Extracted from:

The Cucumber Book, Second Edition

Behaviour-Driven Development
for Testers and Developers

This PDF file contains pages extracted from *The Cucumber Book, Second Edition*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

Copyright © 2017 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Cucumber 2.4

The Cucumber Book

Second Edition

Behaviour-Driven Development for Testers and Developers

Matt Wynne
Aslak Hellesøy
with Steve Tooke

edited by Jacquelyn Carter

The Cucumber Book, Second Edition

Behaviour-Driven Development
for Testers and Developers

Matt Wynne

Aslak Hellesøy

with Steve Tooke

The Pragmatic Bookshelf

Raleigh, North Carolina

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Jacquelyn Carter (editor)
Gilson Graphics (layout)
Janet Furlow (producer)

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Copyright © 2017 Pragmatic Programmers, LLC.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Printed in the United States of America.
ISBN-13: 978-1-68050-238-1
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—February 2017

Software starts as an idea.

Let's assume it's a good idea—an idea that could make the world a better place, or at least make someone some money. The challenge of the software developer is to take the idea and make it real, into something that actually delivers that benefit.

The original idea is perfect, beautiful. If the person who has the idea happens to be a talented software developer, then we might be in luck: the idea could be turned into working software without ever needing to be explained to anyone else. More often, though, the person with the original idea doesn't have the necessary programming skill to make it real. Now the idea has to travel from that person's mind into other people's. It needs to be *communicated*.

Most software projects involve teams of several people working collaboratively together, so high-quality communication is critical to their success. As you probably know, good communication isn't just about eloquently describing your ideas to others; you also need to solicit feedback to ensure you've been understood correctly. This is why agile software teams have learned to work in small *increments*, using the software that's built incrementally as the feedback that says to the stakeholders "Is this what you mean?"

Even this is not enough. If the developers spend a two-week iteration implementing a misunderstanding, not only have they wasted two weeks of effort, but they've corrupted the integrity of the codebase with concepts and functionality that do not reflect the original idea. Other developers may have already innocently started to build more code on top of those bad ideas, making it unlikely they'll ever completely disappear from the codebase.

We need a filter to protect our codebase from these misunderstood ideas.

Automated Acceptance Tests

The idea of *automated acceptance tests* originates in eXtreme Programming[1] (XP), specifically in the practice of Test-Driven Development[2] (TDD).

Instead of a business stakeholder passing requirements to the development team without much opportunity for feedback, the developer and stakeholder collaborate to write automated tests that express the outcome that the stakeholder wants. We call them acceptance tests because they express what the software needs to do in order for the stakeholder to find it *acceptable*. The test fails at the time of writing, because no code has been written yet, but it

1. *Extreme Programming Explained: Embrace Change [Bec00]*
2. *Test-Driven Development: By Example [Bec02]*

captures what the stakeholder cares about and gives everyone a clear signal as to what it will take to be *done*.

These tests are different from *unit tests*, which are aimed at developers and help them to drive out and check their software designs. It's sometimes said that unit tests ensure you *build the thing right*, while acceptance tests ensure you *build the right thing*.

Automated acceptance testing has been an established practice among good XP teams for years, but many less experienced agile teams seem to see TDD as being a programmer activity only. As Lisa Crispin and Janet Gregory point out in *Agile Testing: A Practical Guide for Testers and Agile Teams [CG08]*, without the business-facing automated acceptance tests, it's hard for the programmers to know which unit tests they need to write. Automated acceptance tests help your team to focus, ensuring the work you do each iteration is the most valuable thing you could possibly be doing. You'll still make mistakes—but you'll make a lot less of them—meaning you can go home on time and enjoy the rest of your life.

Behaviour-Driven Development

Behaviour-Driven Development (BDD) builds upon Test-Driven Development (TDD) by formalizing the good habits of the best TDD practitioners. The best TDD practitioners work from the *outside-in*, starting with a failing customer acceptance test that describes the behavior of the system from the customer's point of view. As BDD practitioners, we take care to write the acceptance tests as *examples* that anyone on the team can read. We make use of the process of writing those examples to get feedback from the business stakeholders about whether we're setting out to build the right thing before we get started. As we do so, we make a deliberate effort to develop a shared, *ubiquitous language* for talking about the system.

Ubiquitous Language

As Eric Evans describes in his book *Domain Driven Design [Eva03]*, many software projects suffer from low-quality communication between the domain experts and programmers on the team:

> "A project faces serious problems when its language is fractured. Domain experts use their jargon while technical team members have their own language tuned for discussing the domain in terms of design... Across this linguistic divide, the domain experts vaguely describe what they want. Developers, struggling to understand a domain new to them, vaguely understand."

With a conscious effort by the team, a ubiquitous language can emerge that is used and understood by everyone involved in the project. When the team uses this language consistently in their conversations, documentation, and code, the friction of translating between everyone's different little dialects is gone, and the chances of misunderstandings are greatly reduced.

Cucumber helps facilitate the discovery and use of a ubiquitous language within the team, by giving the two sides of the linguistic divide a place where they can meet. Cucumber tests interact directly with the developers' code, but they're written in a medium and language that business stakeholders can understand. By working together to write these tests—*specifying collaboratively*—not only do the team members decide what behavior they need to implement next, but they learn how to describe that behavior in a common language that everyone understands.

When we write these tests before development starts, we can explore and eradicate many misunderstandings long before they ooze their way into the codebase.

Examples

What makes Cucumber stand out from the crowd of other testing tools is that it has been designed specifically to ensure the acceptance tests can easily be read—and written—by anyone on the team. This reveals the true value of acceptance tests: as a communication and collaboration tool. The easy readability of Cucumber tests draws business stakeholders into the process, helping you really explore and understand their requirements.

Here's an example of a Cucumber acceptance test:

```
Feature: Sign up

  Sign up should be quick and friendly.

  Scenario: Successful sign up

    New users should get a confirmation email and be greeted
    personally by the site once signed in.

    Given I have chosen to sign up
    When I sign up with valid details
    Then I should receive a confirmation email
    And I should see a personalized greeting message

  Scenario: Duplicate email

    Where someone tries to create an account for an email address
    that already exists.

    Given I have chosen to sign up
    But I enter an email address that has already registered
    Then I should be told that the email is already registered
    And I should be offered the option to recover my password
```

Notice how the test is specified as *examples* of the way we want the system to behave in particular scenarios. Using examples like this has an unexpectedly powerful effect in enabling people to visualize the system before it has been built. Anyone on the team can read a test like this and tell you whether it reflects their understanding of what the system should do, and it may well spark their imagination into thinking of other scenarios that you'll need to consider too. Gojko Adzic's book *Specification by Example [Adž11]* contains many case studies of teams who have discovered this and used it to their advantage.

Acceptance tests written in this style become more than just tests; they are *executable specifications*.