CHAPTER 9

Developing

Imagine you've given up the world of software to become a master chef. After years of study and practice, you've reached the top of your profession. One day you receive the ultimate invitation: to cook for 500 attendees at a $10,000-a-plate fundraiser.

A limo takes you from your chartered plane to the venue, and you and your cooks walk confidently into a kitchen... only to stop in shock. The kitchen is a mess: rotting food, unwashed cooking implements, standing water in the sinks. It's the morning of the event. You have 12 hours.

What do you do? You roll up your sleeves and start cleaning. As soon as the first space is clear, a few cooks begin the longest and most complicated food preparation. Others head to the market to get fresh ingredients. The rest keep cleaning. Working around the mess will take too long.

It's a lesson you've learned from software over and over again.

Software development requires the cooperation of everyone on the team. Programmers are often called "developers," but in reality everyone on the team is part of the development effort. When you share the work, customers identify the next requirements while programmers work on the current ones. Testers help the team figure out how to stop introducing bugs. Programmers spread the cost of technical infrastructure over the entire life of the project. Above all, everyone helps keep everything clean.

The best way I know to reduce the cost of writing software is to improve the internal quality of its code and design. I've never seen high quality on a well-managed project fail to repay its investment. It always reduces the cost of development in the short term as well as in the long term. On a successful XP project, there's an amazing feeling—the feeling of being absolutely safe to change absolutely anything without worry.

Here are nine practices that keep the code clean and allow the entire team to contribute to development:

- *Incremental Requirements* allows the team to get started while customers work out requirements details.
- *Customer Tests* help communicate tricky domain rules.
- *Test-Driven Development* allows programmers to be confident that their code does what they think it should.
- *Refactoring* enables programmers to improve code quality without changing its behavior.
- *Simple Design* allows the design to change to support any feature request, no matter how surprising.

- *Incremental Design and Architecture* allows programmers to work on features in parallel with technical infrastructure.
- *Spike Solutions* use controlled experiments to provide information.
- *Performance Optimization* uses hard data to drive optimization efforts.
- *Exploratory Testing* enables testers to identify gaps in the team's thought processes.

"DEVELOPING" MINI-ÉTUDE

The purpose of this étude is to practice reflective design. If you're new to agile development, you may use it to understand your codebase and identify design improvements, even if you're not currently using XP. If you're an experienced agile practitioner, review Chapter 15 and use this étude to discover process changes that will help your team improve its approach to design.

Conduct this étude for a timeboxed half-hour every day for as long as it is useful. Expect to feel rushed by the deadline at first. If the étude becomes stale, discuss how you can change it to make it interesting again.

This étude is for programmers only. You will need pairing workstations, paper, and writing implements.

Step 1. Form pairs. Try to pair with someone you haven't paired with recently.

Step 2. *(Timebox this step to 15 minutes.)* Look through your code and choose a discrete unit to analyze. Pick something that you have not previously discussed in the larger group. You may choose a method, a class, or an entire subsystem. Don't spend too long picking something; if you have trouble deciding, pick at random.

Reverse-engineer the design of the code by reading it. Model the design with a flow diagram, UML, CRC cards, or whatever technique you prefer. Identify any flaws in the code and its design, and discuss how to fix them. Create a new model that shows what the design will look like after it has been fixed.

If you have time, discuss specific refactoring steps that will allow you to migrate from the current design to your improved design in small, controlled steps.

Step 3. *(Timebox this step to 15 minutes.)* Within the entire group of programmers, choose three pairs to lead a discussion based on their findings. Each pair has five minutes.

Consider these discussion questions:

- Which sections of the code did you choose?
- What was your initial reaction to the code?
- What are the benefits and drawbacks of the proposals?
- Which specific refactorings can you perform based on your findings?

Incremental Requirements

Audience
Customers

We define requirements in parallel with other work.

A team using an up-front requirements phase keeps their requirements in a requirements document. An XP team doesn't have a requirements phase and story cards aren't miniature requirements documents, so where do requirements come from?

The Living Requirements Document

In XP, the on-site customers sit with the team. They're expected to have all the information about requirements at their fingertips. When somebody needs to know something about the requirements for the project, she asks one of the on-site customers rather than looking in a document.

Ally
Sit Together (p. 113)

Face-to-face communication is much more effective than written communication, as [Cockburn] discusses, and it allows XP to save time by eliminating a long requirements analysis phase. However, requirements work is still necessary. The on-site customers need to understand the requirements for the software before they can explain it.

The key to successful requirements analysis in XP is *expert customers*. Involve real customers, an experienced product manager, and experts in your problem domain (see "The XP Team"" in Chapter 3 and "Real Customer Involvement"" in Chapter 6). Many of the requirements for your software will be intuitively obvious to the right customers.

Ally
Real Customer Involvement (p. 121)

> NOTE
> **If you have trouble with requirements, your team may not include the right customers.**

Some requirements will necessitate even expert customers to consider a variety of options or do some research before making a decision. Customers, you can and should include other team members in your discussions if it helps clarify your options. For example, you may wish to include a programmer in your discussion of user interface options so you can strike a balance between an impressive UI and low implementation cost.

Write down any requirements you might forget. These notes are primarily for your use as customers so you can easily answer questions in the future and to remind you of the decisions you made. They don't need to be detailed or formal requirements documents; keep them simple and short. When creating screen mock-ups, for example, I often prefer to create a sketch on a whiteboard and take a digital photo. I can create and photograph a whiteboard sketch in a fraction of the time it takes me to make a mock-up using an electronic tool.

Ally
Version Control (p. 169)

NOTE

Some teams store their requirements notes in a Wiki or database, but I prefer to use normal office tools and check the files into version control. Doing this allows you to use all of the tools at your disposal, such as word processors, spreadsheets, and presentation software. It also keeps requirements documents synchronized with the rest of the project and allows you to travel back in time to previous versions.

Work Incrementally

Work on requirements *incrementally*, in parallel with the rest of the team's work. This makes your work easier and ensures that the rest of the team won't have to wait to get started. Your work will typically parallel your release-planning horizons, discussed in "Release Planning"" in Chapter 8).

Vision, features, and stories

Start by clarifying your project vision, then identify features and stories as described in "Release Planning"" in Chapter 6. These initial ideas will guide the rest of your requirements work.

Allies

Vision (p. 202)

Release Planning (p. 207)

Rough expectations

Figure out what a story means to you and how you'll know it's finished slightly before you ask programmers to estimate it. As they estimate, programmers will ask questions about your expectations; try to anticipate those questions and have answers ready. (Over time, you'll learn what sorts of questions your programmers will ask.) A rough sketch of the visible aspects of the story might help.

NOTE

Sometimes the best way to create a UI mock-up is to work in collaboration with the programmers. The iteration-planning meeting might be the best time for this work.

Mock-ups, customer tests, and completion criteria

Figure out the details for each story just before programmers start implementing it. Create rough mock-ups that show what you expect the work to look like when it's done. Prepare customer tests that provide examples of tricky domain concepts, and describe what "done done" means for each story. You'll typically wait for the corresponding iteration to begin before doing most of this work.

Allies

Customer Tests (p. 280)

"Done Done" (p. 155)

Customer review

While stories are under development, before they're "done done," review each story to make sure it works as you expected. You don't need to exhaustively test the application—you can rely on the programmers to test their work—but you should check those areas in which programmers might think differently than you do. These areas include terminology, screen layout, and interactions between screen elements.

> Only working applications show customers what they're really going to get.

> NOTE
> **Prior to seeing the application in action, every conversation is theoretical. You can discuss options and costs, but until you have an implementation, everyone can only imagine how the choices will feel. Only working applications show you what you're really going to get.**

Some of your findings will reveal errors due to miscommunication or misunderstanding. Others, while meeting your requirements, won't work as well in practice as you had hoped. In either case, the solution is the same: talk with the programmers about making changes. You can even pair with programmers as they work on the fixes.

Many changes will be minor, and the programmers will usually be able to fix them as part of their iteration slack. If there are major changes, however, the programmers may not have time to fix them in the current iteration. (This can happen even when the change seems minor from the customer's perspective.) Create story cards for these changes. Before scheduling such a story into your release plan, consider whether the value of the change is worth its cost.

Ally

Slack (p. 247)

Over time, programmers will learn about your expectations for the application. Expect the number of issues you discover to decline each iteration.

A CUSTOMER REVIEW SESSION

By Elisabeth Hendrickson

"Jill, do you have a moment?" Mary asked, walking over to Jill's workstation. "I need to review a story and I was hoping you could show me some of your exploratory testing techniques."

Ally

Exploratory Testing (p. 342)

"Sure, I'd be happy to," Jill said. "What's the charter for our session?"

"I want to explore the Defaults story that Pradeep and Jan are working on," Mary said. "They're not done yet, but they said that the UI is mostly filled in. I wanted to make sure they're going in the right direction."

"OK," Jill said as she wrote "Explore Defaults UI" on a card. "Let's start by setting some explicit defaults in the Preferences page." She navigated to that area of the application, and Mary picked up a notepad, ready to take notes.

Jill reached the Preferences page. "Here are the defaults users can set explicitly," she said, pointing at the screen. "Is this what you were expecting?"

Mary nodded. "We sketched out a mock-up of the screen with the programmers earlier in the iteration. Now that I look at it, though, it seems crowded. I think an icon would be better than those repeated *Set Default* links." She made a note to bring up the issue with the programmers.

"Let's look at the default credit cards next," Jill said. "It's newest."

Mary took the mouse and clicked over to the credit cards screen. "This user has several credit cards, and one has already been marked as the default. We should be able to change it by clicking this radio button," she said, and did so. "Nice! I like how they enabled the *Undo* button when I clicked. But does the save button have to be labeled *Submit*?" She laughed as she made a few notes. "I feel like I'm on the set of a B movie whenever I see that. *Submit to your computer master, puny mortal!*"

Jill grinned. "I never thought of it that way." She paused to consider. "Let's use the zero/one/many heuristic* to see what happens when we create a new user with no credit cards." She created a new user, logged in, then navigated back to the Preferences page. The default credit card area on the page showed the heading and no credit cards.

"Hmmm," Mary said, then pointed at the screen. "I hadn't thought about this before, but I'd like to have an *Add New Card* button here. Users will be able to add credit cards both here and on the Payment Method page when they go to check out. I wonder if Pradeep and Jan have time to get that in this iteration?" She made a note to talk to the programmers about the new button.

"In the meantime, we can add a new credit card from the payment screen," Jill said. "I wonder what happens when you add the first card..."

Questions

Our customers don't know what the team should build. What should we do?

Do you have a clear, compelling vision? If so, your customers should know where to start. If you don't, you may not have the right customers on your team. In this case, you can use traditional requirements gathering techniques (see "Further Reading"" at the end of this section) to determine the software's requirements, but you're better off involving real experts (see "The XP Team"" in Chapter 3 and "Real Customer Involvement"" in Chapter 6).

Ally

Vision (p. 202)

What if the customer review finds too many problems for us to deal with?

This is most likely to happen at the beginning of the project, before programmers have learned what the customers like. If this happens to you, spend more time with programmers so that your perspective is captured sooner and more accurately. In some cases, customers should pair with programmers as they work on error-prone areas.

As a programmer, I'm offended by some of the things customers find in their reviews. They're too nitpicky.

* See "Exploratory Testing"" later in this chapter.

Things that can seem nitpicky to programmers—such as the color of the screen background, or a few pixels of alignment in the UI—represent polish and professionalism to customers. This goes both ways: some things that seem important to programmers, such as quality code and refactoring, often seem like unnecessary perfectionism to customers.

Rather than getting upset about these differences of perspective, try to learn what your customers care about and why. As you learn, you will anticipate your customers' needs better, which will reduce the need to make changes.

