AUT UNIVERSITY | COMPUTER + MATHEMATICAL SCIENCES

Week 5: Requirements and BDD

409232 Software Development Methods

Jim Buchan

SERL

Quizz

- Chapter 10 on Agile Methods

Ass1 Case Study

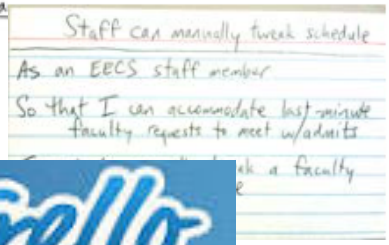
- Any issues?

Assignment 2

- Sprint 0 - weeks 3,4,5
- Sprint 1 weeks 6,7
- Sprint 2 weeks 8,9
- Sprint 3 weeks 10,11
- Delivery week 12
- 6 Teams of 6-8 members
  - Analyst x 1, testers x 2, coders x 2, scrum master x 1
  - 2 competing streams of 3 teams developing different versions of the app.
    - 1 team - input of information
    - 1 team - querying information
    - 1 team - workflow and administration

Collaboratively developing an Application

Understand the Problem/Value



User Stories

Scenarios

Mockups

Personas

Design

Features

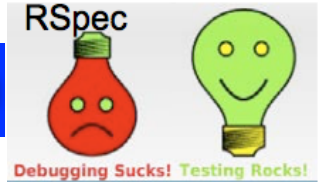
Test

Code

Behaviour Driven Development

Cucumber

Test First Design



Share Code Version Control

GitHub

Continuous Integration

Travis CI

Deploy as a SaaS

heroku

Behaviour Driven Development

BDD combines ideas from test driven design (TDD) and Acceptance test driven development (ATDD) to get developers to share deep understanding of the features that have value to the PO.

A single notation is used that is readable by developers, domain experts, clients, testers. This should improve communications.

http://guide.agilealliance.org/guide/bdd.html

Release

How can we release value incrementally?

What subset of business objectives will each release achieve?

What user constituencies will the release serve?

What general capabilities (big stories) will the release offer?

Release Roadmap

Target Customers

Target Personas

Product or Project

What business objectives will the product fulfill?

Product Goals

Product Charter

Customers

User Personas

Iteration or Sprint

What specifically will we build? (user stories)

How will this iteration move us toward release objectives?

Iteration Goal

Development or Construction Tasks

Story (Backlog Item)

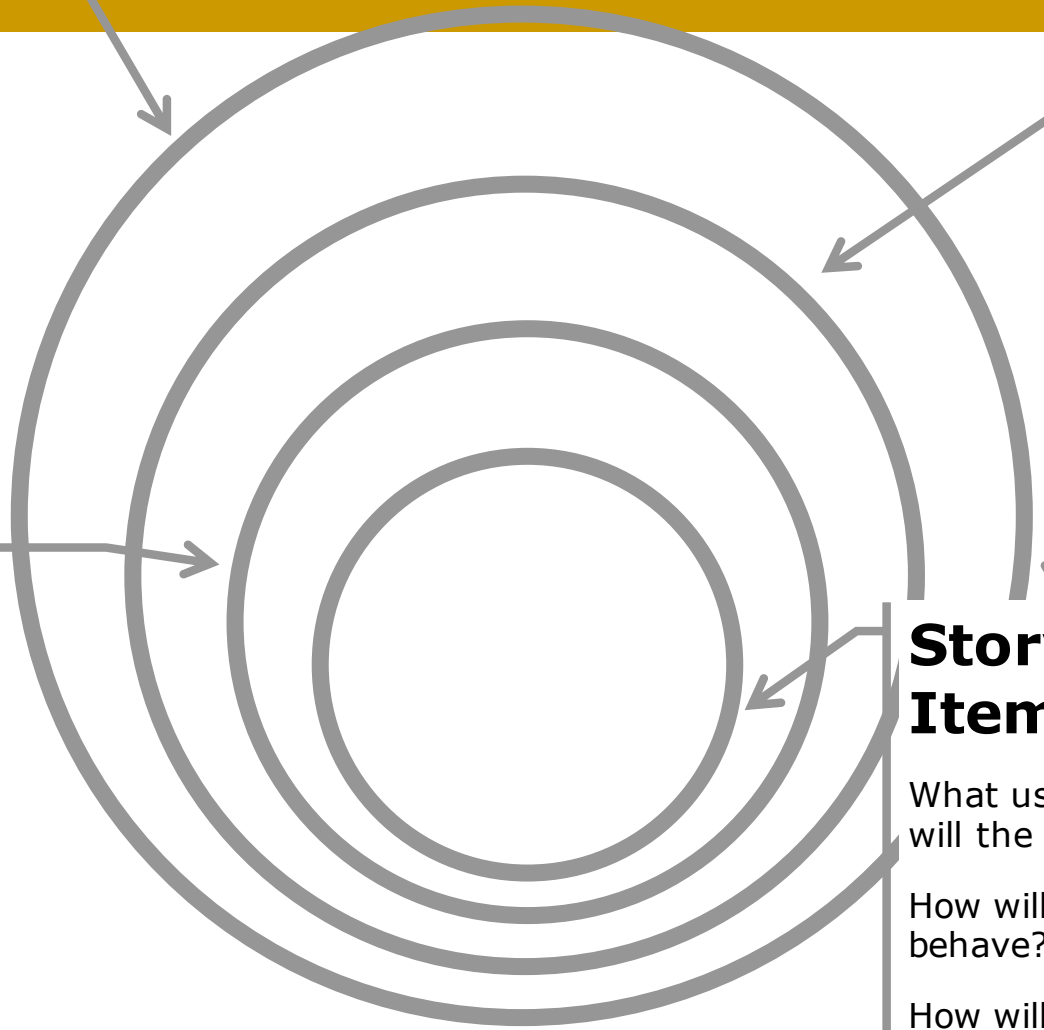
What user or stakeholder need will the story serve?

How will it specifically look and behave?

How will I determine if it's completed?

Story Details

Acceptance Tests



The problems with requirements

"Customer: Hello, I'd like to order a cake.

Employee: Sure, what would you like written on it?

Customer: Could you write "So long, Alicia" in purple?

Employee: Sure.

Customer: And put stars around it?

Employee: No problem. I've written this up, and will hand it to my cake decorator right away. We'll have it for you in the morning."

Excerpt From: Jeff Patton. "User Story Mapping:

What you heard is not what I said!

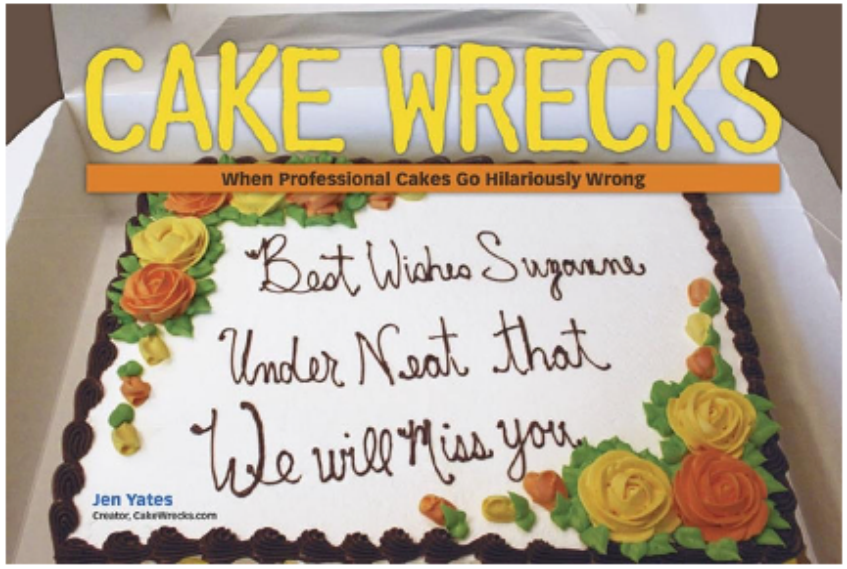
Could you write

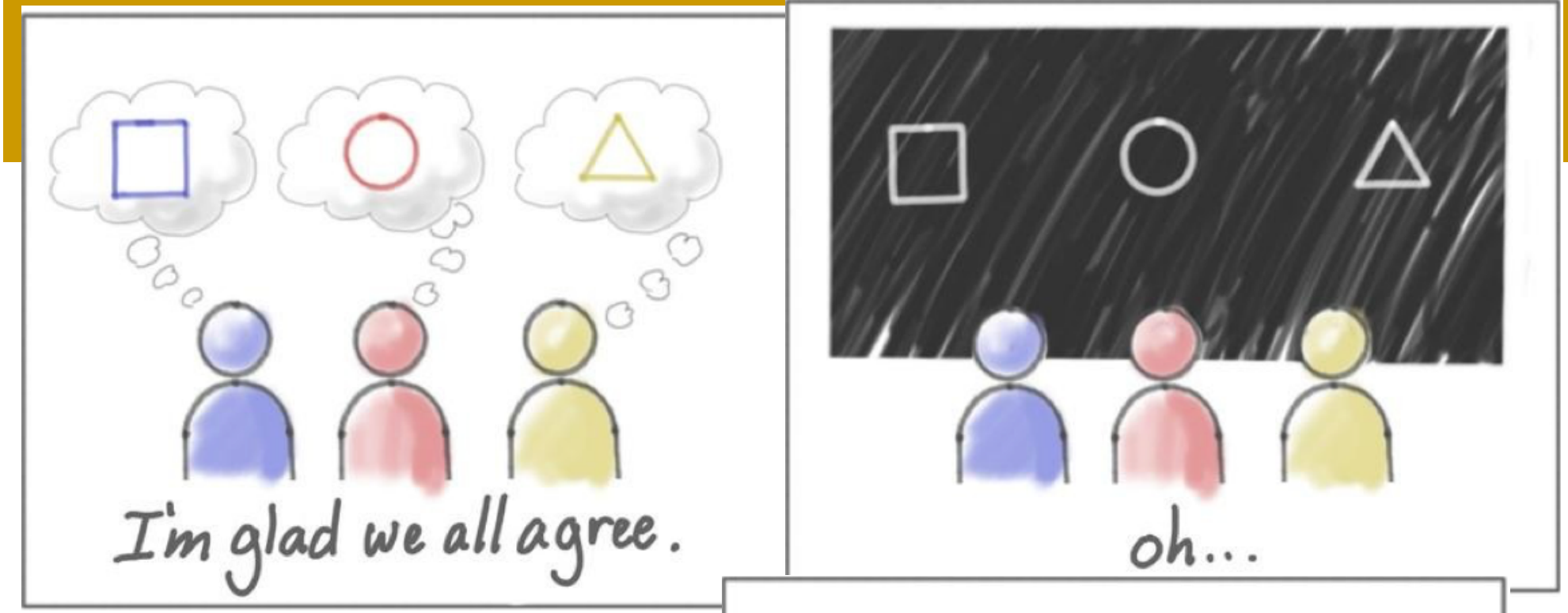
"So long, Alicia" in purple?

