

The Addison-Wesley Signature Series



A KENT BECK SIGNATURE
BOOK A
Kent Beck
SIGNATURE

ATDD BY EXAMPLE

A PRACTICAL GUIDE TO ACCEPTANCE
TEST-DRIVEN DEVELOPMENT

Markus Gartner



Forewords by Kent Beck and Dale Emery

ATDD by Example

The Addison-Wesley Signature Series

Kent Beck, Mike Cohn, and Martin Fowler, Consulting Editors



▲ Addison-Wesley

Visit informit.com/awss for a complete list of available products.

The Addison-Wesley Signature Series provides readers with practical and authoritative information on the latest trends in modern technology for computer professionals. The series is based on one simple premise: Great books come from great authors. Titles in the series are personally chosen by expert advisors, world-class authors in their own right. These experts are proud to put their signatures on the covers, and their signatures ensure that these thought leaders have worked closely with authors to define topic coverage, book scope, critical content, and overall uniqueness. The expert signatures also symbolize a promise to our readers: You are reading a future classic.



Make sure to connect with us!
informit.com/socialconnect



informIT.com
THE TRUSTED TECHNOLOGY LEARNING SOURCE

Safari
Books Online

ATDD by Example

A Practical Guide to Acceptance
Test-Driven Development

Markus Gärtner

◆◆ Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

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 publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales
international@pearson.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data

Gärtner, Markus, 1979-

ATDD by example / Markus Gärtner.

p. cm.

Includes bibliographical references and index.

ISBN-10: 0-321-78415-4 (pbk. : alk. paper)

ISBN-13: 978-0-321-78415-5 (pbk. : alk. paper)

1. Agile software development - Case studies. 2. Automation. 3. Systems engineering. I. Title.

QA76.76.D47G374 2013

005.1-dc23

2012016163

Copyright © 2013 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.

ISBN-13: 978-0-321-78415-5

ISBN-10: 0-321-78415-4

Text printed in the United States on recycled paper at Courier in Westford, Massachusetts.

First printing, July 2012

*To my wife Jennifer, my pet-son Leon, and our daughter Katrin,
who allowed me to spend way too little time
with them while writing this.*

This page intentionally left blank

Contents

Foreword by Kent Beck xi

Foreword by Dale Emery xiii

Preface xv

Acknowledgments xxi

About the Author xxiii

Part I Airport Parking Lot 1

1 Parking Cost Calculator Workshop 3

- Valet Parking 3
- Short-Term Parking 5
- Economy and Long-Term Parking 6
- Essential Examples 9
- Summary 12

2 Valet Parking Automation 17

- The First Example 18
- Pairing for the First Test 25
 - Initializers 26
 - Checking the Results 31
- Tabulated Tests 36
- Summary 39

3 Automating the Remaining Parking Lots 41

- Short-Term Parking Lot 41
- Economy Parking Lot 44
- Summary 46

4	Wish and Collaborate	47
	Specification Workshops	48
	Wishful Thinking	49
	Collaboration	50
	Summary	52
Part II	Traffic Light Software System	53
5	Getting Started	55
	Traffic Lights	55
	FitNesse	58
	Supporting Code	59
	Summary	60
6	Light States	61
	State Specifications	61
	The First Test	62
	Diving into the Code	66
	Refactoring	70
	Packages	71
	The LightState Enum	71
	Editing LightStates	78
	Summary	90
7	First Crossing	93
	Controller Specifications	93
	Driving the Controller	94
	Refactoring	103
	Summary	118
8	Discover and Explore	119
	Discover the Domain	120
	Drive the Production Code	121
	Test Your Glue Code	122

Value Your Glue Code	124
Summary	125

Part III Principles of Acceptance Test-Driven Development 127

9 Use Examples 129

Use a Proper Format	130
Behavior-Driven Development	132
Tabulated Formats	133
Keyword-Driven Automation	137
Glue Code and Support Code	139
The Right Format	140
Refine the Examples	142
Domain Testing	143
Boundary Values	144
Pairwise Testing	145
Cut Examples	146
Consider Gaps	149
Build Your Testing Orchestra	150
Summary	151

10 Specify Collaboratively 153

Meet the Power of Three	153
Hold Workshops	155
Participants	156
Goal of the Workshop	156
Frequency and Duration	157
Trawl Requirements	158
Summary	159

11 Automate Literally 161

Use Friendly Automation	162
Collaborate on Automation	164
Discover the Domain	166
Summary	167

12 Test Cleanly	169
Develop Test Automation	170
Listen to the Tests	172
Refactor Tests	176
Extract Variable	178
Extract Keyword	179
Summary	180
13 Successful ATDD	183
Appendix A Cucumber	187
Feature Files	187
Step Definitions	188
Production Code	189
Appendix B FitNesse	191
Wiki Structure	191
SLiM Tables	192
Support Code	193
Appendix C Robot Framework	195
Sections	195
Library Code	199
<i>References</i>	201
<i>Index</i>	205

Foreword

by Kent Beck

There is a curious symmetry to the way this book presents Acceptance Test-Driven Development and the way software is developed with ATDD. Just as there is an art to picking the specific examples of program behavior that will elicit the correct general behavior for the system, there is an art to picking specific examples of a programming technique like ATDD to give you, the reader, a chance to learn the technique for yourself. Markus has done an admirable job in selecting and presenting examples.

To read this book you will need to read code. If you follow along, you will have the opportunity to learn the shift in thinking that is required to succeed with ATDD. That shift is, in short, to quickly go from, “Here’s a feature I’d like,” to “How are we going to test that? Here’s an example.” Reading the examples, you will see, over and over, what that transition looks like in various contexts.

What I like about this code-centric presentation is the trust it shows in your powers of learning. This isn’t “12 Simple Rules for Testing Your Web App” printed on intellectual tissue paper that falls apart at first contact with the moisture of reality. Here you will read about concrete decisions made in concrete contexts, decisions that you could (and that, if you want to get the most out of this book, you will) disagree with, debate, and decide for yourself.

The latter portions of the book do draw general conclusions, summarizing the principles at work in the examples. If you are someone who learns more efficiently when you are familiar with general concepts, that will be a good place to start. Regardless, what you get out of this book is directly proportional to the investment you are willing to make in following the examples.

One of the weaknesses of TDD as originally described is that it can devolve into a programmer’s technique used to meet a programmer’s needs. Some programmers

take a broader view of TDD, facily shifting between levels of abstraction for their tests. However, with ATDD there is no ambiguity—this is a technique for enhancing communication with people for whom programming languages are foreign. The quality of our relationships, and the communication that underlies those relationships, encourages effective software development. ATDD can be used to take a step in the direction of clearer communication, and *ATDD by Example* is a thorough, approachable introduction.