Results

When customers work out requirements incrementally, programmers are able to work on established stories while customers figure out the details for future stories. Customers have ready answers to requirements questions, which allows estimation and iteration planning to proceed quickly and smoothly. By the time a story is "done done," it reflects the customers' expectations, and customers experience no unpleasant surprises.

Contraindications

In order to incrementally define requirements, the team must include on-site customers who are dedicated to working out requirements details throughout the entire project. Without this dedicated resource, your team will struggle with insufficient and unclear requirements.

Ally

Sit Together (p. 113)

When performing customer reviews, think of them as tools for conveying the customers' perspective rather than as bug-hunting sessions. The programmers should be able to produce code that's nearly bug-free (see "No Bugs"" in Chapter 7); the purpose of the review is to bring customers' expectations and programmers' work into alignment.

Ally

No Bugs (p. 159)

Alternatives

The traditional approach to requirements definition is to perform requirements analysis sessions and document the results. This requires an up-front requirements gathering phase, which takes extra time and introduces communication errors.

You can use an up-front analysis phase with XP, but good on-site customers should make that unecessary.

Further Reading

Software Requirements [Wiegers 1999] is a good resource for classic approaches to requirements gathering.

Customer Tests

Audience
Whole Team

We implement tricky domain concepts correctly.

Customers have specialized expertise, or *domain knowledge*, that programmers don't have. Some areas of the application—what programmers call *domain rules*—require this expertise. You need to make sure that the programmers understand the domain rules well enough to code them properly in the application. *Customer tests* help customers communicate their expertise.

Ally

Ubiquitous Language (p. 125)

Don't worry; this isn't as complicated as it sounds. Customer tests are really just examples. Your programmers turn them into automated tests, which they then use to check that they've implemented the domain rules correctly. Once the tests are passing, the programmers will include them in their 10-minute build, which will inform the programmers if they ever do anything to break the tests.

Ally

Ten-Minute Build (p. 177)

To create customer tests, follow the *Describe, Demonstrate, Develop* processes outlined in the next section. Use this process during the iteration in which you develop the corresponding stories.

Describe

At the beginning of the iteration, look at your stories and decide whether there are any aspects that programmers might misunderstand. You don't need to provide examples for everything. Customer tests are for *communication*, not for proving that the software works. (See "No Bugs"" in Chapter 7.)

> Customer tests are for communication.

For example, if one of your stories is "Allow invoice deleting," you don't need to explain how invoices are deleted. Programmers understand what it means to delete something. However, you might need examples that show *when* it's OK to delete an invoice, especially if there are complicated rules to ensure that invoices aren't deleted inappropriately.

> NOTE
> **If you're not sure what the programmers might misunderstand, ask. Be careful, though; when business experts and programmers first sit down to create customer tests, both groups are often surprised by the extent of existing misunderstandings.**

Once you've identified potential misunderstandings, gather the team at a whiteboard and summarize the story in question. Briefly describe how the story should work and the rules you're going to provide examples for. It's OK to take questions, but don't get stuck on this step.

For example, a discussion of invoice deletion might go like this:

> Customer: One of the stories in this iteration is to add support for deleting invoices. In addition to the screen mock-ups we've given you, we thought some customer tests

would be appropriate. Deleting invoices isn't as simple as it appears because we have to maintain an audit trail.

There are a bunch of rules around this issue. I'll get into the details in a moment, but the basic rule is that it's OK to delete invoices that haven't been sent to customers—because presumably that kind of invoice was a mistake. Once an invoice *has* been sent to a customer, it can only be deleted by a manager. Even then, we have to save a copy for auditing purposes.

Programmer: When an invoice *hasn't* been sent and gets deleted, is it audited?

Customer: No—in that case, it's just deleted. I'll provide some examples in a moment.

Demonstrate

After a brief discussion of the rules, provide concrete examples that illustrate the scenario. Tables are often the most natural way to describe this information, but you don't need to worry about formatting. Just get the examples on the whiteboard. The scenario might continue like this:

Customer (continued): As an example, this invoice hasn't been sent to customers, so an Account Rep can delete it.

Sent	User	OK to delete
N	Account Rep	Y

In fact, anybody can delete it—CSRs, managers, and admins.

Sent	User	OK to delete
N	CSR	Y
N	Manager	Y
N	Admin	Y

But once it's sent, only managers and admins can delete it, and even then it's audited.

Sent	User	OK to delete
Y	Account Rep	N
Y	CSR	N
Y	Manager	Audited
Y	Admin	Audited

Also, it's not a simple case of whether something has been sent or not. "Sent" actually means one of several conditions. If you've done anything that *could* have resulted in a customer seeing the invoice, we consider it "Sent." Now only a manager or admin can delete it.

Sent	User	OK to delete
Printed	Account Rep	N
Exported	Account Rep	N
Posted to Web	Account Rep	N
Emailed	Account Rep	N

> NOTE
> **As you provide examples, be completely specific. It's tempting to create generic examples, such as "This invoice hasn't been sent to customers, so anybody can delete it," but those get confusing quickly and programmers can't automate them. Provide specifics. "This invoice hasn't been sent to customers, so *an account rep* can delete it." This will require you to create more examples—that's a good thing.**

Your discussion probably won't be as smooth and clean as in this example. As you discuss business rules, you'll jump back and forth between describing the rules and demonstrating them with examples. You'll probably discover special cases you hadn't considered. In some cases, you might even discover whole new categories of rules you need customer tests for.

One particularly effective way to work is to *elaborate on a theme*. Start by discussing the most basic case and providing a few examples. Next, describe a special case or additional detail and provide a few more examples. Continue in this way, working from simplest to most complicated, until you have described all aspects of the rule.

You don't need to show all possible examples. Remember, the purpose here is to communicate, not to exhaustively test the application. You only need enough examples to show the differences in the rules. A handful of examples per case is usually enough, and sometimes just one or two is sufficient.

Develop

When you've covered enough ground, document your discussion so the programmers can start working on implementing your rules. This is also a good time to evaluate whether the examples are in a format that works well for automated testing. If not, discuss alternatives with the programmers. The conversation might go like this:

> *Programmer:* OK, I think we understand what's going on here. We'd like to change your third set of examples, though—the ones where you say "Y" for "Sent." Our invoices don't have a "Sent" property. We'll calculate that from the other properties you mentioned. Is it OK if we use "Emailed" instead?
>
> *Customer:* Yeah, that's fine. Anything that sends it works for that example.

Don't formalize your examples too soon. While you're brainstorming, it's often easiest to work on the whiteboard. Wait until you've worked out all the examples around a particular business rule (or part of a business rule) before formalizing it. This will help you focus on the business rule rather than formatting details.

In some cases, you may discover that you have more examples and rules to discuss than you realized. The act of creating specific examples often reveals scenarios you hadn't considered. Testers are particularly good at finding these. If you have a lot of issues to discuss, consider letting some or all of the programmers get started on the examples you have while you figure out the rest of the details.

Programmers, once you have some examples, you can start implementing the code using normal test-driven development. Don't use the customer tests as a substitute for writing your own tests. Although it's possible to drive your development with customer tests—in fact, this can feel quite natural and productive—the tests don't provide the fine-grained support that TDD does. Over time, you'll discover holes in your implementation and regression suite. Instead, pick a business rule, implement it with TDD, then confirm that the associated customer tests pass.

> Don't use customer tests as a substitute for test-driven development.

Ally

Test-Driven Development (p. 287)

Focus on Business Rules

One of the most common mistakes in creating customer tests is describing what happens in the user interface rather than providing examples of business rules. For example, to show that an account rep must not delete a mailed invoice, you might make the mistake of writing this:

1. Log in as an account rep
2. Create new invoice
3. Enter data
4. Save invoice
5. Email invoice to customer
6. Check if invoice can be deleted (should be "no")

What happened to the core idea? It's too hard to see. Compare that to the previous approach:

Sent	User	OK to delete
Emailed	Account Rep	N

Good examples focus on the *essence* of your rules. Rather than imagining how those rules might work in the application, just think about what the rules are. If you weren't creating an application at all, how would you describe those rules to a colleague? Talk about *things* rather than *actions*. Sometimes it helps to think in terms of a template: "When (*scenario X*), then (*scenario Y*)."

It takes a bit of practice to think this way, but the results are worth it. The tests become more compact, easier to maintain, and (when implemented correctly) faster to run.

Ask Customers to Lead

> Remember the "customer" in "customer tests."

Team members, watch out for a common pitfall in customer testing: no customers! Some teams have programmers and testers do all the work of customer testing, and some teams don't involve their customers at all. In others, a customer is present only as a mute observer. Don't forget the "customer" in "customer tests." The purpose of these activities to bring the customer's knowledge and perspective to the team's work. If programmers or testers take the reins, you've lost that benefit and missed the point.

In some cases, customers may not be willing to take the lead. Programmers and testers may be able to solve this problem by asking the customers for their help. When programmers need domain expertise, they can ask customers to join the team as they discuss examples. One particularly effective technique is to ask for an explanation of a business rule, pretend to be confused, then hand a customer the whiteboard marker and ask him to draw an example on the board.

NOTE

If customers won't participate in customer testing at all, this may indicate a problem with your relationship with the customers. Ask your mentor (see "Find a Mentor"" in Chapter 2) to help you troubleshoot the situation.

TESTERS' ROLE

Testers play an important support role in customer testing. Although the customers should lead the effort, they benefit from testers' technical expertise and ability to imagine diverse scenarios. While customers should generate the initial examples, testers should suggest scenarios that customers don't think of.

On the other hand, testers should be careful not to try to cover every possible scenario. The goal of the exercise is to help the team understand the customers' perspective, not to exhaustively test the application.

Automating the Examples

Programmers may use any tool they like to turn the customers' examples into automated tests. Ward Cunningham's *Fit* (*Framework for Integrated Test*),* is specifically designed for this purpose. It allows you to use HTML to mix descriptions and tables, just as in my invoice auditing example, then runs the tables through programmer-created *fixtures* to execute the tests.

* Available free at *http://fit.c2.com/*.

NOTE

See *http://fit.c2.com/* or [Mugridge & Cunningham] for details about using Fit. It's available in several languages, including Java, .NET, Python, Perl, and C++.

You may be interested in *FitNesse* at *http://fitnesse.org/*, a variant of Fit. FitNesse is a complete IDE for Fit that uses a Wiki for editing and running tests. (Fit is a command-line tool and works with anything that produces tables in HTML.)

Fit is a great tool for customer tests because it allows customers to review, run, and even expand on their own tests. Although programmers have to write the fixtures, customers can easily add to or modify existing tables to check an idea. Testers can also modify the tests as an aid to exploratory testing. Because the tests are written in HTML, they can use any HTML editor to modify the tests, including Microsoft Word.

Ally

Exploratory Testing (p. 342)

Programmers, don't make Fit too complicated. It's a deceptively simple tool. Your fixtures should work like unit tests, focusing on just a few domain objects. For example, the invoice auditing example would use a custom `ColumnFixture`. Each column in the table corresponds to a variable or method in the fixture. The code is almost trivial (see Example 9-1).

Example 9-1. Example fixture (C#)

```
public class InvoiceAuditingFixture : ColumnFixture
{
    public InvoiceStatus Sent;
    public UserRole User;

    public Permission OkayToDelete() {
        InvoiceAuditer auditer = new InvoiceAuditer(User, InvoiceStatus)
        return auditer.DeletePermission;
    }
}
```

Using Fit in this way requires a ubiquitous language and good design. A dedicated domain layer with *Whole Value* objects* works best. Without it, you may have to write end-to-end tests, with all the challenges that entails. If you have trouble using Fit, talk to your mentor about whether your design needs work.

