THE EXPERT'S VOICE® IN PROGRAMMING

Design Driven Testing

Test Smarter, Not Harder

Program and test from the same design

Matt Stephens and Doug Rosenberg

Apress®

Design Driven Testing

Test Smarter, Not Harder

Matt Stephens
Doug Rosenberg

Apress®

Design Driven Testing: Test Smarter, Not Harder

ISBN-13 (pbk): 978-1-4302-2943-8

ISBN-13 (electronic): 978-1-4302-2944-5

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

President and Publisher: Paul Manning
Lead Editor: Jonathan Gennick
Technical Reviewer: Jeffrey Kantor and David Putnam
Editorial Board: Clay Andres, Steve Anglin, Mark Beckner, Ewan Buckingham, Gary Cornell, Jonathan Gennick, Jonathan Hassell, Michelle Lowman, Matthew Moodie, Duncan Parkes, Jeffrey Pepper, Frank Pohlmann, Douglas Pundick, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Coordinating Editor: Anita Castro
Copy Editor: Mary Ann Fugate
Compositor: MacPS, LLC
Indexer: BIM Indexing & Proofreading Services
Artist: April Milne
Cover Designer: Anna Ishchenko

Distributed to the book trade worldwide by Springer Science+Business Media, LLC, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/info/bulksales.

Contents at a Glance

Contents

Foreword

As program manager for ESRI's ArcGIS mapping software product, Jim is responsible for a daily build of 20 million lines of code. He previously managed the transportation and logistics department for ESRI Professional Services, where he brought many multi-million–dollar software projects in on-schedule and on-budget using the design techniques described in this book.

Lots of people have many strong opinions on virtually all aspects of testing software. I've seen or heard of varied methodologies, systems, procedures, processes, patterns, hopes, prayers, and simple dumb luck that some code paths will never get executed. With all this help, we must have figured out how to ship outstanding rock-solid bug-less software, right? Yet, with each new release of the next great, most stable revision of our software products, test engineers still make that wincing face (you know the one), drawing back a little when asked, "Do you have the appropriate tests across the entire software stack?"

The wincing exists because the truthful answer to that question is almost always: "I think so, but I don't really know all the areas I should have had tests for." And the software shipped anyway, didn't it—bummer. This is job security for the technical support team, because the bugs shipped out will be coming right back to your team for the next service pack or emergency hot fix. We can do much better, and this book will show you how.

This book walks you through a proven software development process called ICONIX Process, and focuses on the creation and maintenance of both unit and acceptance tests based on and driven by the software design. This is design-driven testing (DDT). This is leveraging your design to pinpoint where critical tests need to be based on the design and object behavior. This is not test-driven design (TDD), where unit tests are written up front, before design is complete and coding starts. I don't know about you, but I think it's hard to predict the future, and even harder to get a software engineer to code something that "fits" a set of tests.

While lots of folks have opinions about testing, one thing that I think we can all agree upon is that testing is often very hard and complex. As a program manager for a large development team, I know how quickly testing can get out of hand, or just stall out on a development project. Organizations have so much variance in the investment in testing, and, unfortunately, in the return on that investment. It's possible to do way too much testing, thus wasting investment. But it's more likely that you will do too little testing (thinking you did more than enough, of course), in the wrong areas of the software, not investing enough. This can happen because you just don't know where the tests need to be to balance your investments, yielding the right testing coverage.

This book shows how to achieve this balance and optimize the return on your testing investment by designing and building a real web mapping application. Using ICONIX process and DDT makes very clear precisely what tests are needed and where they need to be. Moreover, many of these tests will be automatically generated for you by the tools used (in this case, Enterprise Architect from Sparx Systems), which, in addition to being just super cool, has huge value for your project. So, if you need to build great software using an agile process where your tests are practically generated for free, this book is for you.

Jim McKinney
ArcGIS Program Manager, Esri

About the Authors

Matt Stephens is a software consultant with financial organizations in Central London, and founder of independent book publisher Fingerpress (`www.fingerpress.co.uk`). He has written for a bunch of magazines and websites including The Register and Application Development Trends. Find him online at Software Reality (`http://articles.softwarereality.com`).

Doug Rosenberg founded ICONIX (www.iconixsw.com) in his living room in 1984 and, after several years of building CASE tools, began training companies in object-oriented analysis and design around 1990. ICONIX specializes in training for UML and SysML, and offers both on-site and open-enrollment courses. Doug developed a Unified Booch/Rumbaugh/Jacobson approach to modeling in 1993, several years before the advent of UML, and began writing books around 1995.