—Kent Beck

Foreword

by Dale Emery

Too many software projects fail to deliver what their customers request. Over the years, I've heard scores of project customers explain the failures: *The developers don't pay attention to what we ask them to build.* And I've heard hundreds of developers explain the failures: *The customers don't tell us what they want. Most of the time they don't even know what they want.*

I've observed enough projects to come to a different conclusion: Describing a software system's responsibilities is hard. It requires speaking and listening with precision that is rare—and rarely so necessary—in normal human interactions. Writing good software is hard. Testing software well is hard. But the hardest job in software is communicating clearly about what we want the system to do.

Acceptance Test-Driven Development (ATDD) helps with the challenge. Using ATDD, the whole team collaborates to gain clarity and shared understanding before development begins. At the heart of ATDD are two key practices: Before implementing each feature, team members collaborate to create concrete examples of the feature in action. Then the team translates these examples into automated acceptance tests. These examples and tests become a prominent part of the team's shared, precise description of “done” for each feature.

What is shared understanding worth? One developer at an ATDD workshop explained it this way: “Once we started to work together to create examples, I started to *care* about the work we were doing. I finally understood what we were building and why. Even more importantly, I knew that the whole team understood what we were trying to accomplish. Suddenly we all had the same goal—we were all on the same team.”

ATDD helps us not only to know when we're done, but also to know when we're making progress. As we automate each test and write the software that passes

the test (and all of the previous tests), the examples serve as signposts along the road to completion. And because each example describes a responsibility that customers value, we can have confidence that not only are we making progress, we're making progress that matters.

Okay, I've listed a few of ATDD's key features and a few of its key benefits. That's the easy part. As for the heavy lifting: How do you actually do this stuff so that it works in the real world? I'll leave that to Markus Gärtner. In *ATDD by Example*, Markus rolls up his sleeves and not only tells you but *shows* you how ATDD works in practice. He lets you peek over the shoulders and into the minds of testers, programmers, and business experts as they apply the principles and practices of ATDD.

I offer one caveat as you read this book: The first few chapters—in which we follow business expert Bill, tester Tony, and programmers Phyllis and Alex as they describe and implement a small software system—may seem at first glance to be overly simple, or even simplistic. Don't be fooled by that appearance. There is a *lot* going on in these chapters. This is a skilled team, and some of their skills are subtle. Notice, for example, that in the requirements workshop the team members avoid any mention of technology. They focus entirely on the system's business responsibilities. And notice that as Alex and Tony automate the first few tests, Tony makes good use of his *lack* of programming experience. Whenever he is confused by some technical detail, he asks Alex to explain, and then works with Alex to edit the code so that the code explains itself. And notice how frequently Alex insists on checking the tests into the source control system—but only when the code is working. If you're new to ATDD, these skills may not be obvious, but they're essential to success.

Fortunately, all you need to do to learn about these subtle skills is to keep reading. Markus pauses frequently to explain what the team is doing and why. At the end of each chapter he summarizes how the team worked together, what they were thinking, and the practices they applied. And in the final portion of the book, Markus brings it all together by describing in detail the principles that make ATDD work.

ATDD by Example is a great introduction to Acceptance Test-Driven Development. It also offers a fresh perspective for people like me who have been practicing ATDD for a while. Finally, it is a book that rewards multiple readings. So read, practice, and read again. You'll learn something new and useful each time.

—Dale Emery

Preface

In this book I give an entry-level introduction to the practice that has become known as Acceptance Test-Driven Development—or ATDD. When I first came across the term ATDD in 2008, I assumed that it was artificial and unnecessary. It seemed superfluous to me as I had learned test-driven development in 2008 and found it sufficient. In the end, why would I need to test for acceptance criteria?

“Time wounds all heels” [Wei86]. So, four years later I find myself writing a book on what has become known as Acceptance Test-Driven Development. Throughout 2009 I ran into Gojko Adzic, who had just finished his book *Bridging the Communication Gap* [Adz09]. He gave me a copy of that book, and I immediately started to read it on my way back from London. Once I had finished it, I had a good understanding about what ATDD is and why we should avoid that name.

But why did I still use the name *ATDD by Example* for the paper stack you hold in your hands?¹

On the Name

ATDD has been around for some time now. It is known by different terms. Here is an incomplete list:

- Acceptance Test-Driven Development
- Behavior-Driven Development (BDD)
- Specification by Example
- Agile Acceptance Testing
- Story Testing

1. Or, why did I use the particular arrangement of 1s and 0s that displays as “ATDD by Example” on your electronic device?

From my perspective, any of these names comes with a drawback. Acceptance Test-Driven Development creates the notion that we are finished with the iteration once the acceptance tests pass. This is not true, because with any selection of tests, the coverage is incomplete. There are gaps in the net of tests. In the testing world, this is well known as the impossibility to test everything. Instead we know exactly we are not finished when an acceptance test fails—as Michael Bolton put it.

Despite arguing for one name or another, I decided to put a selection of possible alternatives here and have the readers decide which fits best their need. In the end it does not matter to me what you call it, as long as it's working for you. The world of software development is full of misleading terms and probably will stay so for some more years. Software engineering, test automation, test-driven development are all misleading in one way or another. As with any abstraction, don't confuse the name for the thing. The expert knows the limitations of the name of the approach.

But why have there been different names for a similar approach? The practices you use may very well differ. Having visited and consulted multiple teams in multiple companies on ATDD, they all have one thing in common: Each team is different from the others. While one practice might work for your team in your current company, it might fail dramatically in another. Have you ever wondered about the answer “it depends” from a consultant? This is the source of it.

For his book *Specification by Example* [Adz11], Gojko Adzic interviewed more than fifty teams that apply ATDD in one form or another. What he found is a variety of practices accompanying the ATDD approach. All of the teams that apply ATDD successfully start with a basic approach, then revisit it after some time, and adapt some changes in order to fit their particular context. Starting with a lightweight process and adapting new things as you find problems is a very agile way of implementing any approach. As you apply ATDD, keep in mind that your first set of practices is unlikely to solve all your problems. Over time you will adapt the solution process as you gain more and more experience.

Why Another Book on ATDD?

While Gojko describes many patterns of successful ATDD implementations, I found there is a major gap in the books on ATDD up until now. There is a considerable difference between advanced adopters of a skill or approach and entry-level demands for the same skill or approach.