Ally

Ubiquitous Language (p. 125)

NOTE

I often see programmers try to make a complete library of generic fixtures so that no one need write another fixture. That misses the point of Fit, which is to segregate customer-written examples from programmer implementation. If you make generic fixtures, the implementation details will have to go into the tests, which will make them too complicated and obscure the underlying examples.

* See *Domain-Driven Design* [Evans] for a discussion of domain layers, and see *http://c2.com/ppr/checks.html#1* [Cunningham] for information about *Whole Value*.

Questions

Most tests can be expressed with a simple `ColumnFixture` or `RowFixture`.

When do programmers run the customer tests?

Once the tests are passing, make them a standard part of your 10-minute build. Like programmers' tests, you should fix them immediately if they ever break.

Ally
Ten-Minute Build (p. 177)

Should we expand the customer tests when we think of a new scenario?

Absolutely! Often, the tests will continue to pass. That's good news; leave the new scenario in place to act as documentation for future readers. If the new test doesn't pass, talk with the programmers about whether they can fix it with iteration slack or whether you need a new story.

Ally
Slack (p. 247)

What about acceptance testing (also called functional testing)?

Automated acceptance tests tend to be brittle and slow. I've replaced acceptance tests with customer reviews (see "Customer review"" later in this chapter) and a variety of other techniques (see "A Little Lie"" in Chapter 3).

Ally
No Bugs (p. 159)

Results

When you use customer tests well, you reduce the number of mistakes in your domain logic. You discuss domain rules in concrete, unambiguous terms and often discover special cases you hadn't considered. The examples influence the design of the code and help promote a ubiquitous language. When written well, the customer tests run quickly and require no more maintenance than unit tests do.

Ally
Ubiquitous Language (p. 125)

Contraindications

Don't use customer tests as a substitute for test-driven development. Customer tests are a tool to help communicate challenging business rules, not a comprehensive automated testing tool. In particular, Fit doesn't work well as a test scripting tool—it doesn't have variables, loops, or subroutines. (Some people have attempted to add these things to Fit, but it's not pretty.) Real programming tools, such as xUnit or Watir, are better for test scripting.

Ally
Test-Driven Development (p. 287)

In addition, customer tests require domain experts. The real value of the process is the conversation that explores and exposes the customers' business requirements and domain knowledge. If your customers are unavailable, those conversations won't happen.

Allies
The Whole Team (p. 28)
Sit Together (p. 113)

Finally, because Fit tests are written in HTML, Fit carries more of a maintenance burden than xUnit frameworks do. Automated refactorings won't extend to your Fit tests. To keep your maintenance costs down, avoid creating customer tests for every business rule. Focus on the tests that will help improve programmer understanding, and avoid further maintenance costs by refactoring your customer tests regularly. Similar stories will have similar tests: consolidate your tests whenever you have the opportunity.

Alternatives

Some teams have testers, not customers, write customer tests. Although this introduces another barrier between the customers' knowledge and the programmers' code, I have seen it succeed. It may be your best choice when customers aren't readily available.

Customer tests don't have to use Fit or FitNesse. Theoretically, you can write them in any testing tool, including xUnit, although I haven't seen anybody do this.

Further Reading

Fit for Developing Software [Mugridge & Cunningham] is the definitive reference for Fit.

"Agile Requirements" [Shore 2005a], online at *http://www.jamesshore.com/Blog/Agile-Requirements.html*, is a series of essays about agile requirements, customer testing, and Fit.

Test-Driven Development

Audience
Programmers

We produce well-designed, well-tested, and well-factored code in small, verifiable steps.

"What programming languages really need is a 'DWIM' instruction," the joke goes. "Do what I mean, not what I say."

Programming is demanding. It requires perfection, consistently, for months and years of effort. At best, mistakes lead to code that won't compile. At worst, they lead to bugs that lie in wait and pounce at the moment that does the most damage.

People aren't so good at perfection. No wonder, then, that software is buggy.

Wouldn't it be cool if there were a tool that alerted you to programming mistakes moments after you made them—a tool so powerful that it virtually eliminated the need for debugging?

There is such a tool, or rather, a technique. It's test-driven development, and it actually delivers these results.

Test-driven development, or TDD, is a rapid cycle of testing, coding, and refactoring. When adding a feature, a pair may perform dozens of these cycles, implementing and refining the software in baby steps until there is nothing left to add and nothing left to take away. Research shows that TDD substantially reduces the incidence of defects [Janzen & Saiedian]. When used properly, it also helps improve your design, documents your public interfaces, and guards against future mistakes.

TDD isn't perfect, of course. (Is anything?) TDD is difficult to use on legacy codebases. Even with greenfield systems, it takes a few months of steady use to overcome the learning curve. Try it anyway—although TDD benefits from other XP practices, it doesn't require them. You can use it on almost any project.

Why TDD Works

Back in the days of punch cards, programmers laboriously hand-checked their code to make sure it would compile. A compile error could lead to failed batch jobs and intense debugging sessions to look for the misplaced character.

Getting code to compile isn't such a big deal anymore. Most IDEs check your syntax as you type, and some even compile every time you save. The feedback loop is so fast that errors are easy to find and fix. If something doesn't compile, there isn't much code to check.

Test-driven development applies the same principle to programmer intent. Just as modern compilers provide more feedback on the syntax of your code, TDD cranks up the feedback on the execution of your code. Every few minutes—as often as every 20 or 30 *seconds*—TDD verifies that the code does what you think it should do. If something goes wrong, there are only a few lines of code to check. Mistakes are easy to find and fix.

TDD uses an approach similar to double-entry bookkeeping. You communicate your intentions twice, stating the same idea in different ways: first with a test, then with production code. When they match, it's likely they were both coded correctly. If they don't, there's a mistake somewhere.

NOTE

It's theoretically possible for the test and code to both be wrong in exactly the same way, thereby making it seem like everything's OK when it's not. In practice, unless you cut-and-paste between the test and production code, this is so rare it's not worth worrying about.

In TDD, the tests are written from the perspective of a class' public interface. They focus on the class' *behavior*, not its *implementation*. Programmers write each test before the corresponding production code. This focuses their attention on creating interfaces that are easy to use rather than easy to implement, which improves the design of the interface.

After TDD is finished, the tests remain. They're checked in with the rest of the code, and they act as living documentation of the code. More importantly, programmers run all the tests with (nearly) every build, ensuring that code continues to work as originally intended. If someone accidentally changes the code's behavior—for example, with a misguided refactoring—the tests fail, signalling the mistake.

How to Use TDD

You can start using TDD today. It's one of those things that takes moments to learn and a lifetime to master.

NOTE

The basic steps of TDD are easy to learn, but the mindset takes a while to sink in. Until it does, TDD will likely seem clumsy, slow, and awkward. Give yourself two or three months of full-time TDD use to adjust.

Imagine TDD as a small, fast-spinning motor. It operates in a very short cycle that repeats over and over again. Every few minutes, this cycle ratchets your code forward a notch, providing code that—although it may not be finished—has been tested, designed, coded, and is ready to check in.

> Every few minutes, TDD provides proven code that has been tested, designed, and coded.

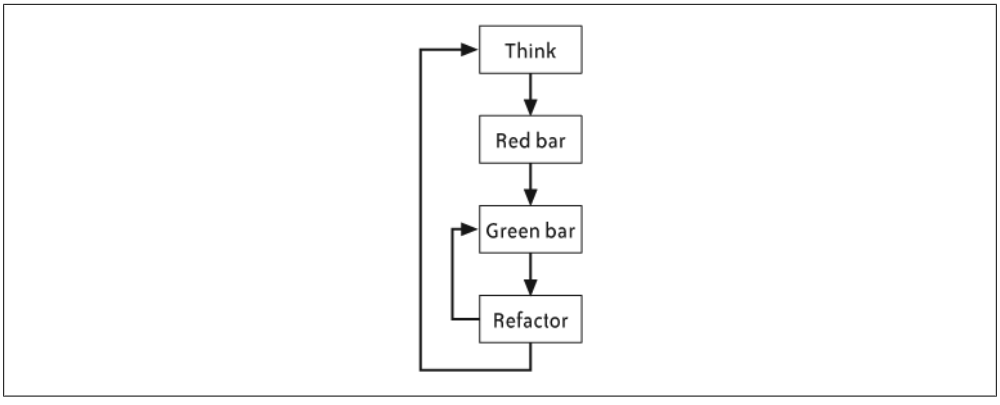


Figure 9-1. The TDD cycle

To use TDD, follow the "red, green, refactor" cycle illustrated in Figure 9-1. With experience, unless you're doing a lot of refactoring, each cycle will take fewer than five minutes. Repeat the cycle until your work is finished. You can stop and integrate whenever all your tests pass, which should be every few minutes.

Step 1: Think

TDD uses small tests to force you to write your code—you only write enough code to make the tests pass. The XP saying is, "Don't write any production code unless you have a failing test."

Your first step, therefore, is to engage in a rather odd thought process. Imagine what behavior you want your code to have, then think of a small increment that will require fewer than five lines of code. Next, think of a test—also a few lines of code—that will fail unless that behavior is present.

In other words, think of a test that will force you to add the next few lines of production code. This is the hardest part of TDD because the concept of tests driving your code seems backward, and because it can be difficult to think in small increments.

Pair programming helps. While the driver tries to make the current test pass, the navigator should stay a few steps ahead, thinking of tests that will drive the code to the next increment.

Ally

Pair Programming (p. 74)

Step 2: Red bar

Now write the test. Write only enough code for the current increment of behavior—typically fewer than five lines of code. If it takes more, that's OK, just try for a smaller increment next time.

Code in terms of the class' behavior and its public interface, not how you think you will implement the internals of the class. Respect encapsulation. In the first few tests, this often means you write your test to use method and class names that don't exist yet. This is intentional—it forces you to design your class' interface from the perspective of a user of the class, not as its implementer.

After the test is coded, run your entire suite of tests and watch the new test fail. In most TDD testing tools, this will result in a red progress bar.

This is your first opportunity to compare your intent with what's actually happening. If the test *doesn't* fail, or if it fails in a different way than you expected, something is wrong. Perhaps your test is broken, or it doesn't test what you thought it did. Troubleshoot the problem; you should always be able to predict what's happening with the code.

NOTE

It's just as important to troubleshoot unexpected *successes* as it is to troubleshoot unexpected *failures*. Your goal isn't merely to have tests that work; it's to remain in control of your code—to always know what the code is doing and why.

Step 3: Green bar

Next, write just enough production code to get the test to pass. Again, you should usually need less than five lines of code. Don't worry about design purity or conceptual elegance—just do what you need to do to make the test pass. Sometimes you can just hardcode the answer. This is OK because you'll be refactoring in a moment.

Run your tests again, and watch all the tests pass. This will result in a green progress bar.

> When your tests fail and you can't figure out why, revert to known-good code.

This is your second opportunity to compare your intent with reality. If the test fails, get back to known-good code as quickly as you can. Often, you or your pairing partner can see the problem by taking a second look at the code you just wrote. If you can't see the problem, consider erasing the new code and trying again. Sometimes it's best to delete the new test (it's only a few lines of code, after all) and start the cycle over with a smaller increment.

NOTE

Remaining in control is key. It's OK to back up a few steps if that puts you back in control of your code. If you can't bring yourself to revert right away, set a 5- or 10-minute timer. Make a deal with your pairing partner that you'll revert to known-good code if you haven't solved the problem when the timer goes off.

Step 4: Refactor

With all your tests passing again, you can now refactor without worrying about breaking anything. Review the code and look for possible improvements. Ask your navigator if he's made any notes.