Design-Driven Testing is his sixth book on software engineering (and the fourth with Matt Stephens). He's also authored numerous multimedia tutorials, including **Enterprise Architect for Power Users**, and several eBooks, including **Embedded Systems Development with SysML**.

When he's not writing or teaching, he enjoys shooting panoramic, virtual reality (VR) photography, which you can see on his travel website, `VResorts.com`.

About the Technical Reviewers

Jeffrey P. Kantor is project manager of Large Synoptic Survey Telescope (LSST) Data Management. In this capacity, Mr. Kantor is responsible for implementing computing and communications systems to provide calibration, quality assessment, processing, archiving, and end user and external system access of astronomical image and engineering data produced by the LSST.

After four years in the U.S. Army as a Russian Linguist/Signals Intelligence Specialist, starting in 1980, as an entry-level programmer, he has held positions at all levels in IT organizations, in many industry segments, including defense and aerospace, semiconductor manufacturing, geophysics, software engineering consulting, home and building control, consumer durables manufacturing, retail, and e-commerce.

Mr. Kantor has created, tailored, applied, and audited software processes for a wide variety of organizations in industry, government, and academia. He has been responsible for some of these organizations achieving ISO9000 Certification and SEI CMM Level 2 assessments. Mr. Kantor has also consulted with and trained over 30 organizations in object-oriented analysis and design, Unified Modeling Language (UML), use case–driven testing, and software project management.

Mr. Kantor enjoys spending time with his family, soccer (playing, refereeing, and coaching), and mountain biking.

David Putnam has been working in software development for the last 25 years, and since the beginning of this century, he has been one of the UK's most ardent agile proponents.

From his early days producing database systems for the construction industry, he has enjoyed an eclectic career, including positions in the fitness industry and a spell as a university lecturer. During this time, he has published many articles on software development and is a familiar sight at agile software development conferences. David now works as an independent agile software development consultant and has provided invaluable advice to some of the UK's best-known corporations.

Acknowledgments

The authors would like to thank:

The mapplet team at ESRI, including Wolfgang Hall, Prakash Darbhamulla, Jim McKinney, and Witt Mathot for their work on the book's example project.

The folks at Sparx Systems, including Geoff Sparks, Ben Constable, Tom O'Reilly, Aaron Bell, Vimal Kumar, and Estelle Gleeson for the development of the ICONIX add-in and the Structured Scenario Editor.

Alanah Stephens for being the best "Alice" ever, and Michelle Stephens for her patience during yet another book project.

Jeff Kantor and Robyn Allsman from the Large Synoptic Survey Telescope project for their help with the use case thread expander specification.

Mike Farnsworth and the folks at Virginia DMV who participated in the preliminary mapplet modeling workshop. Barbara Rosi-Schwartz for her feedback on DDT, and Jerry Hamby for sharing his Flex expertise, and particularly for his help with reverse engineering MXML.

And last but certainly not least our editors at Apress, Jonathan Gennick, Anita Castro, and Mary Ann Fugate.

Prologue

Beware the agile hype

T'was brillig when the YAGNI'd code
    did build itself ten times a day.
All flimsy were the index cards,
    designs refactored clear away.

Beware the agile hype, my son
    more code that smells, more bugs to catch.
Refactoring seem'd much more fun
    until thy skills were overmatch'd.

With vorpal unit tests in hand
    against the manxome bugs he fought.
Quite dazed was he by TDD,
    some sanity was what he sought.

But, in his timebox made of wood,
    determined by some planning game,
yon tests ran green and all was good
    until the deadline came.

It's half past two, I guess we're through
    it's time to have a tasty snack.
What's that you said, some tests ran red
    all fixed, with one quick hack!

And lo the thought came with a shock,
    design comes first, not tests, O joy!
We found upon this frabjous day
    a simpl'r process to employ.

T'was brillig when the YAGNI'd code
    did build itself ten times a day.
All flimsy were the index cards,
    designs refactored clear away…

PART 1

DDT vs. TDD

"Let the developers consider a conceptual design," the King said, for about the twentieth time that day.

"No, no!" said the Queen. "Tests first—design afterwards."

"Stuff and nonsense!" said Alice loudly. "The idea of writing the tests first!"

"Hold your tongue!" said the Queen, turning purple. "How much code have you written recently, anyway?" she sneered.

"I won't," said the plucky little Alice. "Tests shouldn't drive design, design should drive testing. Tests should verify that your code works as it was designed, and that it meets the customer's requirements, too," she added, surprised by her own insight. "And when you drive your tests from a conceptual design, you can test smarter instead of harder."