When going through the literature on ATDD, I found several books that explain ATDD on an advanced level by referring to principles. For an advanced learner, it is easy to apply principles in their particular context. However, this does

not hold for a novice on the same topic. A novice needs more concrete directions in order to get started. Once a person gains experience with the basics, he or she can start to break free from the hard constraints of the approach.

Novices learn best by following a recipe, but by no means is this book a cookbook on ATDD. With the examples in this book, I provide two working approaches to ATDD and expose the thought processes of the people involved. The novice learner can use these to get started with ATDD on her team. As we go along, I provide pointers to more in-depth material.

The basic idea is taken from Kent Beck's *Test-Driven Development: By Example* [Bec02]. Beck provides two working examples on Test-Driven Development and explains some of the principles behind it in the end. It is intended as an entry-level description of TDD and provides the novice with enough learning material to get started—assuming that through reflection and practice TDD can be learned. The same holds true to some degree for this book as well.

Vocabulary

Throughout the book I will use several terms from the Agile software development world. Realizing that not everyone knows about Agile software development, a brief introduction of some terms is in place.

Product Owner In the Agile method Scrum three roles are defined: the development team, the ScrumMaster, and the Product Owner. The Product Owner is responsible for the success of the product that the team will build. He or she sets priorities for the features that the team will be implementing and works together with other stakeholders to derive them. He or she is also the customer representative for the team and decides about details in that function—and has to negotiate with the other stakeholders about this.

Iteration, or Sprint Agile development relies on a regular cycle called the iteration or Sprint in Scrum. These are short bursts where the team implements a single product increment that is potentially shippable. Common iteration lengths vary between one and four weeks.

User Story A user story is a limited set of functionality that the team feels comfortable implementing over the course of a single iteration. These are tiny slices through the functionality. Usually a team strives to implement several user stories in one iteration. The business representative or product owner is responsible for defining these stories.

Taskboard Most Agile teams plan their work on a board visually accessible to anyone. They use cards to indicate what they are working on. The taskboard

usually has several columns, at least `ToDo`, `Doing`, and `Done`. As the work proceeds, the team updates the taskboard to reflect this.

Story Card User stories are usually written on real cards. During the iteration, the cards are put onto the team's taskboard.

Standup Meeting, Daily Scrum At least once per day team members update themselves on the current state of the iteration. The team gets together for 15 minutes and discusses how they can finish currently open tasks until the end of the iteration.

Product Backlog, Sprint Backlog The Product Owner in Scrum organizes unimplemented stories in a product backlog. He or she is responsible for updating the backlog whenever new requirements enter. When the team gets together to plan the next sprint, the team members identify a backlog for the next sprint length. This is called the Sprint Backlog. The selected stories from the Product Backlog automatically become part of the Sprint Backlog. The Sprint Backlog is most often organized on the taskboard after the planning meeting.

Refactoring Refactoring is changing the structure of the source code without changing what it does. Usually I refactor code before introducing changes. By refactoring my code I make the task of implementing the upcoming changes more easy.

Test-Driven Development (TDD) In test-driven development you write one single test that fails, write just enough code that makes this failing test pass (and all the other passing tests still pass), and then refactor your code to prepare it for the next tiny step. TDD is a design approach, and it helps users write better code, because testable code is written by default.

Continuous Integration (CI) In Continuous Integration you integrate the changes in the source code often. A build server then builds the whole branch, executes all unit tests and all acceptance tests, and spreads the information about this build to your colleagues. CI relies on an automated build, and it helps teams to see problems with the current state of the branch very early—not just one hour before the release shall be shipped.

How to Read This Book

In this book I provide a mixture of concrete practices alongside some of the principles that I found useful. There are multiple ways to read this book—depending on your experience level you may pick any of them.

You may read this book cover to cover. You will get to know more about Cucumber, Behavior-Driven Development and how to test webpages using an ATDD tool. The first example is also based on a team that differentiates between testing experts and programming experts. You will find collaboration as one key success factor there.

In the second part I will pair up with you. By pairing up we can compensate for any missing testing or programming knowledge at this point. We will drive our application code using ATDD in a practical way. We will deal with FitNesse, a wiki-based acceptance test framework. The examples in the second part are covered in Java.

In the third part you will find some guidance on how to get started with the approach. I give pointers to further readings as well as hints on how to get started, what worked well, and what did not work so well for other teams.

In the appendixes you will find the two tools used in this book and even a third one explained in some depth to get you started. If you haven't run into Cucumber or FitNesse, you may want to start there.

An advanced-level reader might skip the first two parts initially and directly start with the principles I explain in the third part. Maybe you want to provide some background to your colleagues later. The examples in Parts I and II serve this purpose.

You may also read the first two examples, and then head back to work to start a basic implementation. Once you reach a dead end, you may come back to read further material in Part III—although I wouldn't necessarily recommend reading this book in this order.

If you already have an ATDD implementation in place on your team, you may want to dig deeper in Part II where I explain how to drive the domain code from your examples.

These are some ways in which I can imagine reading this book. If you're like me, you're probably thinking of following the examples by implementing the provided code on your own. I set up a github repository for each of the code examples. These allowed me to acceptance test the code examples on my own. If you find yourself stuck, you can have a peek there as well. You will find the examples for the first part at <http://github.com/mgaertne/airport>, and the sources for the second part at <http://github.com/mgaertne/trafficlights>.

This page intentionally left blank

Acknowledgments

A project such as this book would not be possible without the support of so many helpers. First of all, I would like to thank Dale Emery, who provided me great comments on my writing style. Being a non-native English writer, I really appreciated the feedback I got from Dale.

A special thank you goes to Kent Beck. In August 2010 I approached him on the topic of writing a book on ATDD following the approach he used in *TDD by Example*. He also introduced me to Addison-Wesley and to Christopher Guzikowski, who provided me all the support to get this book published.

Several people have provided me feedback on early drafts. For this feedback I thank Lisa Crispin, Matt Heusser, Elisabeth Hendrickson, Brett Schuchert, Gojko Adzic, George Dinwiddie, Kevin Bodie, Olaf Lewitz, Manuel Küblböck, Andreas Havenstein, Sebastian Sanitz, Meike Mertsch, Gregor Gramlich, and Stephan Kämper.

Last, but not least, I would like to thank my wife Jennifer and our children Katrin and Leon for their support while writing this book. I hope to be able to return the time you had to deal without a husband or a dad in the years to come.