Ally

Refactoring (p. 306)

For each problem you see, refactor the code to fix it. Work in a series of very small refactorings—a minute or two each, certainly not longer than five minutes—and run the tests after each one. They should always pass. As before, if the test doesn't pass and the answer isn't immediately obvious, undo the refactoring and get back to known-good code.

Refactor as many times as you like. Make your design as good as you can, but limit it to the code's existing behavior. Don't anticipate future needs, and certainly don't add new behavior.

Remember, refactorings aren't supposed to change behavior. New behavior requires a failing test.

Step 5: Repeat

When you're ready to add new behavior, start the cycle over again.

Each time you finish the TDD cycle, you add a tiny bit of well-tested, well-designed code. The key to success with TDD is *small increments*. Typically, you'll run through several cycles very quickly, then spend more time on refactoring for a cycle or two, then speed up again.

> The key to TDD is small increments.

With practice, you can finish more than 20 cycles in an hour. Don't focus too much on how fast you go, though. That might tempt you to skip refactoring and design, which are too important to skip. Instead, take very small steps, run the tests frequently, and minimize the time you spend with a red bar.

A TDD Example

I recently recorded how I used TDD to solve a sample problem. The increments are very small—they may even seem ridiculously small—but this makes finding mistakes easy, and that helps me go faster.

> NOTE
> **Programmers new to TDD are often surprised at how small each increment can be. Although you might think that only beginners need to work in small steps, my experience is the reverse: the more TDD experience you have, the smaller steps you take and the faster you go.**

As you read, keep in mind that it takes far longer to explain an example like this than to program it. I completed each step in a matter of seconds.

The task

Imagine you need to program a Java class to parse an HTTP query string.* You've decided to use TDD to do so.

One name/value pair

Step 1: Think. The first step is to imagine the features you want the code to have. My first thought was, "I need my class to separate name/value pairs into a `HashMap`." Unfortunately, this would take more than five lines to code, so I needed to think of a smaller increment.

Often, the best way to make the increments smaller is to start with seemingly trivial cases. "I need my class to put *one* name/value pair into a `HashMap`," I thought, which sounded like it would do the trick.

* Pretend you're in an alternate reality without a gazillion libraries that already do this.

Step 2: Red Bar. The next step is to write the test. Remember, this is partially an exercise in interface design. In this example, my first temptation was to call the class `QueryStringParser`, but that's not very object-oriented. I settled on `QueryString`.

As I wrote the test, I realized that a class named `QueryString` wouldn't return a `HashMap`; it would *encapsulate* the `HashMap`. It would provide a method such as `valueFor(name)` to access the name/value pairs.

> NOTE
> **Thinking about the test forced me to figure out how I wanted to design my code.**

Building that seemed like it would require too much code for one increment, so I decided to have this test to drive the creation of a `count()` method instead. I decided that the `count()` method would return the total number of name/value pairs. My test checked that it would work when there was just one pair.

```
            public void testOneNameValuePair() {
    QueryString qs = new QueryString("name=value");
    assertEquals(1, qs.count());
}
```

The code didn't compile, so I wrote a do-nothing `QueryString` class and `count()` method.

```
            public class QueryString {
    public QueryString(String queryString) {}
    public int count() { return 0; }
}
```

That gave me the red bar I expected.

Step 3: Green Bar. To make this test pass, I hardcoded the right answer. I could have programmed a better solution, but I wasn't sure how I wanted the code to work. Would it count the number of equals signs? Not all query strings have equals signs. I decided to punt.

```
public int count() { return 1; }
```

Green bar.

> NOTE
> **Although I had ideas about how I would finish writing this class, I didn't commit myself to any particular course of action in advance. I remained open to discovering new approaches as I worked.**

Step 4: Refactor. I didn't like the `QueryString` name, but I had another test in mind and I was eager to get to it. I made a note to fix the name on an index card—perhaps `HttpQuery` would be better. I'd see how I felt next time through the cycle.

Step 5: Repeat. Yup.

An empty string

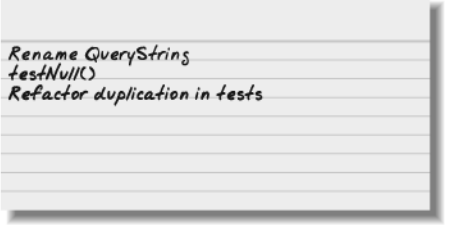
Think. I wanted to force myself to get rid of that hardcoded `return 1`, but I didn't want to have to deal with multiple query strings yet. My next increment would probably be about the `valueFor()` method, and I wanted to avoid the complexity of multiple queries. I decided that testing an empty string would require me to code `count()` properly without making future increments too difficult.

Red Bar. New test.

```
            public void testNoNameValuePairs() {
        QueryString qs = new QueryString("");
        assertEquals(0, qs.count());
    }
```

Red bar. Expected: `<0>` but was: `<1>`. No surprise there.

This inspired two thoughts. First, that I should test the case of a `null` argument to the `QueryString` constructor. Second, because I was starting to see duplication in my tests, I should refactor that. I added both notes to my index card.



Green bar. Now I had to stop and think. What was the fastest way for me to get back to a green bar? I decided to check if the query string was blank.

```
public class QueryString {
    private String _query

    public QueryString(string queryString) {
        _query = queryString;
    }

    public int count() {
        if ("".equals(_query)) return 0;
        else return 1;
```

```
    }
}
```

Refactor. I double-checked my to-do list . I needed to refactor the tests, but I decided to wait for another test to demonstrate the need. "Three strikes and you refactor," as the saying goes.

It was time to do the cycle again.

testNull()

Think. My list included `testNull()`, which meant I needed to test the case when the query string is null. I decided to put that in.

Red Bar. This test forced me to think about what behavior I wanted when the value was null. I've always been a fan of code that fails fast, so I decided that a null value was illegal. This meant the code should throw an exception telling callers not to use null values. ("Simple Design"," later in this chapter, discusses failing fast in detail.)

```
            public void testNull() {
        try {
            QueryString qs = new QueryString(null)
            fail("Should throw exception");
        }
        catch (NullPointerException e) {
            // expected
        }
    }
```

Green Bar. Piece of cake.

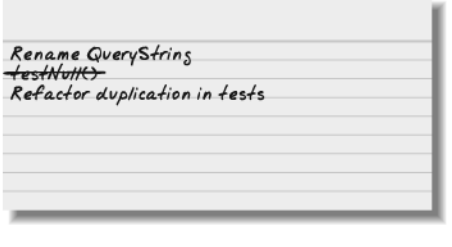
```
    public QueryString(String queryString) {
        if (queryString == null) throw new NullPointerException();

        _query = queryString;
    }
```

Refactor. I still needed to refactor my tests, but the new test didn't have enough in common with the old tests to make me feel it was necessary. The production code looked OK, too, and there wasn't anything significant on my index card. No refactorings this time.

> NOTE
>
> **Although I don't refactor on every cycle, I always stop and seriously consider whether my design needs refactoring.**

valueFor()



Think. OK, now what? The easy tests were done. I decided to put some meat on the class and implement the `valueFor()` method. Given a name of a name/value pair in the query string, this method would return the associated value.

As I thought about this test, I realized I also needed a test to show what happens when the name doesn't exist. I wrote that on my index card for later.

Red Bar. To make the tests fail, I added a new assertion at the end of my existing testOneNameValuePair() test.

```
public void testOneNameValuePair() {
    QueryString qs = new QueryString("name=value");
    assertEquals(1, qs.count());
    assertEquals("value", qs.valueFor("name"));
}
```

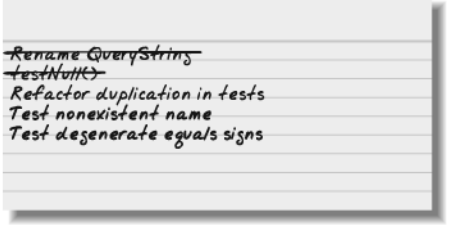
Green Bar. This test made me think for a moment. Could the `split()` method work? I thought it would.

```
            public String valueFor(String name) {
    String[] nameAndValue = _query.split("=");
    return nameAndValue[1];
}
```

This code passed the tests, but it was incomplete. What if there was more than one equals sign, or no equals signs? It needed proper error handling. I made a note to add tests for those scenarios.

Refactor. The names were bugging me. I decided that `QueryString` was OK, if not perfect. The `qs` in the tests was sloppy, so I renamed it `query`.

Multiple name/value pairs



Think. I had a note reminding me to take care of error handling in `valueFor()`, but I wanted to tackle something more meaty. I decided to add a test for multiple name/value pairs.

Red Bar. When dealing with a variable number of items, I usually test the case of zero items, one item, and three items. I had already tested zero and one, so now I tested three.

```
            public void testMultipleNameValuePairs() {
    QueryString query = new QueryString("name1=value1&name2=value2&name3=value3");
    assertEquals("value1", query.valueFor("name1"));
    assertEquals("value2", query.valueFor("name2"));
    assertEquals("value3", query.valueFor("name3"));
}
```

I could have written an assertion for `count()` rather than `valueFor()`, but the real substance was in the `valueFor()` method. I made a note to test `count()` next.

Green Bar. My initial thought was that the `split()` technique would work again.

```
public String valueFor(String name) {
    String[] pairs = _query.split("&");
    for (String pair : pairs) {
        String[] nameAndValue = pair.split("=");
        if (nameAndValue[0].equals(name)) return nameAndValue[1];
    }
    throw new RuntimeException(name + " not found");
}
```

Ewww... that felt like a hack. But the test passed!

> NOTE
>
> **It's better to get to a green bar quickly than to try for perfect code. A green bar keeps you in control of your code and allows you to experiment with refactorings that clean up your hack.**

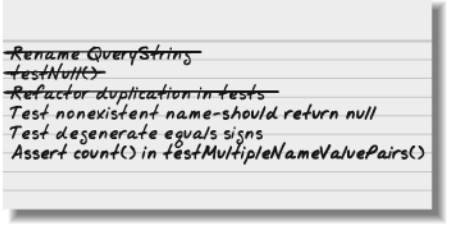
Refactor: The additions to `valueFor()` felt hackish. I took a second look. The two issues that bothered me the most were the `nameAndValue` array and the `RuntimeException`. An attempt to refactor `nameAndValue` led to worse code, so I backed out the refactoring and decided to leave it alone for another cycle.

The `RuntimeException` was worse; it's better to throw a custom exception. In this case, though, the Java convention is to return `null` rather than throw an exception. I already had a note that I should test the case where `name` isn't found; I revised it to say that the correct behavior was to return `null`.

Reviewing further, I saw that my duplicated test logic had reached three duplications. Time to refactor—or so I thought. After a second look, I realized that the only duplication between the tests was that I was constructing a `QueryString` object each time. Everything else was different, including `QueryString`'s constructor parameters. The tests didn't need refactoring after all. I scratched that note off my list.

In fact, the code was looking pretty good... better than I initially thought, at least. I'm too hard on myself.

Multiple count()



Think. After reviewing my notes, I realized I should probably test degenerate uses of the ampersand, such as two ampersands in a row. I made a note to add tests for that. At the time, though, I wanted to get the `count()` method working properly with multiple name/value pairs.

Red Bar. I added the `count()` assertion to my test. It failed as expected.

```
public void testMultipleNameValuePairs() {
    QueryString query = new QueryString("name1=value1&name2=value2&name3=value3");
    assertEquals(3, query.count());
    assertEquals("value1", query.valueFor("name1"));
    assertEquals("value2", query.valueFor("name2"));
    assertEquals("value3", query.valueFor("name3"));
}
```

Green Bar. To get this test to pass, I stole my existing code from `valueFor()` and modified it. This was blatant duplication, but I planned to refactor as soon as I saw the green bar.

```
public int count() {
    String[] pairs = _query.split("&");
    return pairs.length;
}
```

I was able to delete more of the copied code than I expected. To my surprise, however, it didn't pass! The test failed in the case of an empty query string: `expected: <0> but was: <1>`. I had forgotten that `split()` returned the original string when the split character isn't found. My code expected it to return an empty array when no split occurred.