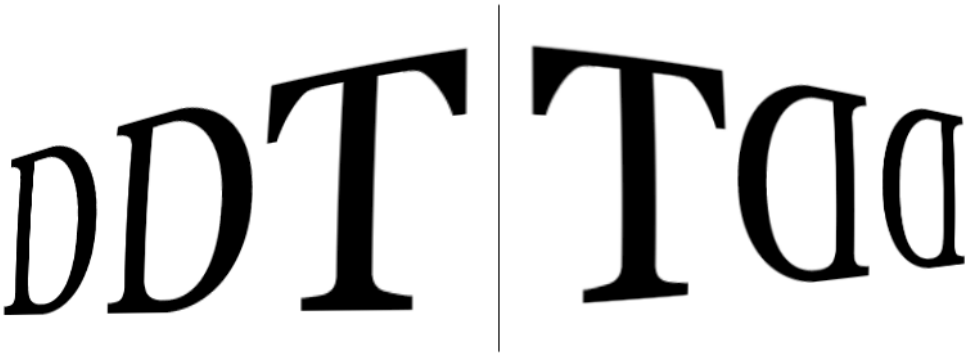
This is a book about testing; not just QA-style visual inspection, but also automated testing—driving unit tests, controller tests, and scenario tests from your design and the customer's requirements. As such, there's bound to be some crossover between the techniques described in this book and what people set out to do with test-driven development (TDD). In some ways the two processes work well together, and we hope that test-driven software developers will gain from combining the best of both worlds. That said, there are also some fundamental differences, both in the practices and the ideas underpinning both disciplines.

Chapter 1 in this book provides a high-level overview of DDT. We also briefly introduce the Mapplet project that we'll be using later in the book to illustrate how to get the best from design-driven testing.

Chapters 2 and 3 go on to compare and contrast DDT and TDD. In Chapter 2 we run through what it's like to approach a project using TDD. By the end of the chapter we hope you're convinced that there must surely be a better way. And there is! We run through the same scenario again in Chapter 3, but this time using DDT. The results are far more satisfying.

CHAPTER 1

Somebody Has It Backwards



The first time we saw a description of Test-Driven Development (TDD), our immediate thought was: "That's just backwards!" Wanting to give the process the benefit of the doubt, Matt went ahead and put TDD into practice on his own projects, attended seminars, kept up with the latest test-driven trends, and so forth. But the nagging feeling was still there that it just didn't feel right to drive the design from the tests. There was also the feeling that TDD is highly labor-intensive, luring developers into an expensive and ultimately futile chase for the holy grail of 100% test coverage. Not all code is created equal, and some code benefits more from test coverage than other code.[1] There just had to be a better way to benefit from automated testing.

Design-Driven Testing (DDT) was the result: a fusion of up-front analysis and design with an agile, test-driven mindset. In many ways it's a reversal of the thinking behind TDD, which is why we had some fun with the name. But still, somebody obviously has it backwards. We invite you to walk through the next few chapters and decide for yourself who's forwards and who's backwards.

In this chapter we provide a high-level overview of DDT, and we compare the key differences between DDT and TDD. As DDT also covers analysis and acceptance testing, sometimes we compare

[1] Think algorithms vs. boilerplate code such as property getters and setters.

DDT with practices that are typically layered on top of TDD (courtesy of XP or newer variants, such as Acceptance TDD).

Then, in Chapters 2 and 3, we walk through a "Hello world!" example, first using TDD and then using DDT.

The rest of the book puts DDT into action, with a full-on example based on a real project, a hotel finder application based in a heterogeneous environment with a Flex front-end and Java back-end, serving up maps from ArcGIS Server and querying an XML-based hotel database. It's a non-trivial project with a mixture of technologies and languages—increasingly common these days—and provides many challenges that put DDT through its paces.

Problems DDT Sets Out to Solve

In some ways DDT is an answer to the problems that become apparent with other testing methodologies, such as TDD; but it's also very much an answer to the much bigger problems that occur when

- no testing is done (or no automated tests are written)
- some testing is done (or some tests are written) but it's all a bit aimless and ad hoc
- *too much* testing is done (or too many low-leverage tests are written)

That last one might seem a bit strange—surely there can be no such thing as too much testing? But if the testing is counter-productive, or repetitive, then the extra time spent on testing could be time wasted (law of diminishing returns and all that). There are only so many ways you can press a doorbell, and how many of those will be done in real life? Is it really necessary to prove that the doorbell still works when submerged in 1000 ft. of water? The idea behind DDT is that the tests you write and perform are closely tied into the customer's requirements, so you spend time testing only what needs to be tested.