This page intentionally left blank

About the Author



Markus Gärtner works as an Agile tester, trainer, coach, and consultant with it-agile GmbH, Hamburg, Germany. Markus, a student of the work of Jerry Weinberg, founded the German Agile Testing and Exploratory workshop in 2011 and is one of the founders of the European chapter of Weekend Testing. He is a black-belt instructor in the Miagi-Do school of Software Testing and contributes to the Agile Alliance FTT-Patterns writing community, as well as the Software Craftsmanship movement. Markus regularly presents at Agile and testing conferences all over the globe, as well as dedicating himself to writing about testing, foremost in an Agile context. He maintains a personal blog at shino.de/blog. He teaches ATDD

and context-driven testing to customers in the Agile world. He has taught ATDD to testers with a nontechnical background, as well as to several programmers.

This page intentionally left blank

Wish and Collaborate

After this short iteration we will take a step back and reflect briefly. The functionality of the airport parking lot calculator was developed. Before the iteration started, the team discussed the requirements for the application that they should build. The format they used was a specification workshop [Adz09, Adz11]. The team identified different parking lots, and in that conversation they noted examples for different parking durations and the costs respectively.

After the examples were clear to the team, they started to work on the functionality. This particular team seems to work in parallel on coding and testing tasks. The tester wrote down the first example for the automation. After that he worked his way through the framework until he got stuck with the automation. You may recall that Tony started with a happy path example. When automating your examples, this is essential because it forces you to get the implementation correct right from the start before fiddling with too many details and corner conditions. The first example will provide insights about the necessary user interface for the end user. Starting from this basis, you can extend the examples in multiple directions. It does not really matter which happy path example you start with, if you apply responsive design techniques and object-oriented design. In this example there wasn't much magic happening to the automation code, but there are some possible evolution points for this code. For one example all the dates seem to cluster around one particular date. In a future version of the automation code you may want to vary this, maybe leaving the calculation of particular durations to a helper class like a `DurationFactory`, which calculates randomized starting dates.

One important thing happened when Tony got up and walked over to a programmer, maybe the most important thing about successful test automation. A tester and a programmer collaborating in order to achieve the team goal of test automation provides great value when introducing the ATDD approach. Despite leaving Tony alone with the task to automate the tests, Alex offers him full support for the first test. Tony learned from Alex some of the underlying design principles and how to treat code to keep it readable and maintainable. Over time Tony got

more and more familiar with test automation code. This enabled him to proceed with the automation code.

Let's take a look at each of the three elements we saw here: specification workshops, wishful thinking, and collaboration.

Specification Workshops

In specification workshops teams discuss the stories for upcoming iterations. At first specification workshops appeared to me as a waterfall approach to requirements. Programmers and testers get together with some business representative to nail down requirements. But there are more benefits for agile teams to hold these workshops.

Getting everyone involved helps build a common language for the project. Eric Evans calls this language the ubiquitous language [Eva03]. When programmers, testers, and business people get their heads together to reach a common understanding about the project, they can sort out many misunderstandings before these blow up the whole project.

A workshop can help the whole team reach that shared understanding. There are some things you will have to keep in mind in order to make these workshops a success for everyone—regardless of whether they may be participating.

First of all, you should not waste the time of your business representatives. If you invite an expert user to your specification workshop, everyone in the room should respect the precious time of this person. A business representative could be a ProductOwner, a real user, or a subject matter expert for the application domain. If your team starts to discuss the latest technology at such a workshop, the business representative is probably going to reject your invitation the next time. At that point you will have lost crucial information for your application.

Pre-select some stories from your backlog. If you know which stories you will most likely implement in the near future, you can sort them out. If you end up with a list of stories that is probably too large to discuss in your allotted time for the workshop, then you have to cut it further.

For stories where the business flow seems obvious or straightforward to you, you can prepare data and bring those to the workshop. The business representative will value your engagement in the project and the easier stories. By preparing examples, you will also help keep the businessperson engaged and seeing the advantages of these workshops.

During the workshop it is crucial to ask clarifying questions. You can prepare yourself by going through the stories with your team and collecting open questions for the moment. Over time you may gain experience to come up more spontaneously

with clarifying questions, but initially you may need full team feedback for the stories.

Finally, one element I consider mandatory for any meeting or workshop is visualization. Rather than leaving the discussion abstract, note down what you understand and ask for agreement based on your notes. You can do this publicly on a flipchart, or take notes on paper and share them around the table. For larger meetings I prefer flipcharts, while in a setting of three participants as in this first example, a piece of paper will suffice.

If your customer is located in a completely different country or timezone, you may want to try a different multimedia setting. With instant messaging and screen-sharing tools around, you can easily collaborate even if you are not in the same room with the whole team. However, you should set some preparation time aside to get these tools set up before the meeting.

Wishful Thinking

A vital implementation of acceptance test-driven development includes at least two spoonfuls of wishful thinking. In the example at the Major International Airport Corp. we saw Tony implementing the tests without any previous knowledge about details of the parking cost calculator.

Instead, Tony applied wishful thinking in order to automate the examples that the team had identified in the workshop. Tony avoided considering the available user interface. Instead, he used the interface he wished he would have. The examples clearly stated that there are different durations to be considered for different parking costs. The entry and exit dates did not play a role when writing down the examples with the business expert. Tony didn't clutter up his examples with these unnecessary details.

Instead of programming against a real user interface, abstract from the GUI to the business cases behind your examples. As Tony demonstrated, consider that you could have any interface for your tests. Dale Emery recommended writing your tests as if you already have the interface you wish you had. Use the most readable interface to automate your examples. If you hook your automation code to the application under test, you may find out that you have to write a lot of code to get the application automated. If you listen to your tests [FP09], you will find that your application needs a different interface—at least for your automated tests.

Wishful thinking is especially powerful if you can apply it before any code is written. At the time you start implementing your production code, you can discover the interface your application needs in order to be testable. In our example, we saw that Tony and Alex started their work in parallel. The interface that Alex designed

is sufficient for the discussed examples, but the lack of input parking durations directly forces the need for more test automation code.

The translation between parking durations and entry and exit dates and times is simple in this example. You may have noticed that all the examples start on the same date. Most testers and programmers faced with these hard-coded values feel uneasy about it. While it takes little effort to generate parking duration on the fly while the tests execute, the amount and complexity of support code would rise. As a software developer, I would love to write unit tests for this complex code and drive the implementation of the support code using test-driven development.

The translation between durations, entry and exit dates and times is an early sign that something might be wrong. Maybe the user interface is wrong. But as a customer at an airport, I would probably like to input my departure and arrival dates and times. So, the user interface seems to be correct based on the goal of the potential customers.

Another option could be that the tests point to a missing separation of concerns. Currently, the calculator calculates the parking duration first, and after that the parking costs. The cost calculation could be extracted from the code, so that it becomes testable separately without the need to drive the examples through the user interface.