I added a guard clause that took care of the problem. It felt like a hack, so I planned to take a closer look after the tests passed.

```
public int count() {
    if ("".equals(_query)) return 0;
    String[] pairs = _query.split("&");
    return pairs.length;
}
```

Refactor. This time I definitely needed to refactor. The duplication between `count()` and `valueFor()` wasn't too strong—it was just one line—but they both parsed the query string, which was a duplication of function if not code. I decided to fix it.

At first, I wasn't sure how to fix the problem. I decided to try to parse the query string into a `HashMap`, as I had originally considered. To keep the refactoring small, I left `count()` alone at first and just modified `valueFor()`. It was a small change.

```
public String valueFor(String name) {
    HashMap<String, String> map = new HashMap<String, String>();

    String[] pairs = _query.split("&");
    for (String pair : pairs) {
        String[] nameAndValue = pair.split("=");
        map.put(nameAndValue[0], nameAndValue[1]);
    }
    return map.get(name);
}
```

> NOTE
> **This refactoring eliminated the exception that I threw when `name` wasn't found. Technically, it changed the behavior of the program. However, because I hadn't yet written a test for that behavior, I didn't care. I made sure I had a note to test that case later (I did) and kept going.**

This code parsed the query string during every call to `valueFor()`, which wasn't a great idea. I had kept the code in `valueFor()` to keep the refactoring simple. Now I wanted to move it out of `valueFor()` into the constructor. This required a sequence of refactorings, as described in "Refactoring"," later in this chapter.

I reran the tests after each of these refactorings to make sure that I hadn't broken anything... and in fact, one refactoring did break the tests. When I called the parser from the constructor, `testNoNameValuePairs()`—the empty query test—bit me again, causing an exception in the parser. I added a guard clause as before, which solved the problem.

After all that refactoring, the tests and production code were nice and clean.

```
public class QueryStringTest extends TestCase {
    public void testOneNameValuePair() {
        QueryString query = new QueryString("name=value");
        assertEquals(1, query.count());
        assertEquals("value", query.valueFor("name"));
    }

    public void testMultipleNameValuePairs() {
        QueryString query = new QueryString("name1=value1&name2=value2&name3=value3");
        assertEquals(3, query.count());
        assertEquals("value1", query.valueFor("name1"));
        assertEquals("value2", query.valueFor("name2"));
        assertEquals("value3", query.valueFor("name3"));
    }

    public void testNoNameValuePairs() {
        QueryString query = new QueryString("");
        assertEquals(0, query.count());
    }

    public void testNull() {
        try {
            QueryString query = new QueryString(null);
            fail("Should throw exception");
        }
        catch (NullPointerException e) {
            // expected
        }
    }
}

public class QueryString {
    private HashMap<String, String> _map = new HashMap<String, String>();

    public QueryString(String queryString) {
        if (queryString == null) throw new NullPointerException();
        parseQueryString(queryString);
    }

    public int count() {
        return _map.size();
```

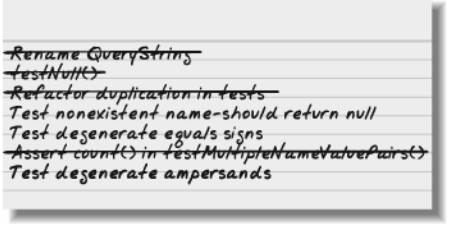
```
    }

    public String valueFor(String name) {
        return _map.get(name);
    }

    private void parseQueryString(String query) {
        if ("".equals(query)) return;

        String[] pairs = query.split("&");
        for (String pair : pairs) {
            String[] nameAndValue = pair.split("=");
            _map.put(nameAndValue[0], nameAndValue[1]);
        }
    }
}
```

Your turn



The class wasn't done—it still needed to handle degenerate uses of the equals and ampersand signs, and it didn't fully implement the query string specification yet, either.* In the interest of space, though, I leave the remaining behavior as an exercise for you to complete yourself. As you try it, remember to take very small steps and to check for refactorings every time through the cycle.

Testing Tools

To use TDD, you need a testing framework. The most popular are the open source *xUnit* tools, such as *JUnit* (for Java) and *NUnit* (for .NET). Although these tools have different authors, they typically share the basic philosophy of Kent Beck's pioneering SUnit.

> NOTE
> **Instructions for specific tools are out of the scope of this book. Introductory guides for each tool are easily found online.**

If your platform doesn't have an xUnit tool, you can build your own. Although the existing tools often provide GUIs and other fancy features, none of that is necessary. All you need is a way to run all your test methods as a single suite, a few assert methods, and an unambiguous pass or fail result when the test suite is done.

* For example, the semicolon works like the ampersand in query strings.

Speed Matters

As with programming itself, TDD has myriad nuances to master. The good news is that the basic steps alone—red, green, refactor—will lead to very good results. Over time, you'll fine-tune your approach.

One of the nuances of TDD is test speed—not the frequency of each increment, which is also important, but how long it takes to run all the tests. In TDD, you run the tests as often as one or two times every *minute*. They must be fast. If they aren't, they'll be a distraction rather than a help. You won't run them as frequently, which reduces the primary benefit of TDD: micro-increments of proof.

[Nielsen] reports that users lose interest and switch tasks when the computer makes them wait more than 10 seconds. Computers only seem "fast" when they make users wait less than a second.

> Make sure your tests take under 10 seconds to run.

Although Nielsen's research explored the area of user interface design, I've found it to be true when running tests as well. If they take more than 10 seconds, I'm tempted to check my email, surf the Web, or otherwise distract myself. Then it takes several minutes for me to get back to work. To avoid this delay, make sure your tests take under 10 seconds to run. Less than a second is even better.

An easy way to keep your test times down is to run a subset of tests during the TDD cycle. Periodically run the whole test suite to make sure you haven't accidentally broken something, particularly before integrating and during refactorings that affect multiple classes.

Running a subset does incur the risk that you'll make a mistake without realizing it, which leads to annoying debugging problems later. Advanced practitioners design their tests to run quickly. This requires that they make trade-offs between three basic types of automated tests:

- Unit tests, which run at a rate of hundreds per second
- Focused integration tests, which run at a rate of a handful per second
- End-to-end tests, which often require seconds per test

The vast majority of your tests should be unit tests. A small fraction should be integration tests, and only a few should be end-to-end tests.

Unit Tests

Unit tests focus just on the class or method at hand. They run entirely in memory, which makes them very fast. Depending on your platform, your testing tool should be able to run at least 100 unit tests *per second*.

[Feathers] provides an excellent definition of a unit test:

> Unit tests run fast. If they don't run fast, they aren't unit tests.
>
> Other kinds of tests often masquerade as unit tests. A test is not a unit test if:
>
> 1. It talks to a database
> 2. It communicates across a network

3. It touches the file system
4. You have to do special things to your environment (such as editing configuration files) to run it

Tests that do these things are integration tests, not unit tests.

Creating unit tests requires good design.A highly coupled system—a *big ball of mud,* or *spaghetti software*—makes it difficult to write unit tests. If you have trouble doing this, or if Feathers' definition seems impossibly idealistic, it's a sign of problems in your design. Look for ways to decouple your code so that each class, or set of related classes, may be tested in isolation. See "Simple Design"" later in this chapter for ideas, and consider asking your mentor for help.

MOCK OBJECTS

Mock objects are a popular tool for isolating classes for unit testing. When using mock objects, your test substitutes its own object (the "mock object") for an object that talks to the outside world. The mock object checks that it is called correctly and provides a pre-scripted response. In doing so, it avoids time-consuming communication to a database, network socket, or other outside entity.

Beware of mock objects. They add complexity and tie your test to the implementation of your code. When you're tempted to use a mock object, ask yourself if there's a way you could improve the design of your code so that a mock object isn't necessary. Can you decouple your code from the external dependency more cleanly? Can you provide the data it needs—in the constructor, perhaps—rather than having it get the data itself?

Mock objects are a useful technique, and sometimes they're the best way to test your code. Before you assume that a mock object is appropriate for your situation, however, take a second look at your design. You might have an opportunity for improvement.

Focused Integration Tests

Unit tests aren't enough. At some point, your code has to talk to the outside world. You can use TDD for that code, too.

A test that causes your code to talk to a database, communicate across the network, touch the file system, or otherwise leave the bounds of its own process is an *integration test*. The best integration tests are *focused integration tests*, which test just one interaction with the outside world.

> NOTE
> You may think of an integration test as a test that checks that the whole system fits together properly. I call that kind of test an *end-to-end test*.

One of the challenges of using integration tests is the need to prepare the external dependency to be tested. Tests should run exactly the same way every time, regardless of which order you run them in or the state of the machine prior to running them. This is easy with unit tests but

harder with integration tests. If you're testing your ability to select data from a database table, that data needs to be in the database.

Make sure each integration test can run entirely on its own. It should set up the environment it needs and then restore the previous environment afterwards. Be sure to do so even if the test fails or an exception is thrown. Nothing is more frustrating than a test suite that intermittently fails. Integration tests that don't set up and tear down their test environment properly are common culprits.

> Make sure each test is isolated from the others.

> NOTE
> **If you have a test that fails intermittently, don't ignore it, even if you can "fix" the failure by running the tests twice in a row. Intermittent failures are an example of technical debt. They make your tests more frustrating to run and disguise real failures.**

You shouldn't need many integration tests. The best integration tests have a narrow focus; each checks just one aspect of your program's ability to talk to the outside world. The number of focused integration tests in your test suite should be proportional to the types of external interactions your program has, not the overall size of the program. (In contrast, the number of unit tests you have *is* proportional to the overall size of the program.)

If you need a lot of integration tests, it's a sign of design problems. It may mean that the code that talks to the outside world isn't cohesive. For example, if all your business objects talk directly to a database, you'll need integration tests for each one. A better design would be to have just one class that talks to the database. The business objects would talk to that class.* In this scenario, only the database class would need integration tests. The business objects could use ordinary unit tests.

End-to-End Tests

In a perfect world, the unit tests and focused integration tests mesh perfectly to give you total confidence in your tests and code. You should be able to make any changes you want without fear, comfortable in the knowledge that if you make a mistake, your tests will catch them.

How can you be sure your unit tests and integration tests mesh perfectly? One way is to write end-to-end tests. End-to-end tests exercise large swaths of the system, starting at (or just behind) the user interface, passing through the business layer, touching the database, and returning. Acceptance tests and functional tests are common examples of end-to-end tests. Some people also call them integration tests, although I reserve that term for focused integration tests.

End-to-end tests can give you more confidence in your code, but they suffer from many problems. They're difficult to create because they require error-prone and labor-intensive setup and teardown procedures. They're brittle and tend to break whenever any part of the system or its setup data changes. They're very slow—they run in seconds or even minutes per test,

* A still better design might involve a persistence layer.

rather than multiple tests per second. They provide a false sense of security, by exercising so many branches in the code that it's difficult to say which parts of the code are actually covered.

Instead of end-to-end tests, use exploratory testing to check the effectiveness of your unit and integration tests. When your exploratory tests find a problem, use that information to improve your approach to unit and integration testing, rather than introducing end-to-end tests.

Ally

Exploratory Testing (p. 342)

NOTE

Don't use exploratory testing to find bugs; use it to determine if your unit tests and integration tests mesh properly. When you find an issue, improve your TDD strategy.

In some cases, limitations in your design may prevent unit and integration tests from testing your code sufficiently. This often happens when you have legacy code. In that case, end-to-end tests are a necessary evil. Think of them as technical debt: strive to make them unecessary, and replace them with unit and integration tests whenever you have the opportunity.