Let's look at some of the problems that you should be able to address using DDT.

Knowing When You're Done Is Hard

When writing tests, it's sometimes unclear when you're "done"… you could go on writing tests of all kinds, forever, until your codebase is 100% covered with tests. But why stop there? There are still unexplored permutations to be tested, additional tests to write… and so on, until the customer gives up waiting for something to be delivered, and pulls the plug. With DDT, your tests are driven directly from your use cases and conceptual design; it's all quite systematic, so you'll know precisely when you're done.

■ **Note** At a client Doug visited recently, the project team decided they were done with acceptance testing when their timebox indicated they had better start coding. We suspect this is fairly common.

Code coverage has become synonymous with "good work." Code metrics tools such as Clover provide reports that over-eager managers can print out, roll up, and bash developers over the head with if the developers are not writing enough unit tests. But if 100% code coverage is practically unattainable, you could be forgiven for asking: why bother at all? Why set out knowing in advance that

the goal can never be achieved? By contrast, we wanted DDT to provide a clear, achievable goal. "Completeness" in DDT isn't about blanket code coverage, it's about ensuring that key decision points in the code—logical software functions—are adequately tested.

Leaving Testing Until Later Costs More

You still see software processes that put "testing" as a self-contained phase, all the way after requirements, design, and coding phases. It's well established that leaving bug-hunting and fixing until late in a project increases the time and cost involved in eliminating those bugs. While it does make sense intuitively that you can't test something before it exists, DDT (like TDD and other agile processes) gets into the nooks and crannies of development, and provides early feedback on the state of your code and the design.

Testing Badly Designed Code Is Hard

It sounds obvious, but code that is badly designed tends to be rigid, difficult to adapt or re-use in some other context, and full of side effects. By contrast, DDT inherently promotes good design and well-written, easily testable code. This all makes it extremely difficult to write tests for. In the TDD world, the code you create will be inherently testable, because you write the tests first. But you end up with an awful lot of unit tests of questionable value, and it's tempting to skip valuable parts of the analysis and design thought process because "the code is the design." With DDT, we wanted a testing process that inherently promotes good design and well-written, easily testable code.

■ **Note** Every project that Matt has joined halfway through, without exception, has been written in such a way as to make the code difficult (or virtually impossible) to test. Coders often try adding tests to their code and quickly give up, having come to the conclusion that unit testing is too hard. It's a widespread problem, so we devote Chapter 9 to the problem of difficult-to-test code, and look at just *why* particular coding styles and design patterns make unit testing difficult.

It's Easy to Forget Customer-Level Tests

TDD is, by its nature, all about testing at the detailed design level. We hate to say it, but in its separation from Extreme Programming, TDD lost a valuable companion: acceptance tests. Books on TDD omit this vital aspect of automated testing entirely, and, instead, talk about picking a user story (aka requirement) and immediately writing a unit test for it.

DDT promotes writing both acceptance tests and unit tests, but at its core are **controller tests**, which are halfway between the two. Controller tests are "developer tests," that look like unit tests, but that operate at the conceptual design level (aka "logical software functions"). They provide a highly beneficial glue between analysis (the problem space) and design (the solution space).

Developers Become Complacent

It's not uncommon for developers to write a few tests, discover that they haven't achieved any tangible results, and go back to cranking out untested code. In our experience, the 100% code coverage "holy grail," in particular, can breed complacency among developers. If 100% is impossible or impractical, is 90% okay? How about 80%? I didn't write tests for these classes over here, but the universe didn't implode (yet)… so why bother at all? If the goal set out by your testing process is easily and obviously achievable, you should find that the developers in your team go at it with a greater sense of purpose. This brings us to the last issue that DDT tackles.

Tests Sometimes Lack Purpose

Aimless testing is sometimes worse than not testing at all, because it provides an illusion of safety. This is true of both manual testing (where a tester follows a test script, or just clicks around the UI and deems the product good to ship), and writing of automated tests (where developers write a bunch of tests in an ad hoc fashion).

Aimless unit testing is also a problem because unit tests mean more code to maintain and can make it difficult to modify existing code without breaking tests that make too many assumptions about the code's internals. Moreover, writing the tests themselves eats up valuable time.