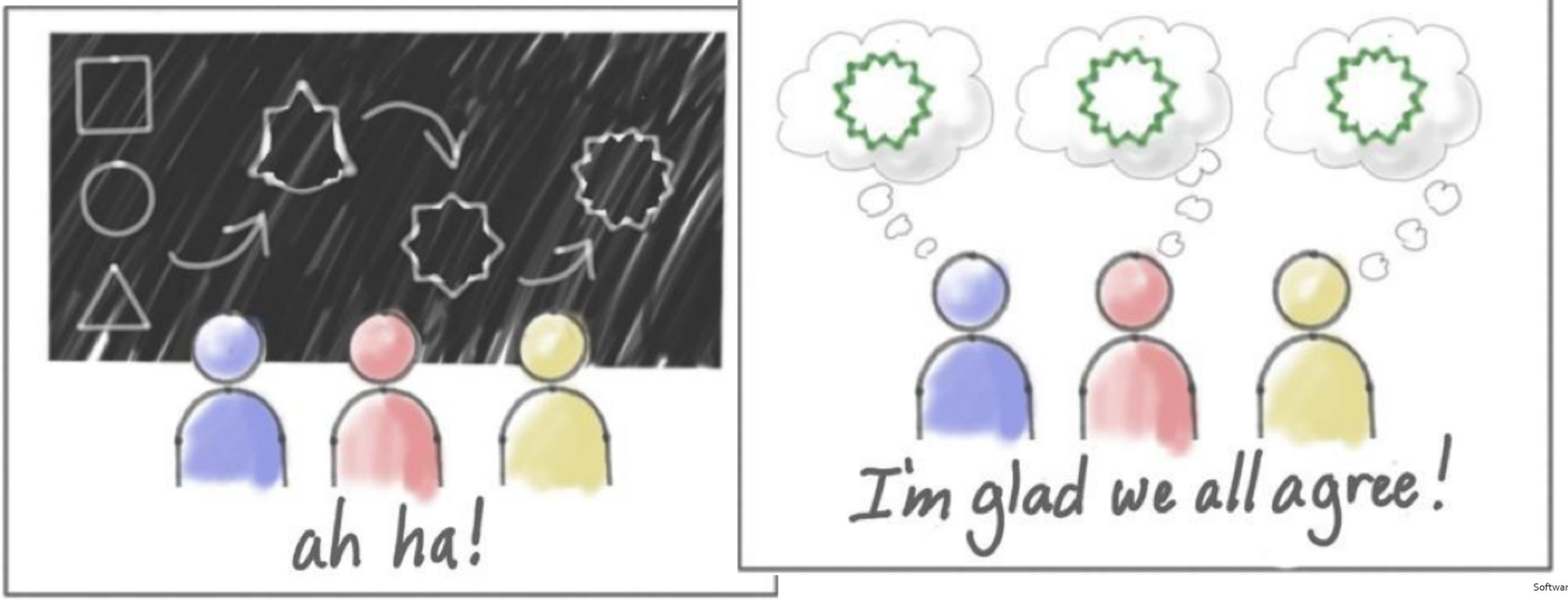
"And put stars around it?"

Customer->phone call taker->decorator

Shared understanding?







SERL

Software Engineering Research Laboratory

Documentation is like a holiday photo ?

What does this mean to you?

Shared documents are NOT shared understanding

Shared understanding is…
two or more people
can describe, explain, predict
some imagined future
in a similar way…

Shared experiences are important to shared understanding

What kind of shared experiences??

Language - Empathy and perspective

SERLER

As a researcher

So that I can focus on a particular method

I want to be able to specify a method that has been studied with a specified metric and outcome

Language - Empathy and perspective

```
Given /^I am on the contact page$/ do
  pending # express the regexp above with the code you wish you had
end

Given /^I fill in "(.*?)" with "(.*?)"$/ do |arg1, arg2|
  pending # express the regexp above with the code you wish you had
end

When /^I press "(.*?)"$/ do |arg1|
  pending # express the regexp above with the code you wish you had
end

Then /^page should have notice message "(.*?)"$/ do |arg1|
  pending # express the regexp above with the code you wish you had
end
```

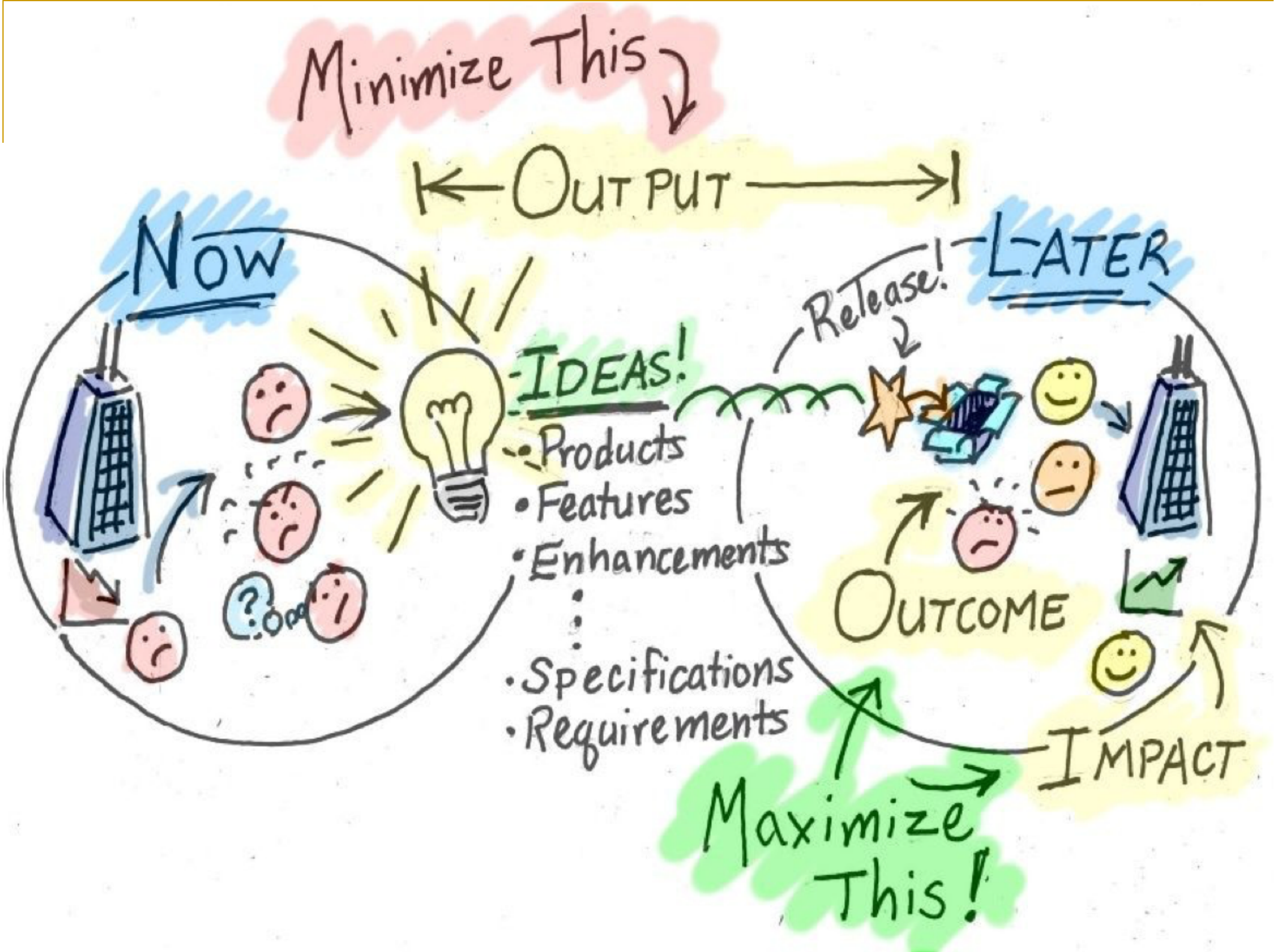
The real goal of User Stories is Shared Understanding

- They don't have to be PERFECT (INVEST)
- Templates are just a thinking guide

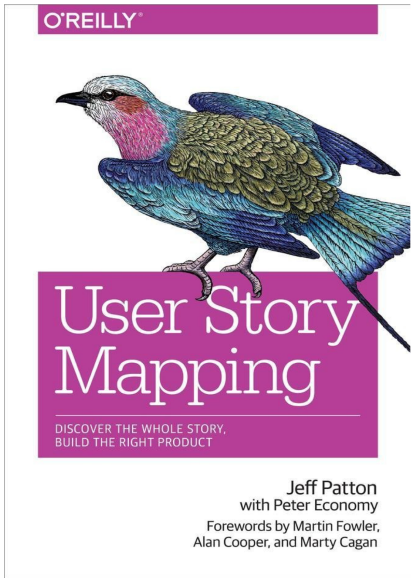
  As I …so that… I want ….

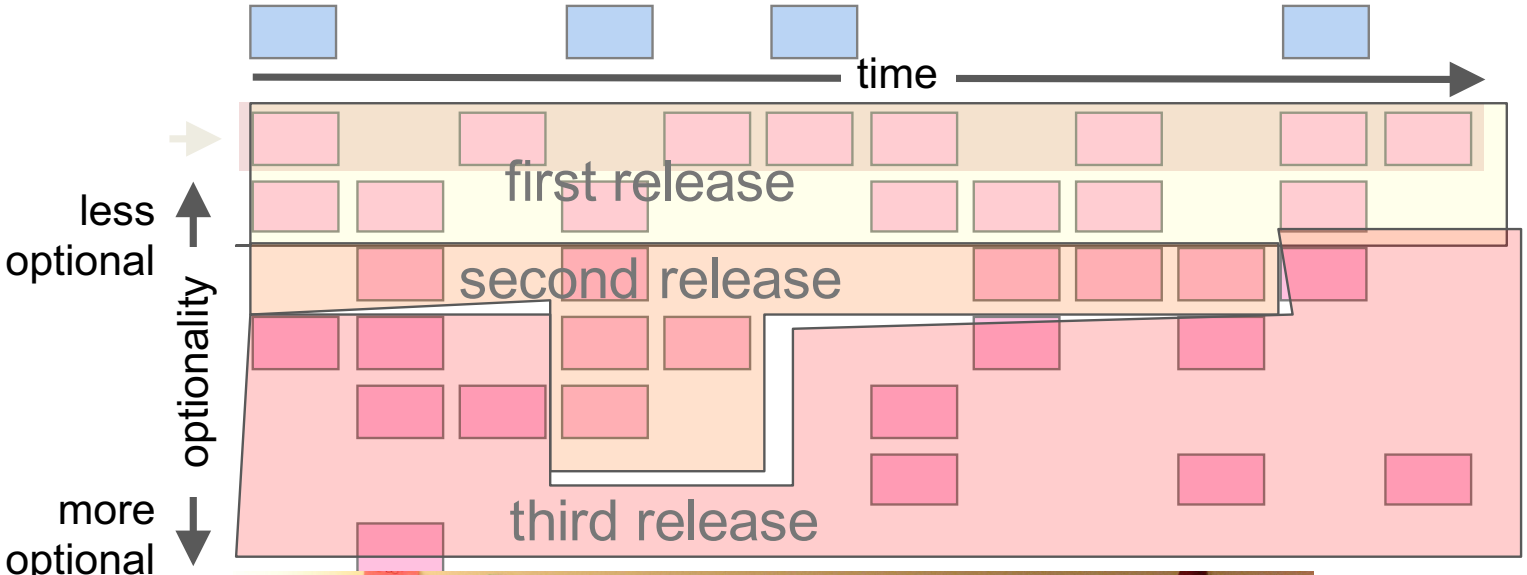
Talk, draw, use stick notes, models, WHATEVER ..take a photo or video!

