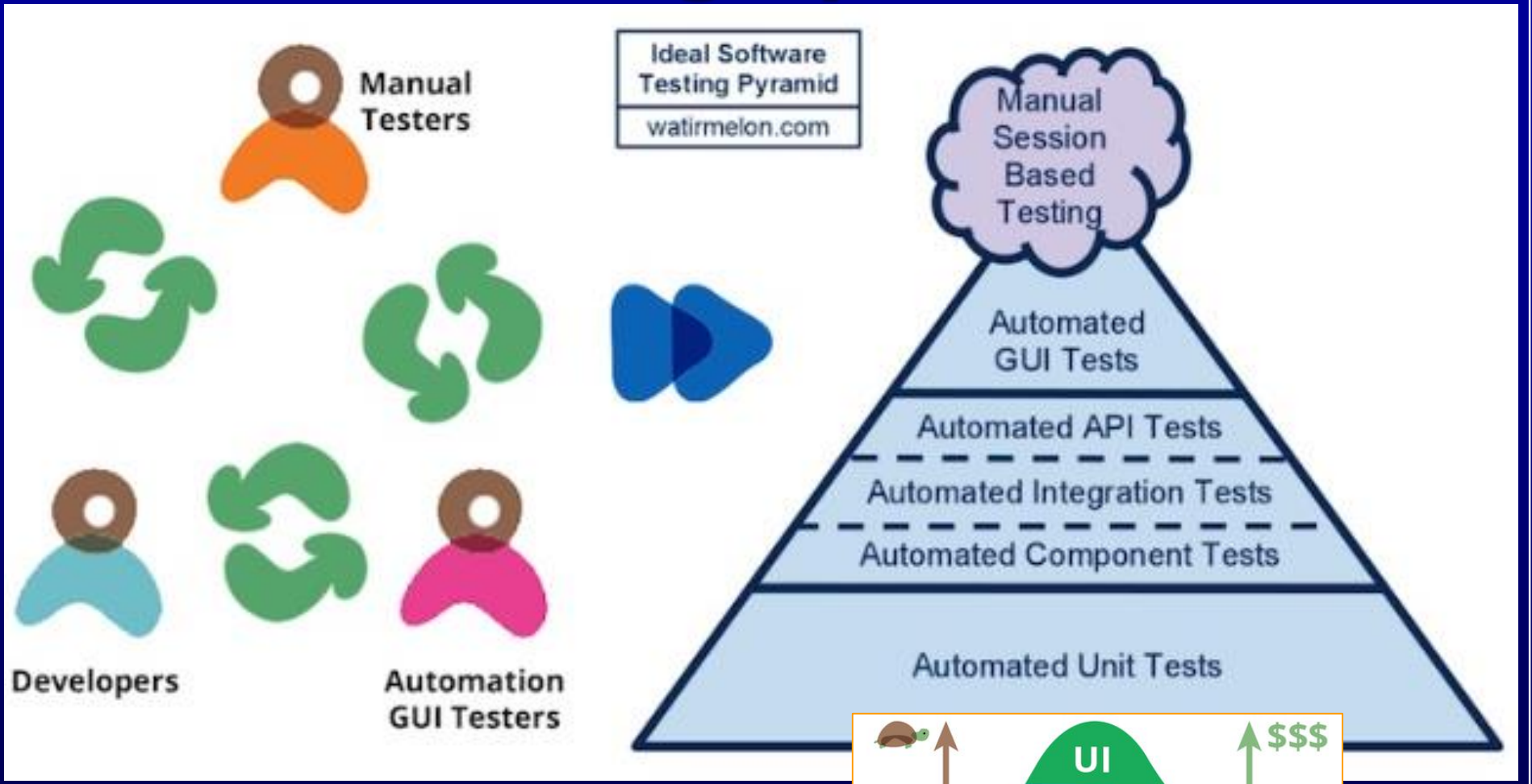
TDD vs. BDD vs. ATDD

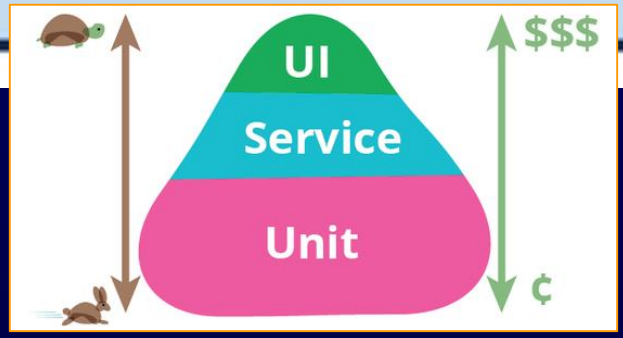
Parameters	TDD	BDD	ATDD
Definition	TDD is a development technique that focuses more on the implementation of a feature	BDD is a development technique that focuses on the system's behavior	ATDD is a technique similar to BDD focusing more on capturing the requirements
Participants	Developer	Developers, Customer, QAs	Developers, Customers, QAs
Language used	Written in a language similar to the one used for feature development (Eg. Java, Python, etc)	Simple English, (Gherkin)	Simple English, Gherkin
Main Focus	Unit Tests	Understanding Requirements	Writing Acceptance Tests
Tools used	JDave, Cucumber, JBehave, Spec Flow, BeanSpec, Gherkin Concordian, FitNesse	Gherkin, Dave, Cucumber, JBehave, Spec Flow, BeanSpec, Concordian	TestNG, FitNesse, EasyB, Spectacular, Concordian, Thucydides