Knowing why you're testing, and knowing why you're writing a particular unit test—being able to state succinctly what the test is proving—ensures that each test must pull its own weight. Its existence, and the time spent writing it, must be justified. The purpose of DDT tests is simple: ***to prove systematically that the design fulfills the requirements and the code matches up with the design.***

A Quick, Tools-Agnostic Overview of DDT

In this section we provide a lightning tour of the DDT process, distilled down to the cookbook steps. While DDT can be adapted to the OOAD process of your choice, it was designed originally to be used with the ICONIX Process (an agile OOAD process that uses a core subset of UML).[2] In this section, we show each step in the ICONIX Process matched by a corresponding DDT step. The idea is that DDT provides instant feedback, validating each step in the analysis/design process.

Structure of DDT

Figure 1–1 shows the four principal test artifacts: unit tests, controller tests, scenario tests, and business requirement tests. As you can see, **unit tests** are fundamentally rooted in the design/solution/implementation space. They're written and "owned" by coders. Above these, **controller tests** are sandwiched between the analysis and design spaces, and help to provide a bridge between the two. **Scenario tests** belong in the analysis space, and are manual test specs containing step-by-step instructions for the testers to follow, that expand out all sunny-day/rainy-day permutations of a use case. Once you're comfortable with the process and the organization is more

[2] We provide enough information on the ICONIX Process in this book to allow you to get ahead with DDT on your own projects; but if you also want to learn the ICONIX Process in depth, check out our companion book, *Use Case Driven Object Modeling with UML: Theory and Practice*.

amenable to the idea, we also highly recommend basing "end-to-end" integration tests on the scenario test specs. Finally, **business requirement tests** are almost always manual test specs; they facilitate the "human sanity check" before a new version of the product is signed off for release into the wild.

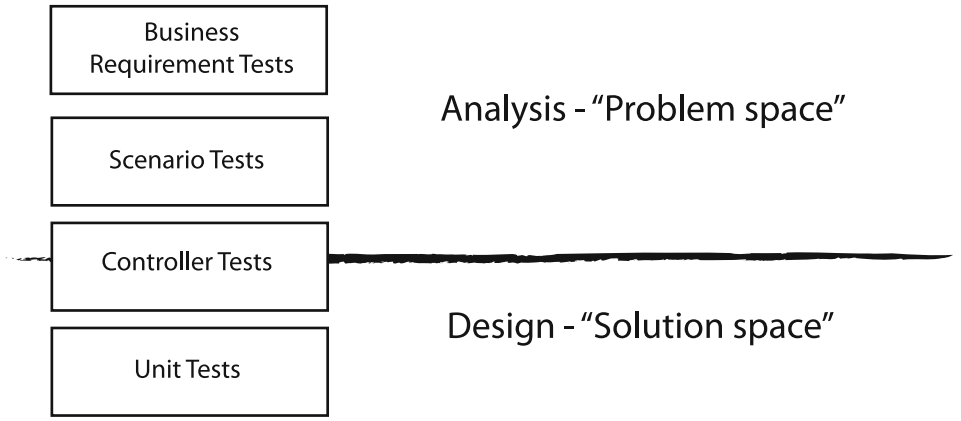


Figure 1–1. *The four principal flavors of tests in DDT*

The tests vary in granularity (that is, the amount of underlying code that each test is validating), and in the number of automated tests that are actually written. We show this in Figure 1–2.