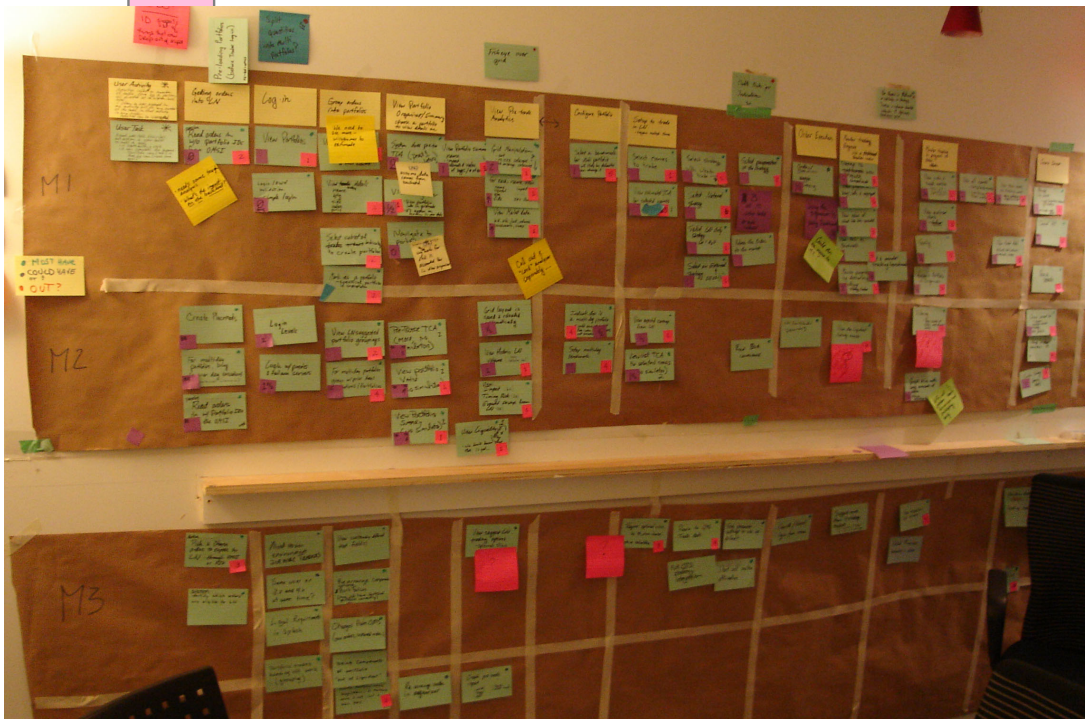
It's not what is written down that is important…it's what you remember when you read it


Minimize This
OUTPUT
NOW
IDEAS!
Release!
LATER
• Products
• Features
• Enhancements
• Specifications
• Requirements
OUTCOME
IMPACT
Maximize This!

User Story Mapping







AUT UNIVERSITY COMPUTER + MATHEMATICAL SCIENCES

SERL
Software Engineering Research Laboratory

In your teams

- Read through the "documentation" for SERLER
- What is different to the imagined future situation compared to the current situation without SERLER
- Write some user stories and acceptance tests

Agile Quality Assurance (Chapter 7 of the Textbook)

Agile Quality Assurance…

- Finding bugs early in development costs less to fix
- Frequent reviews and short feedback cycles
- Frequent refactoring
- Frequent testing in small chunks
- Test First development – unit/acceptance test level
- Acceptance tests – functional testing
- Continuous Integration (CI) costs less to fix

The many faces of Testing

- Unit
- Acceptance
- Functional
- Integration
- Smoke
- Regression
- UI

- Load / Stress
- Performance
- Usability
- Black-box
- White box
- Exploratory

Phases of Testing Activity

- Test Planning
- Test Design
- Test automation
- Test Execution
- Test analysis
- Test monitoring
- Test refactoring
- Test review

Test Driven Development

- What does this mean?
- What benefits claimed?
- How implement it?

Test Driven Development

Test-driven development (TDD) is the craft of producing automated tests for production code, and using that process to drive design and programming.

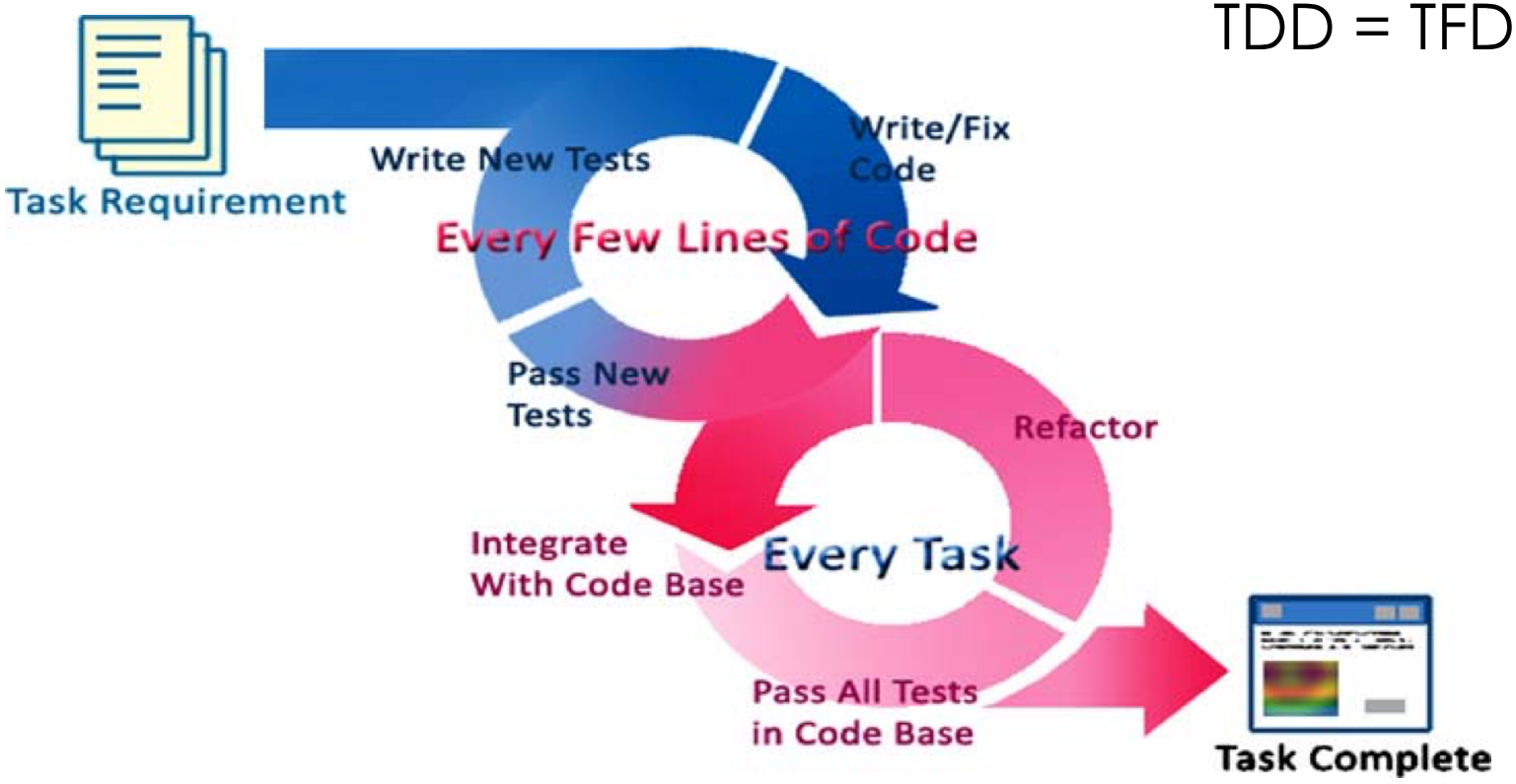
For every tiny bit of functionality in the production code, first develop a test that specifies and validates what the code will do.

You then produce exactly as much code as will enable that test to pass. Then you refactor (simplify and clarify) both the production code and the test code.