In the end, your tests make suggestions for your interface design. This applies to unit tests as well as acceptance tests. When testers and programmers work in isolation, a more problematic interface for test automation can manifest itself than when both programmers and testers work together on that problem.

Collaboration

In the story of the Major International Airport Corp. we saw collaboration on multiple levels. Tony, the tester, joined the workshop together with Bill, the business expert, and Phyllis, the programmer. Later, while automating the examples they had identified in the workshop, Tony worked together with Alex.

Collaboration is another key ingredient to a successful ATDD approach. Consider what would happen if Tony worked out the examples by himself. He probably could have caught many problems within the software. These defects would have been bounced back and forth between Tony and the programmers—eventually getting both upset. In the end, when the product finally was delivered, the customer would have been unhappy about the misinterpreted corner conditions.

If this sounds familiar to you, consider a project that starts with a workshop. In this workshop most ambiguities would be settled between the programmers and the testers. The remaining questions would get answered before the team starts

to work on the implementation. Since the examples express the requirements for the software, the team knows exactly when it has finished the implementation. There is some back and forth between testers and programmers. The programmers eventually find out about the value the automated examples bring them if they execute them before checking in their code to the version control system. In the end, the project delivers on time and with no problems.

To most teams new to acceptance test-driven development this may sound like a fairy tale. But there are many success stories of successful software delivery using an approach like ATDD in combination with other agile practices like refactoring, test-driven development (TDD), continuous integration, and the whole team approach. The combination of technical excellence on one hand and teamwork on the other hand seems to be a magic ingredient.

I also apply collaboration when automating tests. After all, test automation is software development and therefore, I want to apply all the practices and techniques that I also apply to production code. Most of the time I even take more care implementing support code for my tests than I take care for the production code. This means that I apply test-driven development, refactoring, and continuous integration to the support code as well.

Tony worked with Cucumber before he could get started with the support code. But he clearly did not have the expertise to finish the test automation code all on his own. When he noticed that he was stuck, he stopped work, and approached that team member that could help him and had the expertise with programming. Most teams new to ATDD confuse the collaboration aspect with the need for every tester to code. It makes the life of testers easier if they can work independently from programmers on tests and test automation, though. That's why over time testers start to learn more and more tricks to automate their tests, but this is not a precondition. It is rather an outcome and a side effect in the long term.

Once I taught the approach to testers at a medical supplier. The testers were former nurses and had no technical education at all. Up to that point they tested the application manually. The programmers pushed forward for more test automation, but lacked the domain expertise the testers had. They agreed on an approach where the testers would get started with the examples, and the programmers would write most of the support code to get the examples automated.

Lack of programming knowledge does not mean that you cannot get started with the approach. Besides pen and paper, Tony didn't need anything at all to get the examples down and use them as a communication device. In fact, most teams should start with such an approach, not automating the examples at all. The enhanced communication already improves the development process. You won't get the full

benefits of applying the whole approach, but the improved communication and collaboration will get you started. This comes in handy especially if you deal with a legacy code base that is not (yet) prepared to deal with automation code.

Summary

Specification workshops, wishful thinking, and collaboration add so much to your overall testing concert. First, to make sure that your team builds the right thing, you talk to your customer. By working closely together on the acceptance criteria you form a ubiquitous understanding in your team.

Starting from the business user goals, you apply wishful thinking to form the API that you wished your application had. You build your automated tests then against this API that will support all the testability functions that you will need. Your application becomes testable with automated tests by definition and at the same time you make sure that your tests don't get too coupled to the actual implementation of the user interface.

Finally, a thing we all need to remember from time to time is we are not alone in software development. That means that we may work together with others for support when our work gets tough. This especially holds true when you work on a team that is new to agile development and consists of many specialists. In order to perform on a higher level, you will need to work with your teammates to learn some of their special skills. Over time you will be able to compensate for vacation times and sick leaves if you can replace each other.

Index

- Acceptance test-driven development (ATDD)
 - collaboration in, 51
 - in friendly test automation, 162–66
 - gaps in, 149
 - successful, 183–85
 - support codes and, 49–50, 59–60, 70
 - vs.* TDD, 118
 - test automation and, 162–66
- Acceptance test suite, 111–12, 147
- Adzic, Gojko, 121, 148, 154–55, 163, 187
- Agile-friendly automation, 156, 161, 163, 177, 178, 193
- Agile Software Development – Patterns, Principles, and Practices* (Martin), 124
- Agile Testing* (Crispin and Gregory), 150
- Airport Parking Lot test cases. *See also individual headings*
 - Automating Remaining Parking Lots test case, 41–46
 - Parking Cost Calculator Workshop test case, 3–15
 - Valet Parking Automation test case, 17–39
- Alice’s Adventures in Wonderland* (Carroll), 147
- Alpha tests, 150, 151
- Appelo, Jurgen, 150
- Arrange-Act-Assert format, 136
- ArrayFixture, 135
- Automating Remaining Parking Lots test case, 41–46
 - durationMap for, 43–44
 - economy parking lot, 44–45
 - economy parking lot examples, 44–45
 - short-term parking lot, 41–44
 - short-term parking lot examples, 41–42
 - step definitions after generalizing, 42
 - summary, 46
- Automation, 161–68
 - collaboration on, 164–66
 - domain discovery, 166–67
 - domain-specific test language in, 162
 - keyword-driven, 137–39
 - summary, 167–68
 - test, developing, 170–72
 - user-friendly, 162–64
- Behavior-driven development (BDD), 131, 132–33, 187
- Beta tests, 150, 151
- Boundary conditions, 142–43
- Boundary values, 144–45
- Business-facing tests, 150, 152
- CalculateFixture, 134
- Camel-casing, 137, 139, 191, 194, 199
- Carroll, Lewis, 147
- Clean Code* (Martin), 170
- Clojure, 163
- Code. *See also individual codes*
 - library, 199
 - production, 121–22
 - refactoring, 176
 - supporting, 59–60
- Code in Light States test case, developing, 66–70
 - deciding which one to return based on previous state, 69
 - final test table for car states, 69
 - first example passes, 68
 - first flow after implementing light configurations, 70
 - first support code class, 66