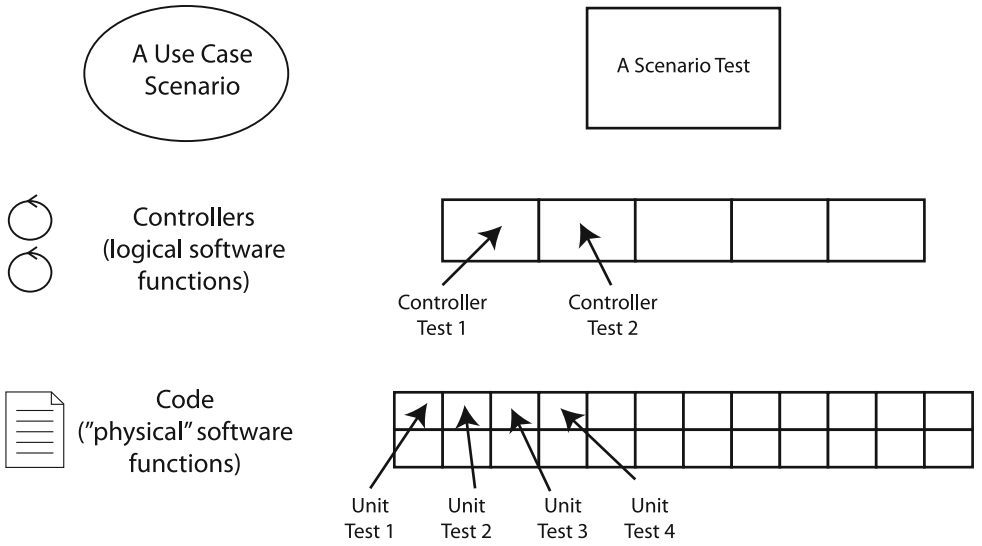


Figure 1–2. *Test granularity: the closer you get to the design/code, the more tests are written; but each test is smaller in scope.*

Figure 1–2 also shows that—if your scenario test scripts are also implemented as automated integration tests—each use case is covered by exactly one test class (it's actually one test case per use case scenario; but more about that later). Using the ICONIX Process, use cases are divided into controllers, and (as you might expect) each controller test covers exactly one controller. The controllers are then divided into actual code functions/methods, and the more important methods get one or more unit test methods.[3]

You've probably seen the traditional "V" model of software development, with analysis/design/development activities on the left, and testing activities on the right. With a little wrangling, DDT actually fits this model pretty well, as we show in Figure 1–3.

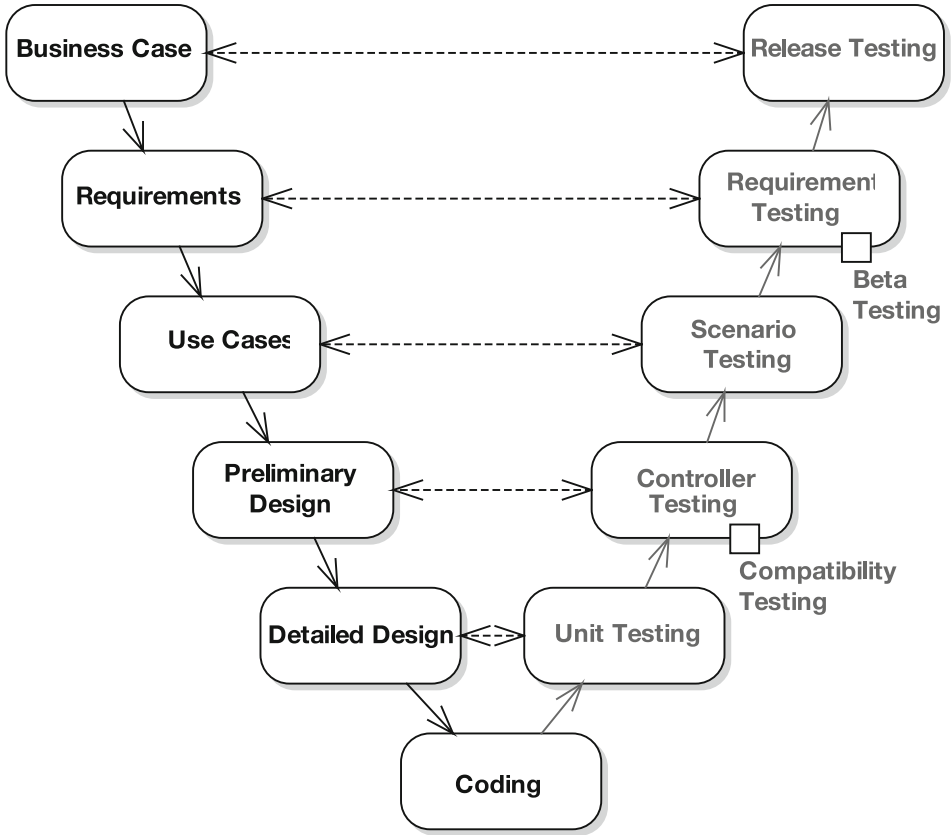


Figure 1–3. *V model of development adapted to DDT*

[3] So (you might well be wondering), how do I decide whether to cover a method with a unit test or a controller test? Simple—implement the controller tests first; if there's any "significant" (non-boilerplate) code left uncovered, consider giving it a unit test. There is more about this decision point in Chapters 5 and 6.

Each step on the left is a step in the ICONIX Process, which DDT is designed to complement; each step on the right is a part of DDT. As you create requirements, you create a requirements test spec to provide feedback—so that, at a broad level, you'll know when the project is done. The real core part of DDT (at least for programmers) centers around controller testing, which provides feedback for the conceptual design step and a systematic method of testing software behavior. The core part of DDT for analysts and QA testers centers around scenario testing.