TDD and Legacy Code

[Feathers] says *legacy code* is "code without tests." I think of it as "code you're afraid to change." This is usually the same thing.

The challenge of legacy code is that, because it was created without tests, it usually isn't designed for testability. In order to introduce tests, you need to change the code. In order to change the code with confidence, you need to introduce tests. It's this kind of chicken-and-egg problem that makes legacy code so difficult to work with.

To make matters worse, legacy code has often accumulated a lot of technical debt. (It's hard to remove technical debt when you're afraid to change existing code.) You may have trouble understanding how everything fits together. Methods and functions may have side effects that aren't apparent.

One way to approach this problem is to introduce end-to-end *smoke tests*. These tests exercise common usage scenarios involving the component you want to change. They aren't sufficient to give you total confidence in your changes, but they at least alert you when you make a big mistake.

With the smoke tests in place, you can start introducing unit tests. The challenge here is finding isolated components to test, as legacy code is often tightly coupled code. Instead, look for ways for your test to strategically interrupt program execution. [Feathers] calls these opportunities *seams*. For example, in an object-oriented language, if a method has a dependency you want to avoid, your test can call a test-specific subclass that overrides and stubs out the offending method.

Finding and exploiting seams often leads to ugly code. It's a case of temporarily making the code worse so you can then make it better. Once you've introduced tests, refactor the code to make it test-friendly, then improve your tests so they aren't so ugly. Then you can proceed with normal TDD.

Adding tests to legacy code is a complex subject that deserves its own book. Fortunately, [Feathers]' *Working Effectively with Legacy Code* is exactly that book.

Questions

What do I need to test when using TDD?

The saying is, "Test everything that can possibly break." To determine if something could possibly break, I think, "Do I have absolute confidence that I'm doing this correctly, and that nobody in the future will inadvertently break this code?"

I've learned through painful experience that I can break nearly anything, so I test nearly everything. The only exception is code without any logic, such as simple accessors and mutators (getters and setters), or a method that only calls another method.

You *don't* need to test third-party code unless you have some reason to distrust it.

How do I test private methods?

As I did in my extended `QueryString` example, start by testing public methods. As you refactor, some of that code will move into private methods, but the existing tests will still thoroughly test its behavior.

If your code is so complex that you need to test a private method directly, this may be a sign that you should refactor. You may benefit from moving the private methods into their own class and providing a public interface. The "Replace Method with Method Object" refactoring [Fowler 1999] (p. 135) may help.

How can I use TDD when developing a user interface?

TDD is particularly difficult with user interfaces because most UI frameworks weren't designed with testability in mind. Many people compromise by writing a very thin, untested translation layer that only forwards UI calls to a presentation layer. They keep all their UI logic in the presentation layer and use TDD on that layer as normal.

There are some tools that allow you to test a UI directly, perhaps by making HTTP calls (for web-based software), or by pressing buttons or simulating window events (for client-based software). These are essentially integration tests, and they suffer similar speed and maintainability challenges as other integration tests. Despite the challenges, these tools can be helpful.

You talked about refactoring your test code. Does anyone really do this?

Yes. I do, and everybody should. Tests are just code. The normal rules of good development apply: avoid duplication, choose good names, factor, and design well.

I've seen otherwise-fine projects go off the rails because of brittle and fragile test suites. By making TDD a central facet of development, you've committed to maintaining your test code just as much as you've committed to maintaining the rest of the code. Take it just as seriously.

Results

When you use TDD properly, you find that you spend little time debugging. Although you continue to make mistakes, you find those mistakes very quickly and have little difficulty fixing them. You have total confidence that the whole codebase is well-tested, which allows you to

aggressively refactor at every opportunity, confident in the knowledge that the tests will catch any mistakes.

Contraindications

Although TDD is a very valuable tool, it does have a two- or three-month learning curve. It's easy to apply to toy problems such as the `QueryString` example, but translating that experience to larger systems takes time. Legacy code, proper unit test isolation, and integration tests are particularly difficult to master. On the other hand, the sooner you start using TDD, the sooner you'll figure it out, so don't let these challenges stop you.

Be careful when applying TDD without permission. Learning TDD could slow you down temporarily. This could backfire and cause your organization to reject TDD without proper consideration. I've found that combining testing time with development time when providing estimates helps alleviate pushback for dedicated developer testing.

Also be cautious about being the only one to use TDD on your team. You may find that your teammates break your tests and don't fix them. It's better to get the whole team to agree to try it together.

Alternatives

TDD is the heart of XP's programming pratices. Without it, all of XP's other technical practices will be much harder to use.

A common misinterpretation of TDD is to design your entire class first, then write all its test methods, then write the code. This approach is frustrating and slow, and it doesn't allow you to learn as you go.

Another misguided approach is to write your tests after you write your production code. This is very difficult to do well—production code must be designed for testability, and it's hard to do so unless you write the tests first. It doesn't help that writing tests after the fact is boring. In practice, the temptation to move on to the next task usually overwhelms the desire for well-tested code.

Although you can use these alternatives to introduce tests to your code, TDD isn't just about testing. It's really about using very small increments to produce high-quality, known-good code. I'm not aware of any alternatives that provide TDD's ability to catch and fix mistakes quickly.

Further Reading

Test-driven development is one of the most heavily explored aspects of Extreme Programming. There are several excellent books on various aspects of TDD. Most are focused on Java and JUnit, but their ideas are applicable to other languages as well.

Test-Driven Development: By Example [Beck 2002] is a good introduction to TDD. If you liked the `QueryString` example, you'll like the extended examples in this book. The TDD patterns in Part III are particularly good.

Test-Driven Development: A Practical Guide [Astels] provides a larger example that covers a complete project. Reviewers praise its inclusion of UI testing.

JUnit Recipes [Rainsberger] is a comprehensive book discussing a wide variety of testing problems, including a thorough discussion of testing J2EE.

Working Effectively with Legacy Code [Feathers] is a must-have for anybody working with legacy code.

Refactoring

Audience
Programmers

Every day, our code is slightly better than it was the day before.

Entropy always wins. Eventually, chaos turns your beautifully imagined and well-designed code into a big mess of spaghetti.

At least, that's the way it used to be, before refactoring. *Refactoring* is the process of changing the design of your code without changing its behavior—*what* it does stays the same, but *how* it does it changes. Refactorings are also reversible; sometimes one form is better than another for certain cases. Just as you can change the expression `x² - 1` to `(x + 1)(x - 1)` and back, you can change the design of your code—and once you can do that, you can keep entropy at bay.

Reflective Design

Refactoring enables an approach to design I call *reflective design*. In addition to creating a design and coding it, you can now analyze the design of existing code and improve it. One of the best ways to identify improvements is with *code smells*: condensed nuggets of wisdom that help you identify common problems in design.

A code smell doesn't necessarily mean there's a problem with the design. It's like a funky smell in the kitchen: it could indicate that it's time to take out the garbage, or it could just mean that Uncle Gabriel is cooking with a particularly pungent cheese. Either way, when you smell something funny, take a closer look.

[Fowler 1999], writing with Kent Beck, has an excellent discussion of code smells. It's well worth reading. Here are a summary of the smells I find most often, including some that Fowler and Beck didn't mention.*

Divergent Change and Shotgun Surgery

These two smells help you identify cohesion problems in your code. *Divergent Change* occurs when unrelated changes affect the same class. It's an indication that your class involves too many concepts. Split it, and give each concept its own home.

Shotgun Surgery is just the opposite: it occurs when you have to modify multiple classes to support changes to a single idea. This indicates that the concept is represented in many places throughout your code. Find or create a single home for the idea.

* Wannabee Static, Time Dependency, Half-Baked Objects, and Coddling Nulls are new in this book.

Primitive Obsession and Data Clumps

Some implementations represent high-level design concepts with primitive types. For example, a `decimal` might represent dollars. This is the *Primitive Obsession* code smell. Fix it by encapsulating the concept in a class.

Data Clumps are similar. They occur when several primitives represent a concept as a group. For example, several strings might represent an address. Rather than being encapsulated in a single class, however, the data just clumps together. When you see batches of variables consistently passed around together, you're probably facing a Data Clump. As with Primitive Obsession, encapsulate the concept in a single class.

Data Class and Wannabee Static Class

One of the most common mistakes I see in object-oriented design is when programmers put their data and code in separate classes. This often leads to duplicate data-manipulation code. When you have a class that's little more than instance variables combined with accessors and mutators (getters and setters), you have a *Data Class*. Similarly, when you have a class that contains methods but no meaningful per-object state, you have a *Wannabee Static Class*.

> NOTE
> **One way to detect a Wannabee Static Class is to ask yourself if you could change all of the methods and instance variables into statics (also called class methods and variables) without breaking anything.**

Ironically, one of the primary strengths of object-oriented programming is its ability to combine data with the code that operates on that data. Data Classes and Wannabee Statics are twins separated at birth. Reunite them by combining instance variables with the methods that operate on those variables.

Coddling Nulls

Null references pose a particular challenge to programmers: they're occasionally useful, but most they often indicate invalid states or other error conditions. Programmers are usually unsure what to do when they receive a null reference; their methods often check for null references and then return `null` themselves.

Coddling Nulls like this leads to complex methods and error-prone software. Errors suppressed with `null` cascade deeper into the application, causing unpredictable failures later in the execution of the software. Sometimes the `null` makes it into the database, leading to recurring application failures.

Instead of Coddling Nulls, adopt a fail fast strategy (see "Simple Design"" later in this chapter). Don't allow `null` as a parameter to any method, constructor, or property unless it has explicitly defined semantics. Throw exceptions to signify errors rather than returning `null`. Make sure that any unexpected null reference will cause the code to throw an exception, either as a result of being dereferenced or by hitting an explicit assertion.

Time Dependencies and Half-Baked Objects

Time Dependencies occur when a class' methods must be called in a specific order. *Half-Baked Objects* are a special case of Time Dependency: they must first be constructed, then initialized with a method call, then used.

Time Dependencies often indicate an encapsulation problem. Rather than managing its state itself, the class expects its callers to manage some of its state. This results in bugs and duplicate code in callers. Look for ways to encapsulate the class' state more effectively. In some cases, you may find that your class has too many responsibilities and would benefit from being split into multiple classes.

Analyzing Existing Code

Reflective design requires that you understand the design of existing code. The easiest way to do so is to ask someone else on the team. A conversation around a whiteboard design sketch is a great way to learn.

In some cases, no one on the team will understand the design, or you may wish to dive into the code yourself. When you do, focus on the *responsibilities* and *interactions* of each major component. What is the purpose of this package or namespace? What does this class represent? How does it interact with other packages, namespaces, and classes?

For example, NUnitAsp is a tool for unit testing ASP.NET code-behind logic. One of its classes is `HttpClient`, which you might infer makes calls to an HTTP (web) server—presumably an ASP.NET web server.

To confirm that assumption, look at the names of the class' methods and instance variables. `HttpClient` has methods named `GetPage`, `FollowLink`, `SubmitForm`, and `HasCookie`, along with some `USER_AGENT` constants and several related methods and properties. In total, it seems pretty clear that `HttpClient` emulates a web browser.

Now expand that understanding to related elements. Which classes does this class depend on? Which classes depend on this one? What are their responsibilities? As you learn, diagram your growing understanding on a whiteboard.

Creating a *UML sequence diagram** can be helpful for understanding how individual methods interact with the rest of the system. Start with a method you want to understand and look at each line in turn, recursively adding each called method to your sequence diagram. This is fairly time-consuming, so I only recommend it if you're confused about how or why a method works.