TDD is not testing - it is an approach to design

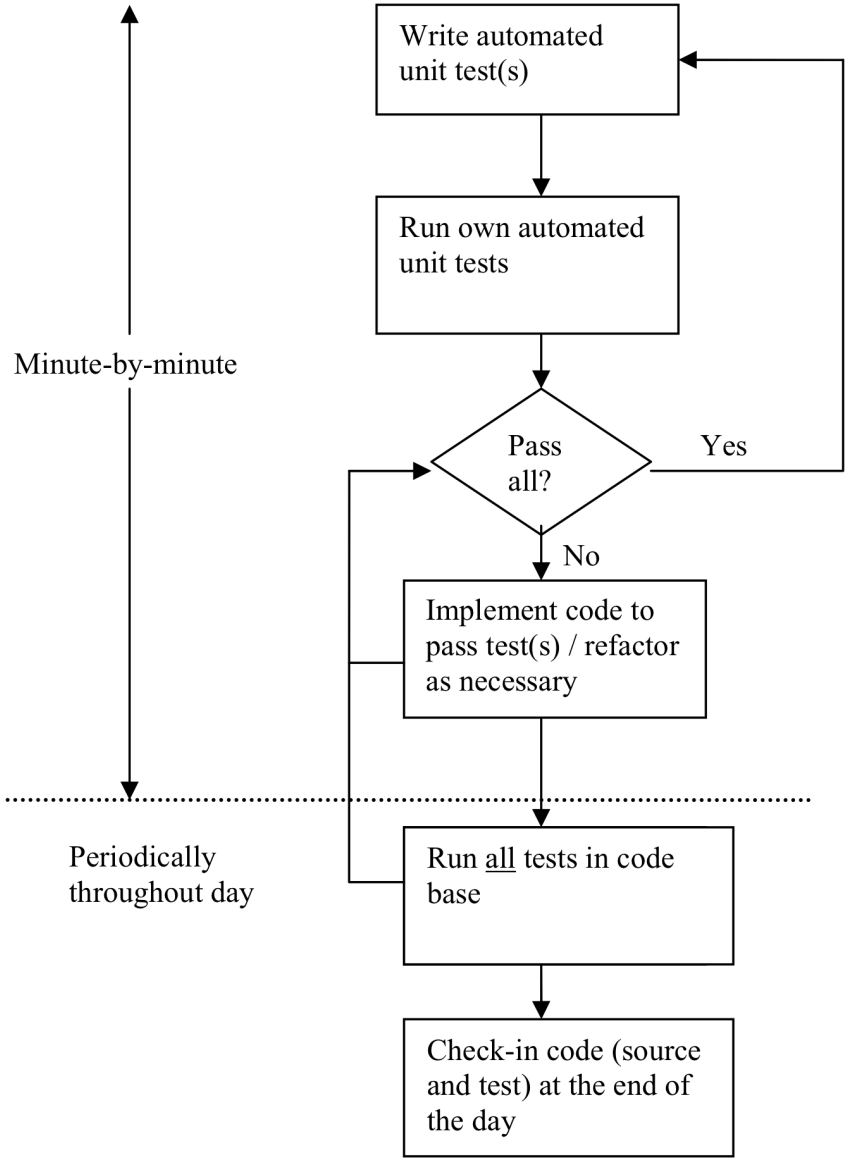
Test Driven Development

TDD = TFD + Refactoring



B. Nagappan, E. Michael Maximilien, T. Bhat & L. Williams Empir Software Eng (2008) 13:289–302

Test Driven Development



(Bhat & Nagappan, 2006)

Figure 1: Test-Driven Development

Test Driven Development

Table 1. Summary of TDD research in industry.

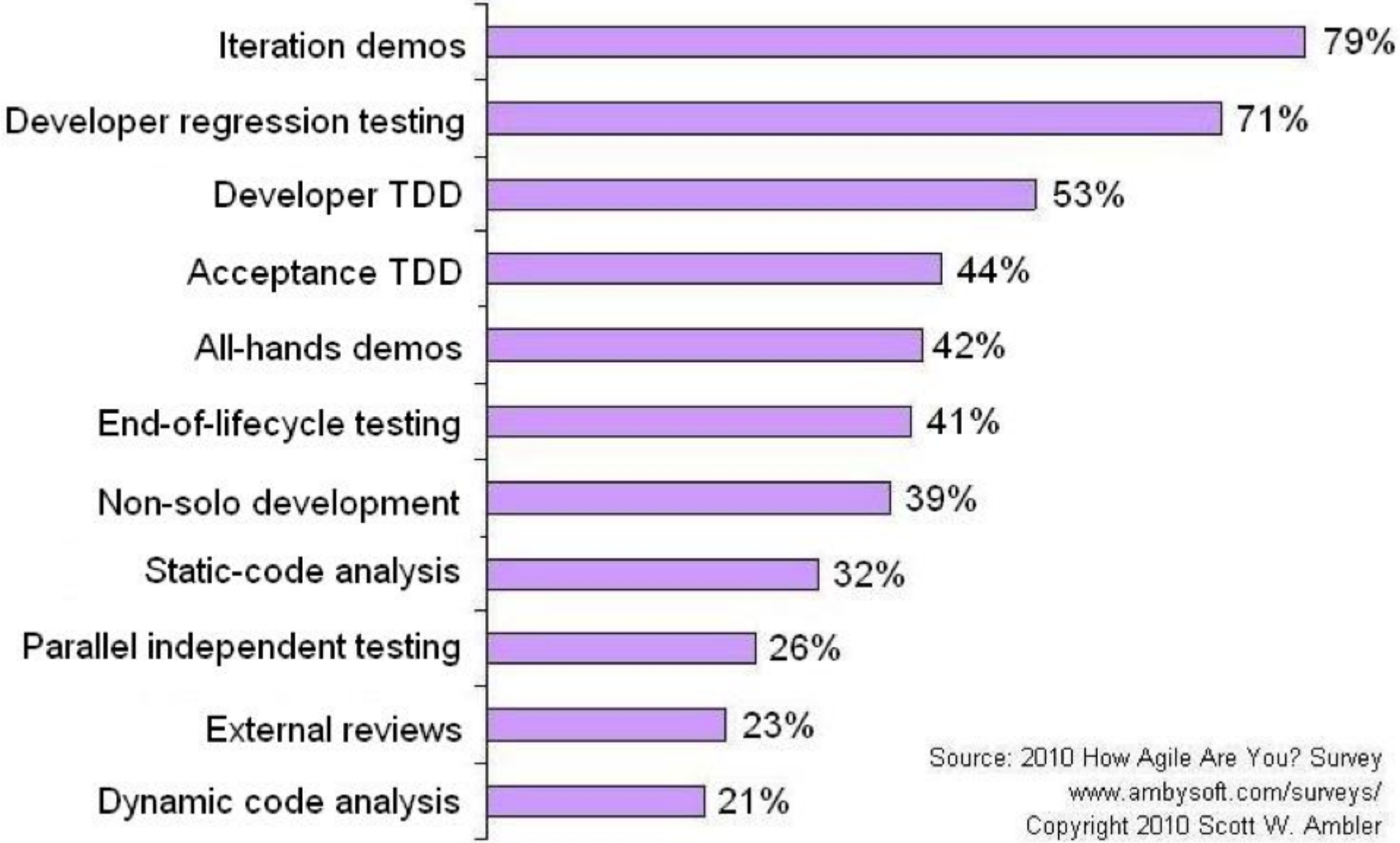
Study	Type	Number of companies	Number of programmers	Quality effects	Productivity effects
George[8]	Controlled experiment	3	24	TDD passed 18% more tests	TDD took 16% longer
Maximilien[9]	Case study	1	9	50% reduction in defect density	Minimal impact
Williams[10]	Case study	1	9	40% reduction in defect density	No change

Table 2. Summary of TDD research in academia.

Controlled experiment	Number of programmers	Quality effects	Productivity effects
Kaufmann[11]	8	Improved information flow	50% improvement
Edwards[12]	59	54% fewer defects	n/a
Erdogmus[13]	35	No change	Improved productivity
Müller[14]	19	No change, but better reuse	No change
Pančur[15]	38	No change	No change

Janzen, D., Hossein, S. (2005) Test-Driven Development: Concepts, Taxonomy and Future Direction. *Computer,* September 2005

How are Agile Teams Validating their own Work?



Source: 2010 How Agile Are You? Survey
www.ambysoft.com/surveys/

Test-Driven Development: Concepts, Taxonomy, and Future Direction

Test-driven development creates software in very short iterations with minimal upfront design. Poised for widespread adoption, TDD has become the focus of an increasing number of researchers and developers.

David Janzen
Simex LLC

Hossein Saiedian
University of Kansas

The *test-driven development* strategy requires writing automated tests prior to developing functional code in small, rapid iterations. Although developers have been applying TDD in various forms for several decades,[1] this software development strategy has continued to gain increased attention as one of the core extreme programming practices.

XP is an *agile method* that develops object-oriented software in very short iterations with little upfront design. Although not originally given this name, TDD was described as an integral XP practice necessary for analysis, design, and testing that also enables design through refactoring, collective ownership, continuous integration, and programmer courage.

Along with pair programming and refactoring, TDD has received considerable individual attention since XP's introduction. Developers have created tools specifically to support TDD across a range of languages, and they have written numerous books explaining how to apply TDD concepts. Researchers have begun to examine TDD's effects on defect reduction and quality improvements in academic and professional practitioner environments, and educators have started to examine how to integrate TDD into computer science and software engineering pedagogy. Some of these efforts have been implemented in the context of XP projects, while others are independent of them.

method, a close examination of the term reveals a more complex picture.

The test aspect

In addition to testing, TDD involves writing automated tests of a program's individual units. A unit is the smallest possible testable software component. There is some debate about what exactly constitutes a unit in software. Even within the realm of object-oriented programming, both the class and method have been suggested as the appropriate unit. Generally, however, the method or procedure is the smallest possible testable software component.

Developers frequently implement test drivers and function stubs to support the execution of unit tests. Test execution can be either a manual or automated process and can be performed by developers or designated testers. Automated testing involves writing unit tests as code and placing this code in a test harness or framework such as JUnit. Automated unit testing frameworks minimize the effort of testing, reducing a large number of tests to a click of a button. In contrast, during manual test execution developers and testers must expend effort proportional to the number of tests executed.

Traditionally, unit testing occurred after developers coded the unit. This can take anywhere from a few minutes to a few months. The unit tests might be written by the same programmer or by a designated tester. With TDD, the programmer writes the

Does Test-Driven Development Really Improve Software Design Quality?

David S. Janzen, *California Polytechnic State University, San Luis Obispo*
Hossein Saiedian, *University of Kansas*

TDD is first and foremost a design practice. The question is, how good are the resulting designs? Empirical studies help clarify the practice and answer this question.

Software developers are known for adopting new technologies and practices on the basis of their novelty or anecdotal evidence of their promise. Who can blame them? With constant pressure to produce more with less, we often can't wait for evidence before jumping in. We become convinced that competition won't let us wait.

Advocates for test-driven development claim that TDD produces code that's simpler, more cohesive, and less coupled than code developed in a more traditional test-last way. Support for TDD is growing in many development contexts beyond its common association with Extreme Programming. Examples such as Robert C. Martin's bowling game demonstrate the clean and sometimes surprising designs that can emerge with TDD,[1] and the buzz has proven sufficient for many software developers to try it. Positive personal experiences have led many to add TDD to their list of "best practices," but for others, the jury is still out. And although the literature includes many publications that teach us how to do TDD, it includes less empirical evaluation of the results.

In 2004, we began a study to collect evidence that would substantiate or question the claims regarding TDD's influence on software.

TDD misconceptions

We looked for professional development teams who were using TDD and willing to participate in the study. We interviewed representatives from four reputable Fortune 500 companies who claimed to be using TDD. However, when we dug a little deeper, we discovered some unfortunate misconceptions:

- Misconception #1: TDD equals automated testing. Some developers we met placed a heavy emphasis on automated testing. Because TDD has helped propel automated testing to the forefront, many seem to think that TDD is only about writing automated tests.
- Misconception #2: TDD means write all tests first. Some developers thought that TDD involved writing the tests (all the tests) first, rather than using the short, rapid test-code iterations of TDD.