DDT in Action

With the V diagram in Figure 1–3 as a reference point, let's walk through ICONIX/DDT one step at a time. We're deliberately not stipulating any tools here. However, we encourage you to have a peek at Chapter 3 where we show a "Hello world!" example (actually a Login use case) of DDT in action, using Enterprise Architect (EA). So, with that in mind, let's step through the process.

1. **ICONIX** ➔ Explore the business requirements (that is, functional and non-functional requirements) in detail. Talk to the relevant people—customers, analysts, end-users, and so forth. Draw UI storyboards (wireframes/mockups) and produce a set of high-level requirements or user stories.

   ➔ ***DDT:*** Create test cases from the requirements. These should be acceptance criteria that, at a broad level, you can "tick off" one by one to confirm that the project is done.

2. **ICONIX** ➔ Create a domain model, and write some use cases. Think of a use case as a step-by-step description of user/system interaction: "The user presses a button; the system does some stuff and then displays the results. The user does something else… etc." Divide each use case into its Basic Course ("sunny day scenario") and as many Alternate Courses ("rainy day scenarios") as you can think of.

   ➔ ***DDT:*** Expand the use case threads into scenario tests. These are test specs that you hand to the testing team, and (if you're heavily "into" the automated testing thing) they can also be automated, end-to-end **integration tests**.[4] Note that the use case descriptions themselves are very useful as test specs, since (written the ICONIX way) they read like step-by-step guides on using the system and exploring the various avenues and success/failure modes.

3. **ICONIX** ➔ For each use case, begin to explore the design at a preliminary level. Using the ICONIX Process, this *conceptual design* is done using **robustness analysis**. This is an effective technique that serves as a "reality check" for your use case descriptions. It also helps you to identify the behavior (verbs, actions, or "controllers") in your design. Think of a controller as a "logical software function"—it may turn into several "real" functions in the code.

[4] Integration tests have their own set of problems that, depending on factors such as the culture of your organization and the ready availability of up-to-date, isolated test database schemas, aren't always easily surmounted. There is more about implementing scenario tests as automated integration tests in Part 3.

➔ ***DDT:*** Systematically create **controller tests** from the robustness diagrams, as follows:

a. For each controller, create a test case. These test cases validate the critical behavior of your design, as driven by the controllers identified in your use cases during robustness analysis (hence the name "controller test"). Each controller test is created as a method in your chosen unit testing framework (JUnit, FlexUnit, NUnit etc). Because each test covers a "logical" software function (a group of "real" functions with one collective output), you'll end up writing fewer controller tests than you would TDD-style unit tests.

b. For each controller test case, think about the expected acceptance criteria—what constitutes a successful run of this part of the use case? (Refer back to the use case alternate courses for a handy list.)

4. **ICONIX** ➔ For each use case, drill down into the detailed design. It's time to "get real" and think in gritty detail about how this system is going to be implemented. Create a sequence diagram for each use case.

➔ ***DDT:*** Systematically create unit tests from the design.

If you're familiar with TDD, you'll notice that this process differs significantly. There are actually many ways in which the two processes are similar (both in mechanics and underlying goals). However, there are also both practical and philosophical differences. We cover the main differences in the next section.

How TDD and DDT Differ

The techniques that we describe in this book are not, for the most part, incompatible with TDD—in fact, we hope TDDers can take these principles and techniques and apply them successfully in their own projects. But there are some fundamental differences between our guiding philosophy and those of the original TDD gurus, as we show in Table 1–1. We explain our comments further in the "top 10" lists at the start of Chapters 2 and 3.

Table 1–1. *Differences Between TDD and ICONIX/DDT*

TDD	ICONIX/DDT
Tests are used to drive the design of the application.	With DDT it's the other way around: the tests are driven from the design, and, therefore, the tests are there primarily to validate the design. That said, there's more common ground between the two processes than you might think. A lot of the "design churn" (aka refactoring) to be found in TDD projects can be calmed down and given some stability by first applying the up-front design and testing techniques described in this book.
The code is the design and the tests are the documentation.	**The design is the design, the code is the code, and the tests are the tests.** With DDT, you'll use modern development tools to keep the documented design model in sync with the code.