BDD vs Agile Development



Testing Pyramid





The Testing Shortfall

- Do TDD tests (acceptance or otherwise) test the software well?
  - Do the tests achieve good coverage on the code?
  - Do the tests find most of the faults?
  - If the software passes, should management feel confident the software is reliable?

NO!

Why Not?

- Most agile tests focus on "*happy paths*"
  - What should happen under normal use
- They often miss things like
  - Confused-user paths
  - Creative-user paths
  - Malicious-user paths

The agile methods literature does not give much guidance

What Should Testers Do?

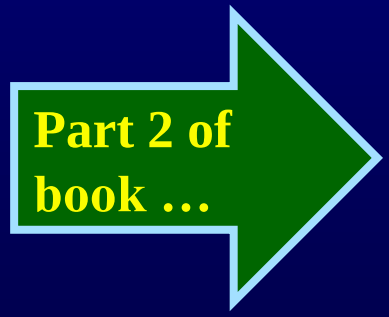
Ummm ... Excuse me, Professor ...

What do I do?

Design Good Tests

1. Use a human-based approach

- Create additional user stories that describe non-happy paths
- How do you know when you're finished?
- Some people are very good at this, some are bad, and it's hard to teach



2. Use modeling and criteria

- Model the input domain to design tests
- Model software behavior with graphs, logic, or grammars
- A built-in sense of completion
- Much easier to teach—engineering
- Requires discrete math knowledge

Summary

- More companies are putting testing first
- This can dramatically decrease cost and increase quality
- A different view of "*correctness*"
  - Restricted but practical
- Embraces evolutionary design
- TDD is definitely not test automation
  - Test automation is a prerequisite to TDD
- Agile tests aren't enough

What is Exploratory Testing?

- **Exploratory testing** is a type of software testing where test cases are not created in advance but testers check system on the fly. They may note down ideas about what to test before test execution. The focus of exploratory testing is more on testing as a "thinking" activity.
  - Exploratory testing is about **discovery, investigation and learning**. It is a simultaneous process of test design and test execution all done at the same time
  - Exploratory testing is also complementary to test automation; that is while automated checks are checking for regression issues, Exploratory testing focuses on new features which have been developed.
- Exploratory testing is, more than strictly speaking a "practice," a style or approach to testing software which is often contrasted to "scripted testing," and characterized by the following aspects among others:
  - it emphasizes the **tester's autonomy, skill and creativity**, much as other Agile practices emphasize these qualities in developers;
  - it recommends performing various test-related activities (such as test design, test execution, and interpretation of results) **in an interleaved manner**, throughout the project, rather than in a fixed sequence and at a particular "phase";
  - it emphasizes the mutually supportive nature of these techniques, and the need for a plurality of testing approaches rather than a formal "test plan"