Unfortunately, these perspectives miss TDD's primary purpose, which is design. Granted, the tests are important, and automated test suites that can run at the click of a button are great. However,

0740-7459/08/$25.00 © 2008 IEEE

Does Test-Driven Development Improve the Program Code? Alarming Results from a Comparative Case Study

Maria Siniaalto[1] and Pekka Abrahamsson[2]

[1] F-Secure Oyj,
Elektroniikkatie 3, FIN-90570 Oulu, Finland
Maria.Siniaalto@f-secure.com
[2] VTT Technical Research Centre of Finland,
P.O. Box 1100, FIN-90571 Oulu, Finland
Pekka.Abrahamsson@vtt.fi

Abstract. It is suggested that test-driven development (TDD) is one of the most fundamental practices in agile software development, which produces loosely coupled and highly cohesive code. However, how the TDD impacts on the structure of the program code have not been widely studied. This paper presents the results from a comparative case study of five small scale software development projects where the effect of TDD on program design was studied using both traditional and package level metrics. The empirical results reveal that an unwanted side effect can be that some parts of the code may deteriorate. In addition, the differences in the program code, between TDD and the iterative test-last development, were not as clear as expected. This raises the question as to whether the possible benefits of TDD are greater than the possible downsides. Moreover, it additionally questions whether the same benefits could be achieved just by emphasizing unit-level testing activities.

Keywords: Test-Driven Development, Test-first Programming, Test-first Development, Agile Software Development, Software Quality.

1 Introduction

Test-driven development (TDD) is one of the core elements of Extreme Programming (XP) method [1]. The use of the TDD is said to yield several benefits. It is claimed to improve test coverage [2] and to produce loosely coupled and highly cohesive systems [3]. It is also believed to encourage the implementation scope to be more explicit [3] and to enable more frequent integration [4]. On the other hand, it is claimed that rapid changes may cause expensive breakage in tests and that the lack of application or testing skills may produce inadequate test coverage [5]. TDD has also received criticism over not being very suitable for systems such as multithreaded applications or security software, since it cannot mechanically demonstrate that their goals have been met [6]. However, the scientific empirical evidence behind all of these claims is currently sparse, and thus it is difficult to

B. Meyer, J.R. Nawrocki, and B. Walter (Eds.): CEE-SET 2007, LNCS 5082, pp. 143–156, 2008.

What Makes Testing Work: Nine Case Studies of Software Development Teams

Christopher D Thomson*
Business School
University of Hull
Hull, UK
c.thomson@hull.ac.uk

Mike Holcombe, Anthony J H Simons
Department of Computer Science
University of Sheffield
Sheffield, UK
{m.holcombe,a.simons}@dcs.shef.ac.uk

***Abstract*— Recently there has been a focus on test first and test driven development; several empirical studies have tried to assess the advantage that these methods give over testing after development. The results have been mixed. In this paper we investigate nine teams who tested during coding to examine the effect it had on the external quality of their code. Of the top three performing teams two used a documented testing strategy and the other an ad-hoc approach to testing. We conclude that their success appears to be related to a testing culture where the teams proactively test rather than carry out only what is required in a mechanical fashion.**

Testing; test first; test driven development; extreme programming;emprical; qualitative; testing culture.

I. INTRODUCTION

Extreme programming (XP) [1] presents what is, on the surface, a simple but effective testing practice known as *test first*, or *test driven development*. The idea is reassuringly simple, that unit tests should be defined and run before any implementation is present. Several studies have attempted to measure the effect of the *test first* practice on the quality of the software produced and time taken, but the results presented are inconclusive.

The testing practice of XP encompassed more than simply *test first*. System testing is automated, incremental, regular, and early. User acceptance testing is similar although often less or not at all automated [1]. But these features can be used independently of *test first* and perhaps with some success. This raises our research question:

How does the practice of testing effect a team following XP – or if test first is not followed then do the other practices of XP still influence the way testing is performed and the external quality?

We collected data from nine teams, which we present as case studies. In the case studies the teams were novice users of XP, who we provided with training. They were given the option of using *test first* and whilst three teams expressed enthusiasm they ultimately did not follow the practice accurately. We found that the teams had a high degree of variation in external quality that cannot be easily explained by the teams' testing practice alone or the practices of XP alone. Instead we find that testing must become part of the culture of the team, and can do so in at least two different ways.

*Author contributed whilst at the University of Sheffield.

II. LITERATURE REVIEW

Test first (TF) is an established development technique which is essentially the same as *test driven development* (TDD). In both cases the aim is to write tests before writing the functional code. This should in theory aid development as the tests form the basis of the specification, design and functional tests, whereas testing after the code is written is regarded as only a testing technique [1; 2]. Whilst TF and TDD are well defined, the traditional or *test last* technique is interpreted differently by the studies in the literature investigating this phenomenon. The studies' definitions of *test last* can be divided into roughly four categories, which we will refer to as TL 1 to 4:

TL-1: Unspecified traditional method. Most experiments provided at least a few hints as to the method that the non-TF teams should follow, however some did not [3-5]. The following statement was typical of the broad definitions that these papers used: "the control group which followed the traditional process" ([5], p132). As no further discussion was made about the process followed we can't make generalizations about what affected their performance.

TL-2: Specified traditional method. This method is typified by manual or ad-hoc testing and was used in three studies [6-8]. The method was defined thus: "no automated tests were written and the project was developed in traditional mode with a large up-front design and manual testing after the software was implemented." ([7], p71) and "a conventional design-develop-test (similar to waterfall)" ([6], p339). Lastly this study also noted that whilst the teams had been asked to use unit tests, only one team wrote any [6].

TL-3: Unit tests written at the end of the development cycle. Two studies identified a method that used unit testing at the end of the development cycle [7; 9]. These studies were able to provide some metrics on coverage to define the amount of testing undertaken, but assumed that all tests were only written and used at the end of the project. One study

978-0-7695-3820-4/09 $26.00 © 2009 IEEE
DOI 10.1109/TAICPART.2009.12

Third International Conference on Software Testing, Verification, and Validation Workshops

Most Common Mistakes in Test-Driven Development Practice: Results from an Online Survey with Developers

Mauricio Finavaro Aniche, Marco Aurélio Gerosa
Department of Computer Science - University of São Paulo (USP) - Brazil
{aniche,gerosa}@ime.usp.br

Abstract

Test-driven development (TDD) is a software development practice that supposedly leads to better quality and fewer defects in code. TDD is a simple practice, but developers sometimes do not apply all the required steps correctly. This article presents some of the most common mistakes that programmers make when practicing TDD, identified by an online survey with 218 volunteer programmers. Some mistakes identified were: to forget the refactoring step, building complex test scenarios, and refactor another piece of code while working on a test. Some mistakes are frequently made by around 25% of programmers.

1. Introduction

Test-driven development (TDD) is an important practice in Extreme Programming (XP) [1]. As agile practices suggest, software design emerges as software grows. In order to respond very quickly to changes, a constant feedback is needed and TDD gives it by making programmers constantly write a small test that fails and then make it pass. TDD is considered an essential strategy in emergent design because when writing a test prior to code, programmers contemplate and decide not only the software interface (e.g. class/method names, parameters, return types, and exceptions thrown), but also on the software behavior (e.g. expected results given certain inputs) [13].

TDD is not only about test. It is about helping the team to understand the features that the users need and to deliver those features reliably and predictably. TDD turns testing into a design activity, as programmers use tests to clarify the expectations of what a piece of code should do [3].

Many other assumptions are made about TDD. Some researches show that it helps the development process by increasing code quality and reducing the number of defects, as presented in Section 2.

Kent Beck sums up TDD as follows: 1) quickly add a test; 2) run all tests and see the new one fail; 3) make a little change; 4) run all tests and see them all succeed; 5) refactor to remove duplication [18]. In order to get all benefits from TDD, programmers should follow each step. As an example, the second step states that programmers should watch the new test fail and the fifth step states to refactor the code to remove duplication. Sometimes programmers just do not perform all steps of Beck's description. Thus, the value TDD aggregates to software development process might be reduced.

This article presents some of the most common mistakes that programmers make during their TDD sessions, based on an online survey conducted during two weeks in January, 2010, with 218 volunteer programmers. The survey and its data can be found at http://www.ime.usp.br/~aniche/tdd-survey/.

This article is structured as follows: Section 2 presents some studies about the effects of TDD on software quality; Section 3 shows the most common mistakes programmers make based on the survey; Section 4 discusses about the mistakes and ideas on how to sort them out; Section 5 presents threats to validity on the results of this article; Section 6 concludes and provides suggestions for future works.

2. The effects of TDD on software quality

Empirical experiments about the effects of TDD have been conducted generally with two different groups: graduate students at universities and professional developers at the industry. Most of them show that TDD increases code quality, reduces the defect density, and provides better maintainability. However, industry studies presented stronger results indicating that TDD is more helpful.

978-0-7695-4050-4/10 $26.00 © 2010 IEEE
DOI 10.1109/ICSTW.2010.16

xUnit

```
// Define a subclass of TestCase
public class StringTest extends TestCase {

// Create fixtures
protected void setUp(){ /* run before */}
protected void tearDown(){ /* after */ }

// Add testing methods public void testSimpleAdd() {
String s1 = new String("abcd");
String s2 = new String("abcd");
assertTrue("Strings not equal",
s1.equals(s2));

// Could run the test in batch mode
}
public static void main(String[] args){
junit.textui.TestRunner.run (suite ());
}
```

Behaviour Driven Development (BDD)

- BDD helps teams focus their efforts on identifying, understanding, and building valuable features that matter to businesses, and it makes sure that these features are well designed and well implemented.
- BDD practitioners use conversations around concrete examples of system behavior to help understand how features will provide value to the business.
- BDD encourages business analysts, software developers, and testers to collaborate more closely by enabling them to express requirements in a more testable way, in a form that both the development team and business stakeholders can easily understand.
- BDD tools can help turn these requirements into automated tests that help guide the developer, verify the feature, and document what the application does.
- BDD isn't a software development methodology in its own right. It's not a replacement for Scrum, XP, Kanban, RUP, or whatever methodology you're currently using. As you'll see, BDD incorporates, builds on, and enhances ideas from many of these methodologies. And no matter what methodology you're using, there are ways that BDD can help make your life easier.

From "BDD in Action" by John ferguson Smart, 2015 Published by Manning Publications

NON Behaviour Driven Development

Chris wants to add a new feature, the process goes something like this (see figure 1.1):

1 Chris tells a business analyst how he would like the feature to work.

2 The business analyst translates Chris's requests into a set of requirements for the developers, describing what the software should do. These requirements are written in English and stored in a Microsoft Word document.

3 The developer translates the requirements into code and unit tests—written in Java, C#, or some other programming language—in order to implement the new feature.

4 The tester translates the requirements in the Word document into test cases, and uses them to verify that the new feature meets the requirements.