TDD	ICONIX/DDT
Following TDD, you may end up with a lot of tests (and we mean a *lot* of tests).	DDT takes a "test smarter, not harder" approach, meaning tests are more focused on code "hot spots."
TDD tests have their own purpose; therefore, on a true test-first project the tests will look subtly different from a "classical" fine-grained unit test. A TDD unit test might test more than a single method at a time.	In DDT, a unit test is usually there to validate a single method. DDT unit tests are closer to "real" unit tests. As you write each test, you'll look at the detailed design, pick the next message being passed between objects, and write a test case for it. DDT also has controller tests, which are broader in scope.[5] So TDD tests are somewhere between unit tests and controller tests in terms of scope.
TDD doesn't have acceptance tests unless you mix in part of another process. The emphasis (e.g., with XP) tends to be on automated acceptance tests: if your "executable specification" (aka acceptance tests) can't be automated, then the process falls down. As we explore in Part 3, writing and maintaining automated acceptance tests can be very difficult.	DDT "acceptance tests" (which encompass both scenario tests and business requirement tests) are "manual" test specs for consumption by a human. Scenario tests can be automated (and we recommend doing this if at all possible), but the process doesn't depend on it.
TDD is much finer-grained when it comes to design.[6] With the test-first approach, you pick a story card from the wall, discuss the success criteria on the back of the card with the tester and/or customer representative, write the first (failing) test, write some code to make the test pass, write the next test, and so on, until the story card is implemented. You then review the design and refactor if needed, i.e., "after-the-event" design.	We actually view DDT as pretty fine-grained: you might base your initial design effort on, say, a package of use cases. From the resulting design model you identify your tests and classes, and go ahead and code them up. Run the tests as you write the code.
A green bar in TDD means "all the tests I've written so far are not failing."	A green bar in DDT means "all the critical design junctures, logical software functions, and user/system interactions I've implemented so far are passing as designed." We know which result gives us more confidence in our code…

[5] If some code is already covered by a controller test, you don't need to write duplicate unit tests to cover the same code, unless the code is algorithmic or mission-critical—in which case, it's an area of code that will benefit from additional tests. We cover design-driven algorithm testing in Chapter 12.

[6] Kent Beck's description of this was "a waterfall run through a blender."

TDD	ICONIX/DDT
After making a test pass, review the design and refactor the code if you think it's needed.	With DDT, "design churn" is minimized because the design is thought through with a broader set of functionality in mind. We don't pretend that there'll be no design changes when you start coding, or that the requirements will never change, but the process helps keep these changes to a minimum. The process also allows for changes—see Chapter 4.
TDD: An essential part of the process is to first write the test and then write the code.	With DDT we don't stipulate: if you feel more comfortable writing the test before the accompanying code, absolutely go ahead. You can be doing this and still be "true" to DDT.
With TDD, if you feel more comfortable doing more up-front design than your peers consider to be cool… go ahead. You can be doing this and still be "true" to TDD. (That said, doing a lot of up-front design and writing all those billions of tests would represent a lot of duplicated effort).	With DDT, an essential part of the process is to first create a design, and then write the tests and the code. However, you'll end up writing fewer tests than in TDD, because the tests you'll most benefit from writing are pinpointed during the analysis and design process.

So, with DDT you don't drive the design from the unit tests. This is not to say that the design in a DDT project isn't affected by the tests. Inevitably you'll end up basing the design around testability. As we'll explore in Chapter 3, code that hasn't been written with testability in mind is an absolute pig to test. Therefore, it's important to build testability into your designs from as early a stage as possible. It's no coincidence that code that is easily tested also generally happens to be well-designed code. To put it another way, the qualities of a code design that result in it being easy to unit-test are also the same qualities that make the code clear, maintainable, flexible, malleable, well factored, and highly modular. It's also no coincidence that the ICONIX Process and DDT place their primary emphasis on creating designs that have these exact qualities.

Example Project: Introducing the Mapplet 2.0

The main example that we'll use throughout this book is known as Mapplet 2.0, a real-world hotel-finder street map that is hosted on `vresorts.com` (a travel web site owned by one of the co-authors). We invite you to compare the use cases in this book with the finished product on the web site.

Mapplet 2.0 is a next-generation version of the example generated for our earlier book, *Agile Development with ICONIX Process*, which has been used successfully as a teaching example in open enrollment training classes by ICONIX over the past few years. The original "Mapplet 1.0" was a thin-client application written using HTML and JavaScript with server-side C#, whereas the groovy new version has a Flex rich-client front-end with server-side components written in Java. In today's world of heterogeneous enterprise technologies, a mix of languages and technologies is a common situation, so we won't shy away from the prospect of testing an application that is collectively written in more than one language. As a result, you'll see some tests written in Flex and some in Java. We'll