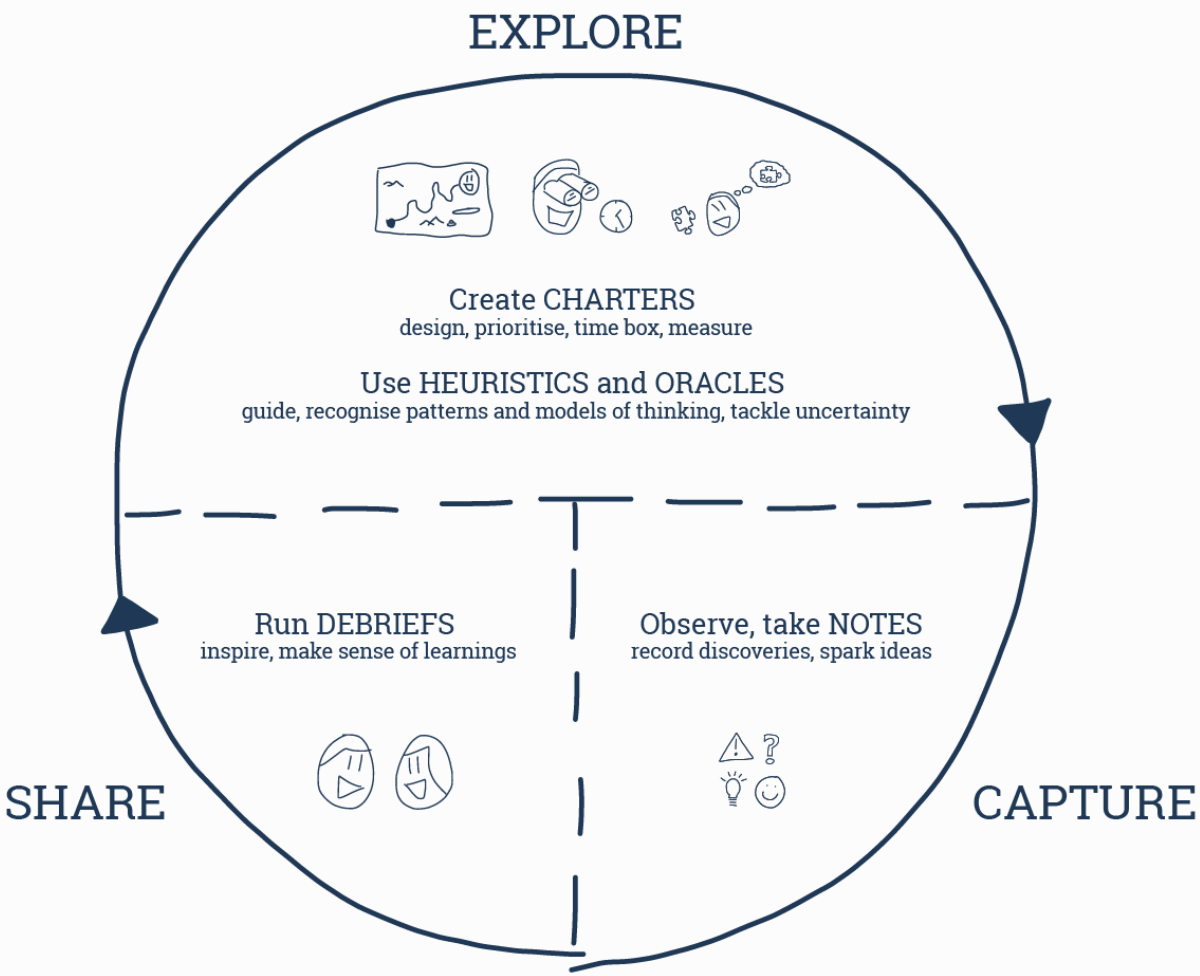
Differences between Scripted and Exploratory Testing

Scripted Testing	Exploratory Testing
Directed from requirements	Directed from requirements and exploring during testing
Determination of test cases well in advance	Determination of test cases during testing
Confirmation of testing with the requirements	Investigation of system or application
Emphasizes prediction and decision making	Emphasizes adaptability and learning
Involves confirmed testing	Involves Investigation
Is about Controlling tests	Is about Improvement of test design
Like making a speech - you read from a draft	Like making a conversation - it's spontaneous
The script is in control	The tester's mind is in control

How to do Exploratory Testing

- Exploratory test preparation goes through following 5 stages detailed below and it is also called session based test management (SBTM Cycle):
  - Create a Bug Taxonomy (classification)
    - Categorize common types of faults found in the past projects
    - Analyze the root cause analysis of the problems or faults
    - Find the risks and develop ideas to test the application.
  - Test Charter
    - Test Charter should suggest
    - what to test
    - how it can be tested
    - What needs to be looked
    - Test ideas are the starting point of exploration testing
    - Test charter helps determine how the end user could use the system
  - Time Box
    - This method includes a pair of testers working together not less than 90 minutes
    - There should not be any interrupted time in those 90 minutes session
    - Timebox can be extended or reduced by 45 minutes
    - This session encourages testers to react on the response from the system and prepare for the correct outcome
  - Review Results: (result analysis)
    - Evaluation of the defects
    - Learning from the testing
    - Analysis of coverage areas
  - Debriefing: (cross-examine)
    - Compilation of the output results
    - Compare the results with the charter
    - Check whether any additional testing is needed

How to do Exploratory Testing



How to do Exploratory Testing

- During exploratory execution, the following needs to be done:
  - The mission of testing should be very clear
  - Keep notes on what needs to be tested, why it needs to be tested and the assessment of the product quality
  - Tracking of questions and issues raised during exploratory testing
  - Better to pair up the testers for effective testing
  - The more we test, more likely to execute right test cases for the required scenarios
- It is very important to take a document and monitor the following
  - Test Coverage - Whether we have taken notes on the coverage of test cases and improve the quality of the software
  - Risks to be covered - Which risks need to be covered and which are all important ones?
  - Test Execution Log - Recordings on the test execution
  - Issues / Queries - Take notes on the question and issues on the system

Summary of Exploratory Testing

- Exploratory testing is performed to overcome the limitations of scripted testing. It helps in improving Test Case suite. It empathizes on learning and adaptability
- Exploratory testing can be used extensively when
  - The testing team has experienced testers
  - Early iteration is required
  - There is a critical application
  - New testers entered into the team
- Challenges of Exploratory Testing
  - Learning to use the application or software system is a challenge
  - Replication of failure is difficult
  - Determining whether tools need to be used can be challenging
  - Determine the best test cases to execute can be difficult
  - Reporting of the test results is a challenge as the report doesn't have planned scripts or cases to compare with the actual result or outcome
  - Documentation of all events during execution is difficult to record
  - Don't know when to stop the testing as exploratory testing has NO definite test cases to execute.