- Code in Light States test case, developing
 - (*continued*)
 - FitNesse result after creating class, 67
 - FitNesse test turns from yellow to red, 67
 - folder layout in, 66
 - hard-coded return value of `nextState` method, 68
 - second test expressing state transition from red and yellow to green, 69
 - support code class with empty method bodies, 67
- Cohn, Mike, 158
- Collaboration, 50–52, 153–59
 - in ATDD, 51
 - on automation, 164–66
 - Power of Three, 153–55
 - summary, 159
 - trawl requirements, 158–59
 - workshops, 155–57
- ColumnFixture, 134
- Command/Query-Separation, 98
- Concordion Framework, 163
- Continuous integration (CI)
 - in domain discovery, 121
 - plug-ins for, 184
 - in software delivery, 51
 - test automation code and, 139
 - test failure in, 147
 - in Value Parking Automation test case, 18
- Controller, driving. *See* Driving controller in First Crossing test case
- Controller specifications in First Crossing test case, 93–94
 - happy path scenario for controller state transitions, 94
 - state transitions to consider, 94
- Copeland, Lee, 143
- Crispin, Lisa, 147, 150, 155
- Cucumber, 163, 187–89
 - feature files, 187–88
 - production code, 189
 - Ruby and, 17, 163, 187, 188
 - step definitions, 188–89
- Decision tables, 134–35, 192
 - FitNesse SLiM, example of, 192
 - function of, 58
 - query tables combined with, 136, 193
 - in refactoring, 104
 - setup table preparing three different accounts, 134
 - support code for, 194
 - in test cases, 61, 65, 94, 98, 104
 - valet parking tests expressed as, in SLiM, 58, 134
- De Florinier, Annette, 163, 187
- De Florinier, David, 163, 187
- Domain, discover, 120–21
- Domain discovery, 166–67
- Domain partitioning, 143
- Domain-specific test language, 162
- Domain testing, 143–44
- Driving controller, 94–118
 - Driving controller in First Crossing test case, 94–118
 - body for controller glue code class that switches state, 96–97
 - first empty body for controller glue code class, 95–96
 - first invalid configuration, 99
 - first LightState changed in `execute` method, 99
 - first test for, 94
 - happy path tests for controller that switches first light only, 96
 - permutated invalid configurations put into table, 102
 - second light state stored in field, 97–98
 - setting of warning configuration extracted from `if` body, 101
 - UNKNOWN light stats supported, 103
 - validation method extracted from `if` clause, 100–101
 - validation step added to controller before switching light states, 100
- Equivalence classes, 143–46
- Evans, Eric, 48, 142
- Examples, 129–52
 - cutting, 146–48
 - format of, 130–42
 - gaps, 149–51
 - refining, 142–46
 - summary, 151–52
- Exploratory tests, 136, 149, 150, 151, 152
- Extract Superclass, 176

- Feature, 17
- Feature files in Cucumber, 187-88
- First Crossing test case, 93-118
 - controller specifications, 93-94
 - driving controller, 94-118
 - refactoring, 103-18
 - summary, 118
- Fishnet scenario, 158
- FitLibrary, 134, 135, 137, 163
 - CalculateFixture in, 134
 - FitNesse and, 163
- FitNesse, 191-94
 - architecture, 192
 - in First Crossing test case, 93-118
 - FIT and, 163, 191, 192
 - introduction to, 55, 58-59
 - in Light States test case, 61-91
 - link added to front page, 63
 - shell output from starting for first time, 62
 - SLiM tables and, 58-59, 163, 191-93
 - support code, 193-94
 - welcome screen when starting for first time, 62
 - wiki structure, 58, 191-92
- Folder layout
 - in Light States test case, 66
 - in Valet Parking Automation test case, 21
- Format, 130-42
 - behavior-driven development, 131, 132-33
 - glue code and support code, 139-40
 - Internet search, basic, 132
 - keywords or data-driven tests, 131, 137-39
 - right, 140-42
 - tabulated formats, 131, 133-37
- Framework for Integrated Tests (FIT), 58, 133
 - ActionFixture in, 137
 - ColumnFixture in, 134
 - FitLibrary and, 134, 135, 137, 163
 - FitNesse and, 163, 191, 192
 - GPLv2 model and, 58, 191
 - RowFixture in, 135
- Freeman, Steve, 176
- Gaps, 149-51
- Getting Started test case
 - FitNesse, 58-59
 - refactoring of traffic lights, 56, 57
 - summary, 60
 - supporting code, 59-60
 - traffic lights, 55-56
 - unit test in Java, example of, 59
 - Valet Parking tests expressed as decision table in SLiM, 58-59
- Given-When-Then format
 - alternative to, 136
 - in BDD, 131, 132
 - in Cucumber, 163, 187
 - script tables and, 140, 193
- Glue code
 - in First Crossing test case, 95-118
 - format, 131, 139-40
 - implementing for test automation, 95-118
 - in refactoring, 107-18
 - signature of EJB2 service executor for Needle Eye pattern, 124
 - summary, 126
 - test, 122-24
 - in Valet Parking Automation test case, 17-18, 20, 21
 - value, 124-25
- GPLv2 model, 58, 191
- Green Pepper Framework, 163
- Gregory, Janet, 150, 155
- Groovy, 163
- Growing Object-oriented Software Guided by Tests* (Freeman and Pryce), 176
- Happy path scenario, 47, 93-94, 96, 103
- Headless Selenium server, 18
- Initializers in Valet Parking Automation test case, 26-31
 - duration map in, 26-27
 - extracted method for filling in parking durations, 28-30
 - final version of ParkCalcPage for initialization steps, 30-31
 - leaving date and time, 28-29
 - leaving date and time filled into form fields, 28-29
 - selecting right parking lot entry from dropdowns, 26
 - starting date and time filled into form fields, 27
- Integrated development environments (IDEs)
 - in domain discovery, 121