5 Documentation engineers then translate the working software and code back into plain English technical and functional documentation.

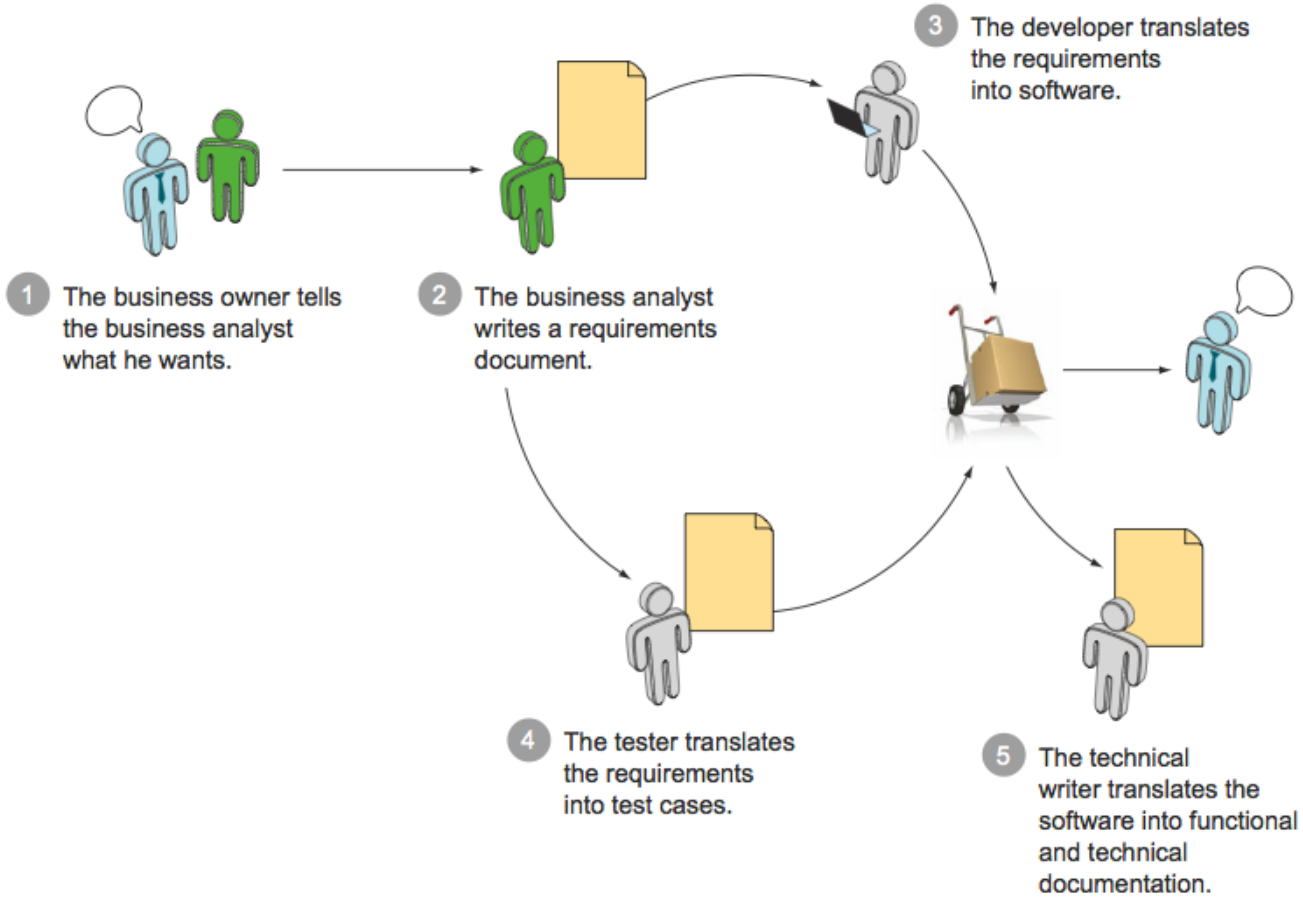


Figure 1.1 The traditional development process provides many opportunities for misunderstandings and miscommunication.

NON Behaviour Driven Development

1. Like Chris,Sarah talks to a business analyst about what she wants.To reduce the risk of misunderstandings and hidden assumptions, they talk through concrete examples of what the feature should do.

2. Before work starts on the feature, the business analyst gets together with the developer and tester who will be working on it, and they have a conversation about the feature. In this conversation, they discuss and translate key examples of how the feature should work into a set of requirements written in a structured, English-language format often referred to as Gherkin.

3. The developer uses a BDD tool to turn these requirements into a set of automated tests that run against the application code and help objectively determine when a feature is finished.

4. The tester uses the results of these tests as the starting point for manual and exploratory tests.

5. The automated tests act as low-level technical documentation, and provide up-to-date examples of how the system works. Sarah can review the test reports to see what features have been delivered, and whether they work the way she expected.

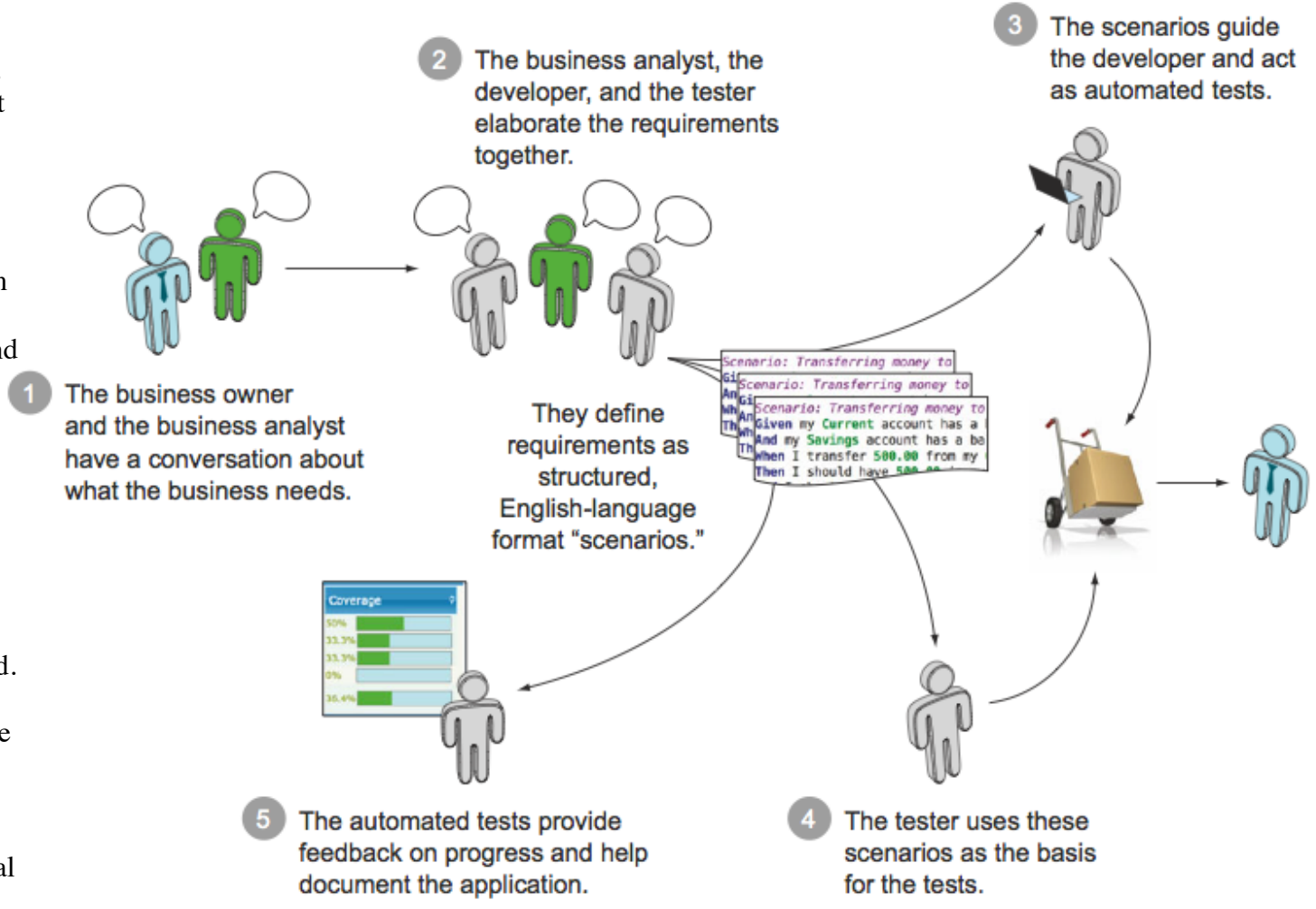


Figure 1.2 BDD uses conversations around examples, expressed In a form that can be easily automated, to reduce lost information and misunderstandings.