> NOTE
> ***Round-trip engineering*** **tools will automatically generate UML diagrams by analyzing source code. I prefer to generate my diagrams by hand on a whiteboard. My goal isn't merely to create a diagram—my true goal is to understand the design. Creating the diagram by hand requires me to study the code more deeply, which allows me to learn and understand more.**

* [Fowler & Scott] is a good resource for learning more about UML.

Although these techniques are useful for understanding the design of unfamiliar code, you shouldn't need them often. You should already have a good understanding of most parts of your software, or be able to talk to someone on the team who does. Unless you're dealing with a lot of legacy code, you should rarely have trouble understanding the design of existing code: your team wrote it, after all. If you find yourself needing these techniques often, something is wrong—ask your mentor for help.

How to Refactor

Reflective design helps you understand what to change; refactoring gives you the ability to make those changes. When you refactor, proceed in a series of small transformations. (Confusingly, each type of transformation is also called a *refactoring*.) Each refactoring is like making a turn on a Rubik's cube. To achieve anything significant, you have to string together several individual refactorings, just as you have to string together several turns to solve the cube.

> Refactoring isn't rewriting.

The fact that refactoring is a sequence of small transformations sometimes gets lost on people new to refactoring. Refactoring isn't rewriting. You don't just change the design of your code; to refactor well, you need to make that change in a series of controlled steps. Each step should only take a few moments, and your tests should pass after each one.

Ally

Test-Driven Development (p. 287)

There are a wide variety of individual refactorings. [Fowler 1999]'s *Refactoring* is the classic work on the subject. It contains an in-depth catalog of refactoring, and is well worth studying—I learned more about good code from reading that book than from any other.

You don't need to memorize all the individual refactorings. Instead, try to learn the mindset behind the refactorings. Work from the book in the beginning. Over time, you'll learn how to refactor intuitively, creating each refactoring as you need it.

NOTE

Some IDEs offer automated refactorings. Although this is helpful for automating some tedious refactorings, learn how to refactor manually, too. There are many more refactoring options available to you than your IDE provides.

Refactoring in Action

Any significant design change requires a sequence of refactorings. Learning how to change your design through a series of small refactorings is a valuable skill. Once you've mastered it, you can make dramatic design changes without breaking anything. You can even do this in pieces, by fixing part of the design one day and the rest of it another day.

To illustrate this point, I'll show each step in the simple refactoring from my TDD example (see the TDD example in "Test-Driven Development"" earlier in this chapter). Note how small each step is. Working in small steps enables you to remain in control of the code, prevents confusion, and allows you to work very quickly.

> **NOTE**
> **Don't let the small scale of this example distract you from the main point: making changes in a series of small, controlled steps. For larger examples, see "Further Reading"" at the end of this section.**

The purpose of this example was to create an HTTP query string parser. At this point, I had a working, bare-bones parser (see "A TDD Example"" earlier in this chapter). Here are the tests:

```
public class QueryStringTest extends TestCase {

    public void testOneNameValuePair() {
        QueryString query = new QueryString("name=value");
        assertEquals(1, query.count());
        assertEquals("value", query.valueFor("name"));
    }

    public void testMultipleNameValuePairs() {
        QueryString query = new QueryString("name1=value1&name2=value2&name3=value3");
        assertEquals(3, query.count());
        assertEquals("value1", query.valueFor("name1"));
        assertEquals("value2", query.valueFor("name2"));
        assertEquals("value3", query.valueFor("name3"));
    }

    public void testNoNameValuePairs() {
        QueryString query = new QueryString("");
        assertEquals(0, query.count());
    }

    public void testNull() {
        try {
            QueryString query = new QueryString(null);
            fail("Should throw exception");
        }
        catch (NullPointerException e) {
            // expected
        }
    }
}
```

The code worked—it passed all the tests—but it was ugly. Both the `count()` and `valueFor()` methods had duplicate parsing code. I wanted to eliminate this duplication and put parsing in just one place.

```
public class QueryString {
    private String _query;

    public QueryString(String queryString) {
        if (queryString == null) throw new NullPointerException();
        _query = queryString;
    }

    public int count() {
        if ("".equals(_query)) return 0;
        String[] pairs = _query.split("&");
        return pairs.length;
    }

    public String valueFor(String name) {
        String[] pairs = _query.split("&");
```

```
            for (String pair : pairs) {
                String[] nameAndValue = pair.split("=");
                if (nameAndValue[0].equals(name)) return nameAndValue[1];
            }
            throw new RuntimeException(name + " not found");
        }
    }
```

To eliminate the duplication, I needed a single method that could do all the parsing. The other methods would work with the results of the parse rather than doing any parsing of their own. I decided that this parser, called from the constructor, would parse the data into a `HashMap`.

Although I could have done this in one giant step by moving the body of `valueFor()` into a `parseQueryString()` method and then hacking until the tests passed again, I knew from hard-won experience that it was faster to proceed in small steps.

My first step was to introduce `HashMap()` into `valueFor()`. This would make `valueFor()` look just like the `parseQueryString()` method I needed. Once it did, I could extract out `parseQueryString()` easily.

```
    public String valueFor(String name) {
        HashMap<String, String> map = new HashMap<String, String>();

        String[] pairs = _query.split("&");
        for (String pair : pairs) {
            String[] nameAndValue = pair.split("=");
            map.put(nameAndValue[0], nameAndValue[1]);
        }
        return map.get(name);
    }
```

After making this refactoring, I ran the tests. They passed.

> **NOTE**
>
> **By removing the `RuntimeException`, I had changed the behavior of the code when the name was not found. That was OK because the `RuntimeException` was just an assertion about an unimplemented feature. I hadn't yet written any tests around it. If I had, I would have changed the code to maintain the existing behavior.**

Now I could extract the parsing logic into its own method. I used my IDE's built-in *Extract Method* refactoring to do so.

```
    public String valueFor(String name) {
        HashMap<String, String> map = parseQueryString();
        return map.get(name);
    }

    private HashMap<String, String> parseQueryString() {
        HashMap<String, String> map = new HashMap<String, String>();

        String[] pairs = _query.split("&");
        for (String pair : pairs) {
            String[] nameAndValue = pair.split("=");
            map.put(nameAndValue[0], nameAndValue[1]);
        }
        return map;
    }
```

The tests passed again, of course. With such small steps, I'd be surprised if they didn't. That's the point: by taking small steps, I remain in complete control of my changes, which reduces surprises.

I now had a `parseQueryString()` method, but it was only available to `valueFor()`. My next step was to make it available to all methods. To do so, I created a `_map` instance variable and had parseQueryString() use it.

```
public class QueryString {
    private String _query;
    private HashMap<String, String> _map = new HashMap<String, String>();

...

    public String valueFor(String name) {
        HashMap<String, String> map = parseQueryString();
        return map.get(name);
    }

    private HashMap<String, String> parseQueryString() {
        String[] pairs = _query.split("&");
        for (String pair : pairs) {
            String[] nameAndValue = pair.split("=");
            _map.put(nameAndValue[0], nameAndValue[1]);
        }
        return _map;
    }
}
```

This is a trickier refactoring than it seems. Whenever you switch from a local variable to an instance variable, the order of operations can get confused. That's why I continued to have `parseQueryString()` return `_map`, even though it was now available as an instance variable. I wanted to make sure this first step passed its tests before proceeding to my next step, which was to get rid of the unnecessary `return`.

```
public class QueryString {
    private String _query;
    private HashMap<String, String> _map = new HashMap<String, String>();

...

    public String valueFor(String name) {
        parseQueryString();
        return _map.get(name);
    }

    private void parseQueryString() {
        String[] pairs = _query.split("&");
        for (String pair : pairs) {
            String[] nameAndValue = pair.split("=");
            _map.put(nameAndValue[0], nameAndValue[1]);
        }
    }
}
```

The tests passed again.

Because `parseQueryString()` now stood entirely on its own, its only relationship to `valueFor()` was that it had to be called before `valueFor()`'s `return` statement. I was finally ready to achieve my goal of calling `parseQueryString()` from the constructor.

```
public class QueryString {
    private String _query;
    private HashMap<String, String> _map = new HashMap<String, String>();

    public QueryString(String queryString) {
        if (queryString == null) throw new NullPointerException();
        _query = queryString;
        parseQueryString();
    }

...

    public String valueFor(String name) {
        return _map.get(name);
    }

...

}
```

This seemed like a simple refactoring. After all, I moved only *one line* of code. Yet when I ran my tests, they failed. My parse method didn't work with an empty string—a degenerate case that I hadn't yet implemented in `valueFor()`. It wasn't a problem as long as only `valueFor()` ever called `parseQueryString()`, but it showed up now that I called `parseQueryString()` in the constructor.

> NOTE
> **This shows why taking small steps is such a good idea. Because I had only changed one line of code, I knew exactly what had gone wrong.**

The problem was easy to fix with a guard clause.

```
private void parseQueryString() {
    if ("".equals(_query)) return;

    String[] pairs = _query.split("&");
    for (String pair : pairs) {
        String[] nameAndValue = pair.split("=");
        _map.put(nameAndValue[0], nameAndValue[1]);
    }
}
```

At this point, I was nearly done. The duplicate parsing in the `count()` method caused all of this mess, and I was ready to refactor it to use the `_map` variable rather than do its own parsing. It went from:

```
public int count() {
    if ("".equals(_query)) return 0;
    String[] pairs = _query.split("&");
    return pairs.length;
}
```

to:

```
public int count() {
    return _map.size();
}
```

I love it when I can delete code.

I reviewed the code and saw just one loose end remaining: the _query instance variable that stored the unparsed query string. I no longer needed it anywhere but parseQueryString(), so I demoted it from an instance variable to a parseQueryString() parameter.

```
public class QueryString {
    private HashMap<String, String> _map = new HashMap<String, String>();

    public QueryString(String queryString) {
        if (queryString == null) throw new NullPointerException();
        parseQueryString(queryString);
    }

    public int count() {
        return _map.size();
    }

    public String valueFor(String name) {
        return _map.get(name);
    }

    private void parseQueryString(String query) {
        if ("".equals(query)) return;

        String[] pairs = query.split("&");
        for (String pair : pairs) {
            String[] nameAndValue = pair.split("=");
            _map.put(nameAndValue[0], nameAndValue[1]);
        }
    }
}
```

When you compare the initial code to this code, there's little in common. Yet this change took place as a series of small, careful refactorings. Although it took me a long time to describe the steps, each individual refactoring took a matter of seconds in practice. The whole series occurred in a few minutes.

Questions

How often should we refactor?

Constantly. Perform little refactorings as you use TDD and bigger refactorings every week. Every week, your design should be better than it was the week before. (See "Incremental Design and Architecture"" later in this chapter.)

Ally

Incremental Design and Architecture (p. 323)

Isn't refactoring rework? Shouldn't we design our code correctly from the beginning?

If it were possible to design your code perfectly from the beginning, then refactoring would be rework. However, as anybody who's worked with large systems knows, mistakes always creep in. It isn't possible to design software perfectly. That's why refactoring is important. Rather than bemoan the errors in the design, celebrate your ability to fix them.

What about our database? That's what really needs improvement.

You can refactor databases, too. Just as with normal refactorings, the trick is to proceed in small, behavior-preserving steps. For example, to rename a table, you can create a new table, copy the data from one to the next, update constraints, update stored procedures and

applications, then delete the old table.* See "Further Reading"" at the end of this section for more.

How can we make large design changes without conflicting with other team members?

Take advantage of communication and continuous integration. Before taking on a refactoring that will touch a bunch of code, check in your existing code and let people know what you're about to do. Sometimes other pairs can reduce the chance of integration conflicts by mirroring any renaming you're planning to do. (IDEs with refactoring support make such renaming painless.)