- Integrated development environments (IDEs) (*continued*)
 - Java, 66
 - in keyword-driven automation, 139
 - packages in, 71
 - in refactoring glue code, 109, 111
 - refactoring support and, 139, 176–78
 - setting up and configuring, 66
- Intent-revealing tests, 123, 141
- Intermodule integration tests, 150
- Java, 193
 - in Cucumber, 163
 - FitNesse in, 163
 - in Robot Framework, 195
 - unit test in, example of, 59
 - for writing support code for SLiM tables, 193–94
- JBehave Framework, 163
- JUnit, 59–60, 72–74, 79, 112
- Just-in-time requirements trawling, 159
- Jython, 195
- Kaner, Cem, 143
- Kerievsky, Joshua, 170
- Keywords
 - in automation, 137–39
 - extracting, 179–80
 - format, 131, 137–39
 - in integrated development environments, 139
 - in Robot Framework, to define test cases, 193, 196
 - scenario tables as, 137–38, 184
 - in test automation code, 138
 - Then, 19, 187
 - When, 19, 187
- Library code, 199
- Light States test case, 61–91. *See also* *individual headings*
 - code, developing, 66–70
 - FitNesse in, 61–91
 - refactoring, 70–90
 - state specifications, 61–62
 - summary, 90–91
- Light States test case, first test in, 62–66
 - expressing state transition from red to red and yellow, 65
 - FitNesse result from, 66
- FitNesse welcome screen when starting, 62
- root suite of traffic light examples, 64
- shell output from starting FitNesse, 62
- TrafficLightStates suite, contents of, 65
- Listening to test, 172–76
 - economy parking lot automated examples, 174
 - setup table preparing account hierarchies with hidden tariffs and products, 175
 - verbose example for parking lot calculator, 173–74
- Marick, Brian, 150
- Martin, Robert C.
 - Agile Software Development – Patterns, Principles, and Practices*, 124
 - Clean Code*, 170
 - Simple List Invocation Method (SLiM), 133, 191
- Meszaros, Gerard, 170
- Nonfunctional requirements, 155
- North, Dan, 132
- .NET, 163, 193
- Page objects, 184, 189
- Pairing, 25–35
 - checking results, 31–35
 - initializers, 26–31
- Pairwise testing, 145–46
- Parallel component, 70
- Parameterized tests, 59, 73–74, 112, 116
- Parking Cost Calculator Workshop test case, 3–15
 - economy parking, 6–8
 - economy parking examples, 7, 11
 - essential examples, 9–12
 - long-term parking, 6–8
 - long-term parking examples, 8, 9–10, 11
 - short-term parking, 5
 - short-term parking examples, 5, 12
 - summary, 12–15
 - valet parking, 3–4
 - valet parking examples, 4, 13
- Parking lot calculator. *See also* Parking Cost Calculator Workshop test case
 - business rules for, 3
 - extended durationMap, 38, 43
 - final version for first test, 34–35
 - final version for initialization steps, 30–31

- library for, 22
- mockup of, 25
- page, 28, 38, 151
- Robot Framework and, 196–98
- support code for, 18
- test executed by, 32
- verbose example for, 173–74
- PHP, 163, 193
- Power of Three, 153–55
 - business perspective, 153
 - introduction to, 153–55
 - mediation between perspectives, 153, 154
 - technical perspective, 153
- Practitioner's Guide to Software Test Design*, A (Copeland), 143, 152
- Production code, 121–22
 - collaboration in, 51
 - in Cucumber, 189
 - developing, 189
 - in domain discovery, 120–21
 - driving, 121–22
 - TDD used for, 71
 - value in, *vs.* glue code, 124
- Production code, drive, 121–22
- Pryce, Nat, 176
- Python, 163, 193, 195, 199
- Quadrants, testing, 150–51
- Query tables, 135–36, 193
 - checking existence of previously set up data, 135–36
 - FitNesse SLiM, example of, 193
 - script tables used to combined with decision tables, 136, 193
- Refactoring, 103–18. *See also* Refactor tests code, 176
 - in editing LightStates, 78–90
 - in LightState enum, 71–78
 - in Light States test case, 70–90
 - packages, 71
 - support code in, 70–90
 - traffic light calling for, 56, 57
- Refactoring in First Crossing test case
 - adapted validation function, 116
 - corrected validation function, 114–15
 - examples, 103
 - final unit tests for validator, 117
 - function refactoring into own class, 107
 - glue, 107–18
 - new empty validator class, 108
 - redundant information eliminated from invalid combinations, 104–5
 - retrofitted unit tests for validator class, 112–13
 - scenario table references transparent operation, 106–7
 - scenario table results in collapsible section, 105
 - scenario table to remove redundant information from invalid combinations, 103
 - second validation added after light was switched, 114
 - tests up to failing configuration point, 115
 - validation function after adding check for blinking second light, 116
 - validation function after introducing two parameters, 108
 - validator after moving validation function, 109–10
 - validator makes parameter to validation function to move method, 109
 - validator method after rename, 111
 - validator turned into field within controller class, 110–11
- Refactoring in Light States test case, 70–90
 - all light states covered, 75–78
 - changed implementation of editor class, 87
 - code for getAsText method in LightStateEditor, 86
 - code for LightState enumeration after implementing first method on editor, 85–86
 - conditional to rescue for red and yellow state, 80
 - duplication indicates the refactoring potential, 81
 - editing LightStates, 78–90
 - extracted body from LightStateEditor as valueFor on LightState class, 87
 - final example page after, 71
 - final TrafficLights implementation making use of domain concept of LightState, 90
 - first and second unit test expressed as data-driven test using JUnit Parameterized runner, 72–74
 - first code base for new traffic light enum, 72–73

- Refactoring in Light States test case
 - (*continued*)
 - first implementation for LightState
 - transition from red to red and yellow, 73
 - first implementation of LightStateEditor, 79
 - first unit test driving domain concept of LightState, 72
 - iterating over all values in LightState
 - checking for matching description, 82
 - LightStateEditor code after driving implementation for setAsText, 85
 - in LightState enum, 71-78
 - LightState takes String parameter for description in constructor, 81
 - new unit test class for valueFor method, 88-89
 - packages, 71
 - property editor in shortest form, 83
 - red state makes use of description field, 83
 - second transition captured, 75
 - second unit test using data driven format, 79-80
 - unit test for editor with defaulting behavior unit test removed, 88
 - unit test for getAsText function in LightStateEditor, 86
 - unit test for setting value to red, 79
 - unit tests for editor after finishing implementation for setAsText, 84
- Refactoring to Patterns* (Kerievsky), 170
- Refactor tests, 176-80
 - extract keyword, 179-80
 - extract variable, 178-79
- Refining examples, 142-46
 - boundary values, 144-45
 - domain testing, 143-44
 - pairwise testing, 145-46
- Regression test suite, 147, 148
- Robot Framework, 163, 195-99
 - formats, 140
 - keywords section used for parking lot calculator, 197-98
 - keywords used to define test cases, 193, 196
 - library code, 199
 - sections, 195-99
 - Selenium library in, 173
 - settings used for parking lot calculator, 196
 - test case section used for parking lot calculator, 196-97
 - test libraries in, 137
 - test using keyword template for data-driven tests, 199
 - Variables section used for parking lot calculator, 198
- RowFixture, 135
- RSpec book, The* (Chelimsky et al.), 187
- Ruby, 17, 163, 187, 188
- Rule of Three Interpretations, 155, 167. *See also* Power of Three
- Scala, 163
- Scenario tables
 - function of, 59
 - Given-When-Then-format and, 193
 - as keyword, 137-38, 184
 - in refactoring, 103-6
 - in test cases, 58-59, 103-6
 - used with script tables, 104, 193
- Script tables, 136-37
 - decision tables combined with query tables, 136, 193
 - in FIT and FitLibrary, 137
 - Given-When-Then format and, 140, 193
 - scenario tables used with, 104, 193
 - support code and, 194
 - with whole flow through system, 136
- Secret Ninja Cucumber Scrolls, The* (de Florinier, Adzic, and de Florinier), 163, 187
- Selenium, 18-22, 173-74, 197
 - headless, 18
 - library, 22, 173
- SetFixture, 135
- Setup table, 134, 175, 193
- Shell output
 - from starting FitNesse for the first time, 62
 - from Valet Parking Automation test case, 19-20, 32, 36
- Simple List Invocation Method (SLiM). *See also individual tables*
 - decision tables in, 134-35
 - FitNesse and, 58-59, 163, 191, 192
 - introduction of, 133
 - query tables in, 135-36
 - script tables in, 136-37
 - support code for tables in, 193-94