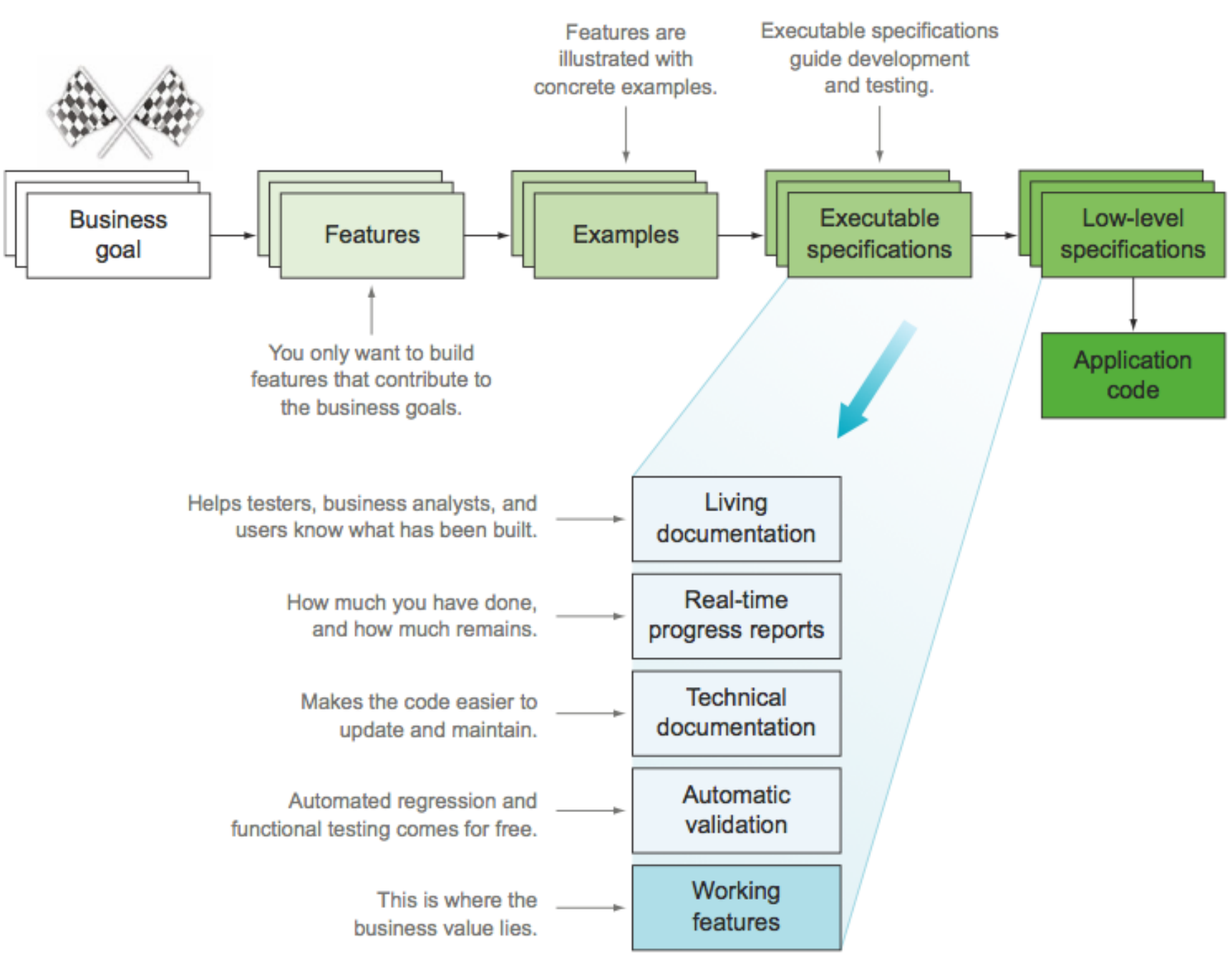


Figure 1.6 The principal activities and outcomes of BDD. Note that these activities occur repeatedly and continuously throughout the process; this isn't a single linear Waterfall-style process, but a sequence of activities that you practice for each feature you implement.

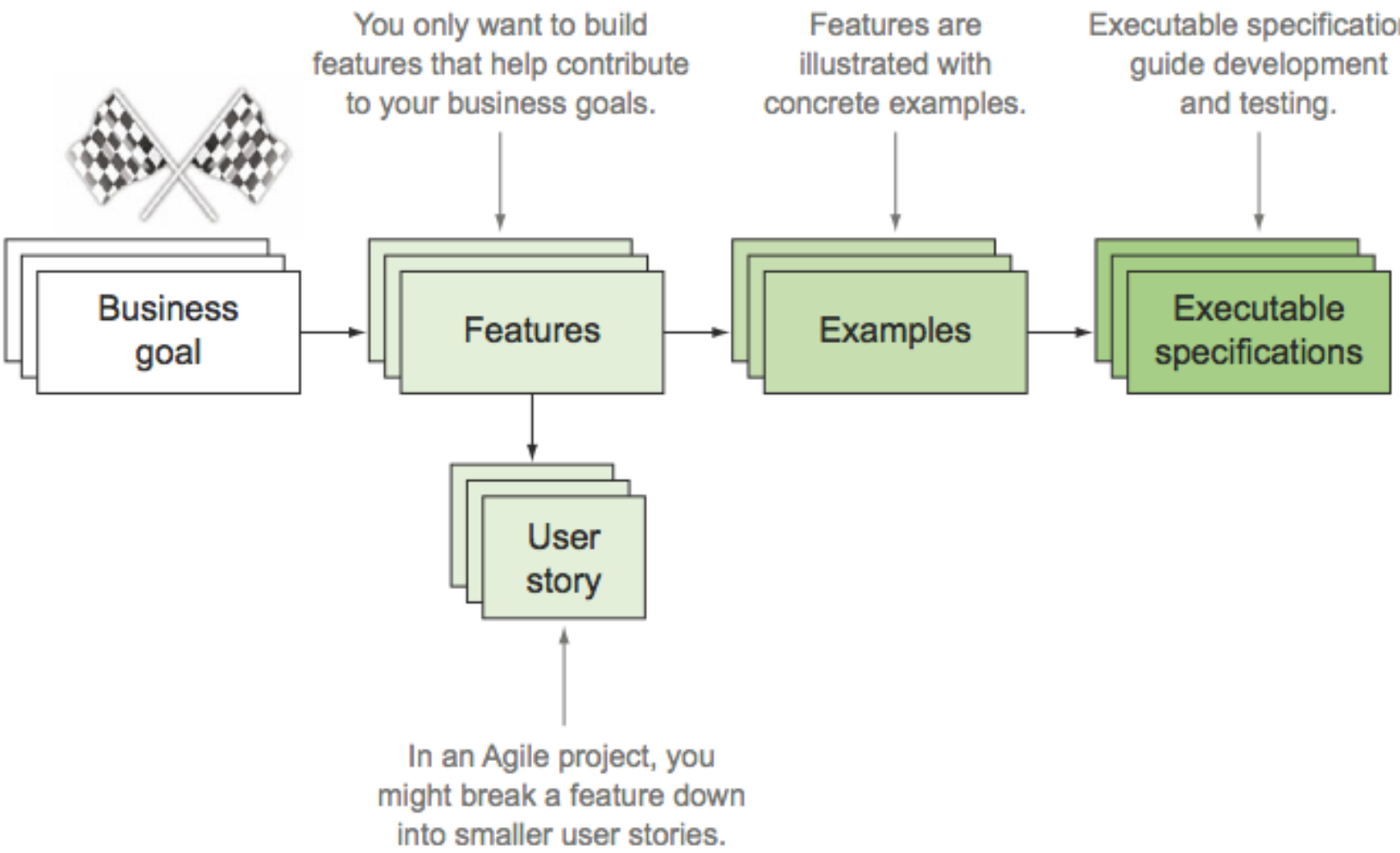


Figure 1.7 Examples play a primary role in BDD, helping everyone understand the requirements more clearly.

A Gherkin primer

Most BDD tools that we'll look at in this book use a format generally known as Gherkin, or a very close variation on this format used by JBehave.[17] This format is designed to be both easily understandable for business stakeholders and easy to automate using dedicated BDD tools such as Cucumber and JBehave. This way, it both documents your requirements and runs your automated tests.

In Gherkin, the requirements related to a particular feature are grouped into a single text file called a *feature file*. A feature file contains a short description of the feature, followed by a number of scenarios, or formalized examples of how a feature works.

```
Feature: Transferring money between accounts
  In order to manage my money more efficiently
  As a bank client
  I want to transfer funds between my accounts whenever I need to

  Scenario: Transferring money to a savings account
    Given my Current account has a balance of 1000.00
    And my Savings account has a balance of 2000.00
    When I transfer 500.00 from my Current account to my Savings account
    Then I should have 500.00 in my Current account
    And I should have 2500.00 in my Savings account
  Scenario: Transferring with insufficient funds
    Given my Current account has a balance of 1000.00
    And my Savings account has a balance of 2000.00
    When I transfer 1500.00 from my Current account to my Savings account
    Then I should receive an 'insufficient funds' error
    Then I should have 1000.00 in my Current account
    And I should have 2000.00 in my Savings account
```

As can be seen here, Gherkin requirements are expressed in plain English, but with a specific structure. Each *scenario* is made up of a number of *steps*, where each step starts with one of a small number of keywords (*Given*, *When*, *Then*, *And*, and *But*).

The natural order of a scenario is *Given* ... *When* ... *Then*:

- *Given* describes the preconditions for the scenario and prepares the test environment.
- *When* describes the action under test.
- *Then* describes the expected outcomes.

The *And* and *But* keywords can be used to join several *Given*, *When*, or *Then* steps together in a more readable way:

```
Given I have a current account with $1000
And I have a savings account with $2000
```

Several related scenarios can often be grouped into a single scenario using a table of examples. For example, the following scenario illustrates how interest is calculated on different types of accounts:

```
Scenario Outline: Earning interest
  Given I have an account of type <account-type> with a balance of
     <initial-balance>
  When the monthly interest is calculated
  Then I should have earned at an annual interest rate of <interest-rate>
  And I should have a new balance of <new-balance>
  Examples:
  | initial-balance | account-type | interest-rate | new-balance
  | 10000           | current      | 1             | 10008.33
  | 10000           | savings      | 3             | 10025
  | 10000           | supersaver   | 5             | 10041.67
```

This scenario would be run three times in all, once for each row in the Examples table. The values in each row are inserted into the placeholder variables, which are indicated by the `<...>` notation (`<account-type>`, `<initial-balance>`, and so forth). This not only saves typing, but also makes it easier to understand the whole requirement at a glance.

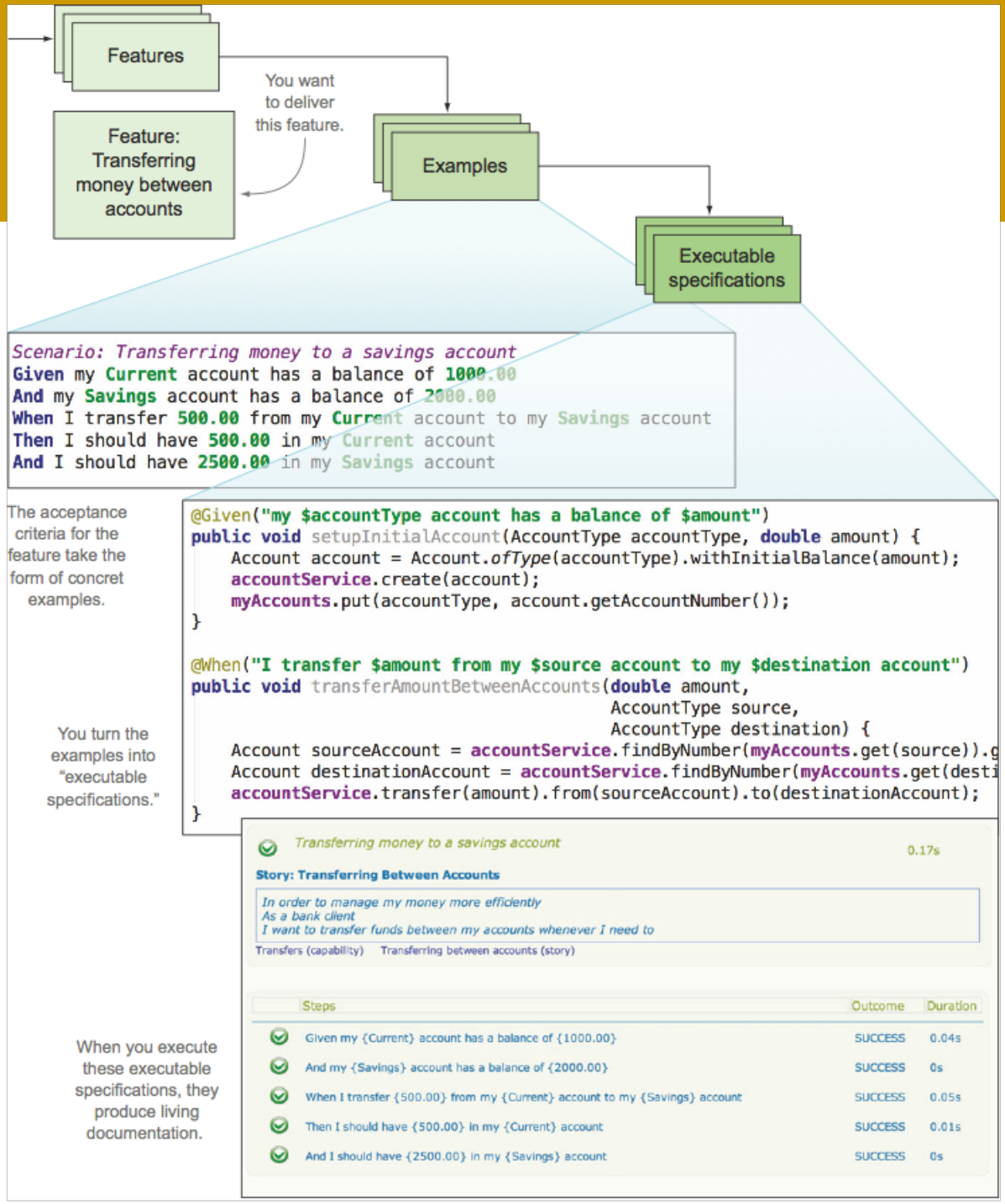


Figure 1.8 Executable specifications are expressed using a common business vocabulary that the whole team can understand. They guide development and testing activities and produce readable reports available to all.

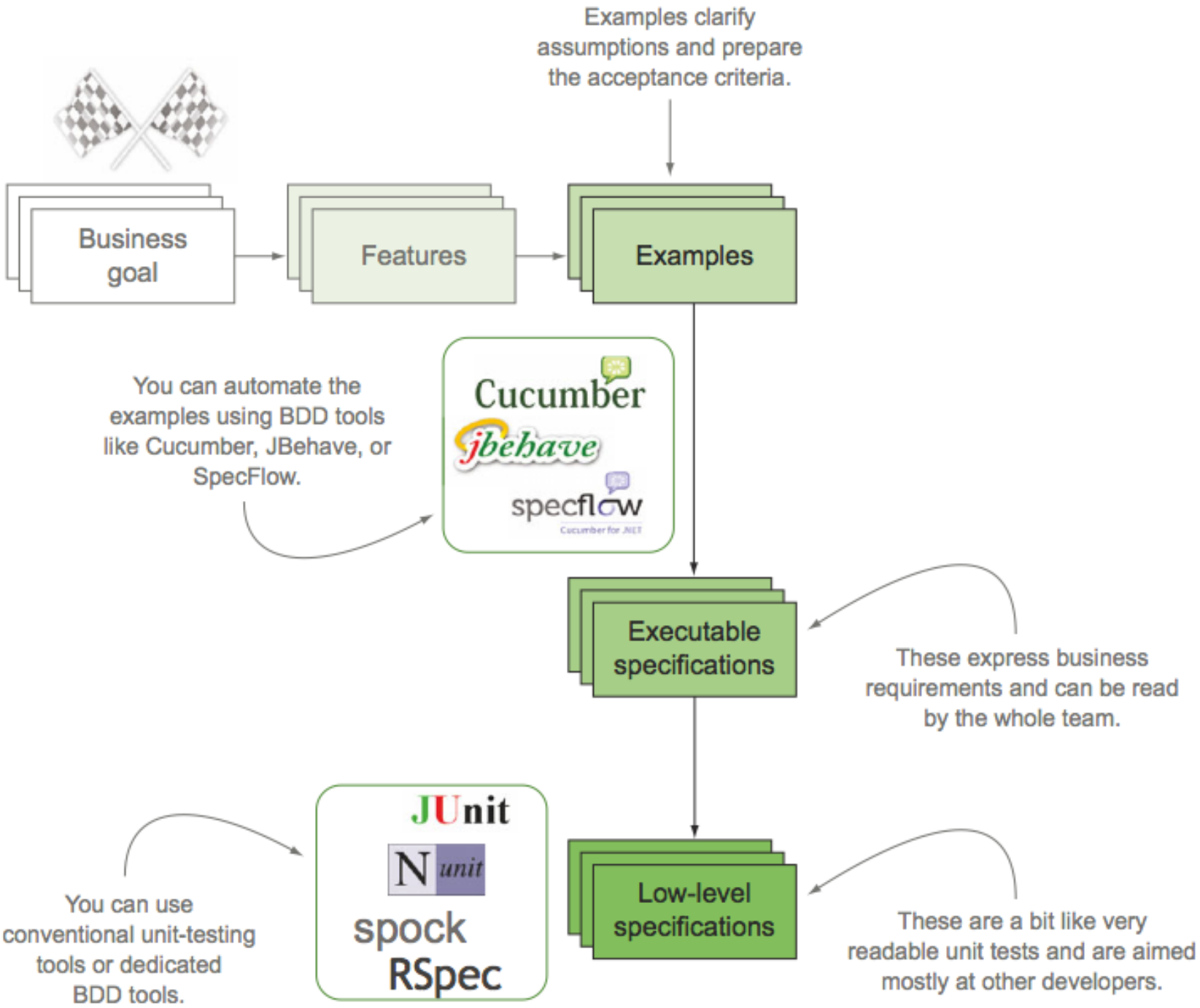


Figure 1.10 High-level and low-level executable specifications are typically implemented using different tool sets.

Specification by Example
A Love Story

Specification by Example: a Love Story

Author:	Alister Scott
Blog	watirmelon.com
Date	18 May 2011

Background Information

Janet and Dave own 'Beautiful Tea': a boutique tea estate in the Byron Bay hinterland in Australia. They grow and sell organic loose leaf tea direct to tea connoisseurs using an online ordering application developed a number of years ago. The online ordering application was originally developed and supported by an external company, but was transitioned in house a few years ago due to poor service and prohibitive cost. In transitioning it in house, Dave and Janet set up a small team to develop and support the online ordering application:

Beautiful Tea Staff

Janet and Dave: Owners & Subject Matter Experts
Henry: Customer Support
Mark: Business Analyst
Madison: Tester
Monique: Programmer

Quality & Velocity Issues

Ever since it was created, the Beautiful Tea online ordering application has been plagued with quality and velocity issues. These problems have meant not only losing money directly (for example miscalculating shipping and GST) but have also hindered business expansion plans. For example, Janet and Dave have had a lot of demand from small tea specialist retailers to resell their product, but the online ordering application is so brittle and difficult to change, they have not been able to make the necessary changes to facilitate this.

The website system documentation provided by the external company was not only sparse, but confusing to read and lacking in detail. The only truth about the system *is* the system, so this means complex system archeology is needed to solve a simple problem, which takes a large amount of Monique's time, and means less changes can be added in short periods of time.

Continuous Integration

Continuous Integration

Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly. This article is a quick overview of Continuous Integration summarizing the technique and its current usage.

01 May 2006

Martin Fowler

Translations: Portuguese Chinese Korean French Chinese

Tags: popular · agile · delivery · extreme programming · continuous integration

For more information on this, and related topics, take a look at my guide page for delivery.

ThoughtWorks, my employer, offers consulting and support around Continuous Integration. The open source CruiseControl, the first continuous integration server, was originally created at ThoughtWorks. Recently ThoughtWorks Studios, our products group, has released Go - a new server for continuous integration and delivery. It supports parallel, staged builds with multiple projects, a snazzy looking dashboard, and support for automated deployments. It's a commercial tool, but is free to use for small setups.

Contents

Enabling Agile Testing Through Continuous Integration

Sean Stolberg
Pacific Northwest National Laboratory
Sean.stolberg@pnl.gov

Abstract

A Continuous Integration system is often considered one of the key elements involved in supporting an agile software development and testing environment. As a traditional software tester transitioning to an agile development environment it became clear to me that I would need to put this essential infrastructure in place and promote improved development practices in order to make the transition to agile testing possible. This experience report discusses a continuous integration implementation I led last year. The initial motivations for implementing continuous integration are discussed and a pre and post-assessment using Martin Fowler's "Practices of Continuous Integration" is provided along with the technical specifics of the implementation. The report concludes with a retrospective of my experiences implementing and promoting continuous integration within the context of agile testing.

1. Introduction

"Hi. My name is Sean and I'm software tester. I've been waterfall-testing-free for over a year now." *(Applause is heard, hugging observed).*

Ok, hopefully you found at least a little bit of humor in that first paragraph, but I really do feel like I've been in "Waterfall Testing Rehab" for over a year now. This experience report describes significant aspects of my journey transitioning from a more traditional "software QA" role to a more effective "agile software tester". Specifically, I will share my experiences implementing a Continuous Integration system to enable the transition to agile testing techniques and approaches.

As a software tester that had worked in a traditional waterfall software development environment for eight years, I took my first job with an agile development team in November of 2007. I had no idea the changes that lay ahead of me in testing.

For my first two sprints I tried applying traditional testing approaches (test plan, clarify requirements, write test cases, test case review, etc) but kept coming up very short on time, and thus coverage, by the end of the iteration. "Technical testing debt" was accumulating in the form of manual regression tests needing to be run at the end of every sprint. We just didn't have time to run them. In summary, all of the instinctive ways I knew how to test were not holding up in the context of short iterations delivering new functionality.

The two team developers and I discussed how things were going and we agreed that I needed to find a way to insert my testing activities much earlier into the development of the sprint if we were to get the coverage needed and be able to test, find, and fix issues before the end of the sprint. We also agreed that we couldn't continue to accumulate technical testing debt in the form of manual regression tests.

Further research into agile testing techniques revealed some critical practices my team would need to implement in order to start using agile testing techniques. The most significant practices identified are listed below:

1. **Define and execute "just-enough" acceptance tests [1] -** This practice allows the customer to define external quality for the team and gives everyone confidence that user stories are complete and functional at the end of the sprint.
2. **Automate as close to 100% of the acceptance tests as possible [2] -** This practice prevents accumulation of technical testing debt in the form of an ever-growing manual regression test set that requires the team to stop and run the tests.
3. **Automate acceptance tests using a "subcutaneous" test approach with a xUnit test framework [2] -** Using an xUnit type framework and our software Application Programmer Interface (API) to automate acceptance tests allows for less-brittle test creation and easier development and

978-0-7695-3768-9/09 $25.00 © 2009 IEEE
DOI 10.1109/AGILE.2009.16