- Software specifications, 153–59
 - Power of Three, 153–55
 - summary, 159
 - trawl requirements, 158–59
 - workshops, 155–57
- Specification by Example* (Adzic), 154–55
- Specifications, 93–94
- Specification workshops, 48–49
- Stable Dependencies Principle, 124
- State pattern, 72, 119, 120
- State Pattern, 72
- State specifications in Light States test case, 61–62
 - all light states for cars, 62
 - valid light states for cars, 61
- Step definitions
 - in BDD, 133
 - Cucumber, 188–89
 - Short-Term Parking lot test case, 42
 - in Valet Parking Automation test case, 17–39
- SubsetFixture, 135
- Successful ATDD, 183–85
- Support code
 - collaboration in, 51
 - developing, 66–90
 - for FitNesse SLiM tables, 193–94
 - format, 139–40
 - in Light States test case, 66–90
 - refactoring, 70–90
 - for setting up Selenium client, 22
 - in Valet Parking Automation test case, 17–18, 20, 21–22
- SWT applications, 152, 183
- SWTBot, 152, 183
- Tab-ordering, 149
- Tabulated formats, 131, 133–37
 - decision tables, 134–35
 - query tables, 135–36
 - script tables, 136–37
 - script tables with whole flow through system, 136
- Tabulated tests in Valet Parking Automation test case, 36–38
 - examples from workshop filled into table, 37
 - first test converted to tabulated format, 36
 - ParkCalcPage class with extended durationMap for Valet Parking tests, 38
 - shell output with first test in tabulated format, 36–37
- Technical Debt, 150
- Technical tests, 150
- Test, 169–81. *See also* Test suite
 - automation, developing, 170–72
 - code (*See* Code)
 - failing, 115–17
 - gaps, 150–51
 - intent-revealing, 123, 141
 - intermodule integration, 150
 - keyword, extracting, 179–80
 - language, 124–25, 131, 162
 - libraries in Robot Framework, 137
 - listening to, 172–76
 - parameterized, 59, 73–74, 112, 116
 - quadrants, 150–52
 - refactor, 176–80
 - retrofitted, 91, 112–13, 122, 173, 183
 - smell, 135
 - summary, 180–81
 - tabulated, 36–38
 - test automation, 170–72
 - unit, 150, 172
 - variable, extracting, 178–79
- Test automation. *See also* Test automation code; Test automation tools
 - architecture, 17
 - ATDD approach to, 47–48
 - collaboration in, 47, 164–66
 - developing, 170–72
 - listening to tests and, 172–76
 - pairing on, 164
 - refactoring in, 176–78
 - in Valet Parking Automation test case, 17–39
- Test automation code. *See also* Glue code; Support code
 - flexibility of, 171
 - format, 131
 - keywords in, 138
 - support code and, 70, 139–40
 - TDD used to implement, 119, 140, 176
 - testing, 170
- Test automation tools
 - Agile-friendly, 156, 161, 163, 177, 178, 193
 - ATDD-friendly, 162–66
 - Cucumber, 187–89
 - Robot Framework, 193

- Test-driven development (TDD). *See also*
 - Test automation code
 - vs.* ATDD, 118
 - collaboration in, 51
 - in domain discovery, 120–21
 - enum class implemented using, 71–78
- Testing Computer Software* (Kaner), 143
- Test suite
 - acceptance test, 111–12, 147
 - benefits of, 122
 - cutting examples, 146–48
 - defined, 62
 - maintenance of, 136
 - regression, 147, 148
 - 36-hour, 148
- Then-keyword, 19, 187
- 36-hour test suite, 148
- Titanic Effect, 150
- Traffic Light Software System test cases. *See also individual headings*
 - First Crossing test case, 93–118
 - Getting Started test case, 55–60
 - Light States test case, 61–91
 - refactoring, 56, 57
 - supporting code, 59–60
 - traffic lights, 55–57
- Trawl requirements, 158–59
- Twist Framework, 163
- Unit test, 150, 172
 - in Java, example of, 59
 - retrofitted, 91, 112–13, 122, 173, 183
- Unknown-Unknowns Fallacy, 150
- Usability tests, 150, 151, 152
- User acceptance tests, 150, 151
- Valet Parking Automation test case, 17–39.
 - See also* Parking lot calculator
 - initializers, 26–31
 - mockup for airline parking lot calculator, 25
 - pairing for first test, 25
 - summary, 39
 - tabulated tests, 36–38
- Valet Parking Automation test case, first example, 18–25
 - duration map in, 26–27
 - empty implementation of parking cost calculation step, 25
 - empty implementations added to ParkCalcPage class, 24
 - first Valet Parking test, 18
 - folder layout, 21
 - initial ParkCalc class, 22
 - initial step definitions for, 20
 - initial wishful implementation of first keyword, 23, 24
 - shell output from first Valet Parking test, 19–20
 - shell output from first Valet Parking test with step definitions, 21
 - support code for setting up Selenium client, 22
- Valet Parking Automation test case results, checking, 31–35
 - check after factoring out two functions for individual steps, 33
 - final version of ParkCalcPage class for first test, 34–35
 - final version of steps for first test, 34
 - final version of Valet Parking steps for first test, 34
 - initial version of check, 32
 - shell output from first Valet Parking test, 32
- Variables
 - extracting, 178–79
 - for parking lot calculator in Robot Framework, 198
- Warning configuration, 101
- Weinberg, Jerry
 - Rule of Three interpretations, 167
 - Titanic Effect, 150
- When-keyword, 19, 187
- Wiki pages, 58, 66, 191–92
- Wiki server, 58, 62
- Wiki structure, 191–92
- Wiki system, 58
- Wishful thinking, 49–50
- Workshops, 155–57
 - frequency and duration, 157
 - goals of, 156–57
 - participants, 156
- XUnit Test Patterns* (Meszaros), 170