Ally

Continuous Integration (p. 183)

During the refactoring, I like to use the distributed version control system SVK, built atop Subversion. It allows me to commit my changes to a local repository one at a time, then push all of them to the main repository when I reach the point of integration. This doesn't prevent conflicts with other pairs, but it allows me to checkpoint locally, which reduces my need to disturb anyone else before I finish.

My refactoring broke the tests! They passed, but then we changed some code and now they fail. What happened?

It's possible you made a mistake in refactoring, but if not, it could be a sign of poor tests. They might test the code's *implementation* rather than its *behavior*. Undo the refactoring and take a look at the tests; you may need to refactor them.

Is refactoring tests actually worthwhile?

Absolutely! Too many developers forget to do this and find themselves maintaining tests that are brittle and difficult to change. Tests have to be maintained just as much as production code does, so they're valid targets for refactoring, too.

Exercise caution when refactoring tests. It's easier to unwittingly break a test than it is to break production code because you can make the test pass when it shouldn't. I like to temporarily change my production code to make the tests fail just to show that they still can. Pair programming also helps.

Ally

Pair Programming (p. 74)

Sometimes it's valuable to leave more duplication in your tests than you would in the code itself. Tests have a documentation value, and reducing duplication and increasing abstraction can sometimes obscure the intent of the tests. This can be a tough judgment call—err on the side of eliminating duplication.

Results

When you use refactoring as an everyday part of your toolkit, the code constantly improves. You make significant design changes safely and confidently. Every week, the code is at least slightly better than it was the week before.

* These steps assume that the database isn't live during the refactoring. A live refactoring would have a few more steps.

Contraindications

Refactoring requires good tests. Without it, it's dangerous because you can't easily tell whether your changes have modified behavior. When I have to deal with untested legacy code, I often write a few end-to-end tests first to provide a safety net for refactoring.

Ally

Test-Driven Development (p. 287)

Refactoring also requires collective code ownership. Any significant design changes will require that you change all parts of the code. Collective code ownership makes this possible. Similarly, refactoring requires continuous integration. Without it, each integration will be a nightmare of conflicting changes.

Allies

Collective Code Ownership (p. 191)

Continuous Integration (p. 183)

It's possible to spend too much time refactoring. You don't need to refactor code that's unrelated to your present work. Similarly, balance your need to deliver stories with the need to have good code. As long as the code is better than it was when you started, you're doing enough. In particular, if you think the code could be better, but you're not sure how to improve it, it's OK to leave it for someone else to improve later.

Alternatives

There is no real alternative to refactoring. No matter how carefully you design, all code accumulates technical debt. Without refactoring, that technical debt will eventually overwhelm you, leaving you to choose between rewriting the software (at great expense and risk) or abandoning it entirely.

Further Reading

"Clean Code: Args—A Command-line Argument Parser" [Martin 2005] is a rare treasure: a detailed walk-through of an extended refactoring. If you liked my refactoring example but want more, read this article. It's online at *http://www.objectmentor.com/resources/articles/Clean_Code_Args.pdf*.

Refactoring: Improving the Design of Existing Code [Fowler 1999] is the definitive reference for refactoring. It's also a great read. Buy it.

Refactoring to Patterns [Kerievsky] takes Fowler's work one step further, showing how refactorings can string together to achieve significant design changes. It's a good way to learn more about how to use individual refactorings to achieve big results.

Refactoring Databases: Evolutionary Database Design [Ambler & Sadalage] shows how refactoring can apply to database schemas.

Simple Design

Audience

Programmers

Our design is easy to modify and maintain.

> Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away. —Antoine de Saint-Exupéry, author of *The Little Prince*

> Any intelligent fool can make things bigger, more complex and more violent. It takes a touch of genius and a lot of courage to move in the opposite direction. —Albert Einstein

When writing code, agile developers often stop to ask themselves, "What is the simplest thing that could possibly work?" They seem to be obsessed with simplicity. Rather than anticipating changes and providing extensibility hooks and plug-in points, they create a simple design that anticipates as little as possible, as cleanly as possible. Unintuitively, this results in designs that are ready for *any* change, anticipated or not.

This may seem silly. How can a design be ready for any change? Isn't the job of a good designer or architect to anticipate future changes and make sure the design can be extended to support them? Doesn't *Design Patterns: Elements of Reusable Software* say that the key to maximizing reuse is to anticipate changes and design accordingly?

I'll let Erich Gamma, coauthor of *Design Patterns*, answer these questions. In the following excerpt, Gamma is interviewed by Bill Venners.*

> *Venners:* The GoF book [*Design Patterns*] says, "The key to maximizing reuse lies in anticipating new requirements and changes to existing requirements, and in designing your systems so they can evolve accordingly. To design a system so that it's robust to such changes, you must consider how the system might need to change over its lifetime. A design that doesn't take change into account risks major redesign in the future." That seems contradictory to the XP philosophy.
>
> *Gamma:* It contradicts absolutely with XP. It says you should think ahead. You should speculate. You should speculate about flexibility. Well yes, I matured too and XP reminds us that it is expensive to speculate about flexibility, so I probably wouldn't write this exactly this way anymore. To add flexibility, you really have to be able to justify it by a requirement. If you don't have a requirement up front, then I wouldn't put a hook for flexibility in my system up front.
>
> But I don't think XP and [design] patterns are conflicting. It's how you use patterns. The XP guys have patterns in their toolbox, it's just that they refactor to the patterns once they need the flexibility. Whereas we said in the book ten years ago, no, you can also anticipate. You start your design and you use them there up-front. In your up-front design you use patterns, and the XP guys don't do that.
>
> *Venners:* So what do the XP guys do first, if they don't use patterns? They just write the code?
>
> *Gamma:* They write a test.
>
> *Venners:* Yes, they code up the test. And then when they implement it, they just implement the code to make the test work. Then when they look back, they refactor, and maybe implement a pattern?
>
> *Gamma:* Or when there's a new requirement. I really like flexibility that's requirement driven. That's also what we do in Eclipse. When it comes to exposing more API, we do that on demand. We expose API gradually. When clients tell us, "Oh, I had to use or duplicate all these internal classes. I really don't want to do that,"

* "Erich Gamma on Flexibility and Reuse: A Conversation with Erich Gamma, Part II," *http://www.artima.com/lejava/articles/reuse.html*.

when we see the need, then we say, OK, we'll make the investment of publishing this as an API, make it a commitment. So I really think about it in smaller steps, we do not want to commit to an API before its time.

BECK ON SIMPLICITY

In the first edition of *Extreme Programming Explained*, Kent Beck described simple design as code that passes its tests and meets four guidelines, with the earlier guidelines taking precedence over the later ones:

1. The system (code and tests together) must communicate everything you want to communicate.
2. The system must contain no duplicate code. (Guidelines 1 and 2 together constitute the Once and Only Once rule).
3. The system should have the fewest possible classes.
4. The system should have the fewest possible methods.

In the second edition, he rephrased the advice:

1. *Appropriate for the intended audience.* It doesn't matter how brilliant and elegant a piece of design is; if the people who need to work with it don't understand it, it isn't simple for them.
2. *Communicative.* Every idea that needs to be communicated is represented in the system. Like words in a vocabulary, the elements of the system communicate to future readers.
3. *Factored.* Duplication of logic or structure makes code hard to understand and modify.
4. *Minimal.* Within the above three constraints, the system should have the fewest elements possible. Fewer elements means less to test, document, and communicate.

Simple, not simplistic.

Simple doesn't mean *simplistic*. Don't make boneheaded design decisions in the name of reducing the number of classes and methods. A simple design is clean and elegant, not something you throw together with the least thought possible. Here are some points to keep in mind as you strive for simplicity.

You Aren't Gonna Need It (YAGNI)

This pithy XP saying sums up an important aspect of simple design: avoid speculative coding. Whenever you're tempted to add something to your design, ask yourself if it supports the stories and features you're currently delivering. If not, well... you aren't gonna need it. Your design could change. Your customers' minds could change.

Similarly, remove code that's no longer in use. You'll make the design smaller, simpler, and easier to understand. If you need it again in the future, you can always get it out of version control. For now, it's a maintenance burden you don't need.

Ally

Version Control (p. 169)

We do this because excess code makes change difficult. Speculative design, added to make specific changes easy, often turns out to be wrong in some way, which actually makes changes more difficult. It's usually easier to add to a design than to fix a design that's wrong. The incorrect design has code that depends on it, sometimes locking bad decisions in place.

Once and Only Once

Once and only once is a surprisingly powerful design guideline. As Martin Fowler said:*

> One of the things I've been trying to do is look for simpler [rules] or rules underpinning good or bad design. I think one of the most valuable rules is avoid duplication. "Once and only once" is the Extreme Programming phrase. The authors of *The Pragmatic Programmer* [Hunt & Thomas] use "don't repeat yourself," or the DRY principle.
>
> You can almost do this as an exercise. Look at some program and see if there's some duplication. Then, without really thinking about what it is you're trying to achieve, just pigheadedly try to remove that duplication. Time and time again, I've found that by simply removing duplication I accidentally stumble onto a really nice elegant pattern. It's quite remarkable how often that is the case. I often find that a nice design can come from just being really anal about getting rid of duplicated code.

There's even more to this idea than removing duplication. Think of it this way:

Express every concept once. (And only once). †

In other words, don't just eliminate duplication; make sure that every important concept has an explicit representation in your design. As [Hunt & Thomas] phrase their DRY Principle: "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system."

An effective way to make your code express itself once (and only once) is to be explicit about core concepts. Rather than expressing these concepts with a primitive data type, create a new type. For example, rather than representing dollar amounts with a `decimal` data type, create a `Dollars` class. (See Example 9-2.)

Example 9-2. Simple value type

```
public class Dollars {
    private decimal _dollars;
    public Dollars(decimal dollars) { _dollars = dollars; }
    public decimal AsDecimal() { return _dollars; }
    public boolean Equals(object o) {...}
}
```

Although using basic data types may seem simpler—it's one less class, after all—it actually makes your design more complex. Your code doesn't have a place for the concept. As a result, every time someone works with that concept, the code may need to reimplement basic behavior, such as string parsing and formatting, which results in widespread duplication. This duplication will likely be only fragments of code, but the net weight of those fragments will

* *http://www.artima.com/intv/principlesP.html*.

† Thanks to Andrew Black for this insight.

make your code hard to change. For example, if you decide to change the display of dollar amounts—perhaps you want negative amounts to be red—you must find every little fragment of formatting code and fix it.

Instead, make sure that every concept has a home. Don't generalize; just make sure the basic concept has an explicit representation. Over time, as needed, add code (such as formatting and parsing) to your type. By starting with a simple but explicit representation of the concept, you provide a location for those future changes to congregate. Without it, they will tend to accumulate in other methods and lead to duplication and complex code.

Self-Documenting Code

Simplicity is in the eye of the beholder. It doesn't matter much if *you* think the design is simple; if the rest of your team or future maintainers of your software find it too complicated, then it is.

To avoid this problem, use idioms and patterns that are common for your language and team. It's OK to introduce new ideas, too, but run them past other team members first. Be sure to use names that clearly reflect the intent of your variables, methods, classes, and other entities.

Pair programming will help you create simple code. If you have trouble understanding something your partner wrote, discuss the situation and try to find a better way to express the concept. Before you use a comment to explain something, ask your partner how to make the code express its intent without needing a comment.

Ally
Pair Programming (p. 74)

NOTE

Comments aren't bad, but they *are* a sign that your code is more complex than it needs to be. Try to eliminate the need for comments when you can. You can't just arbitrarily delete comments, of course—first make the code so expressive that the comments no longer add value.

Isolate Third-Party Components

A hidden source of duplication lies in calls to third-party components. When you have these calls spread throughout your code, replacing or augmenting that component becomes difficult. If you discover that the component isn't sufficient for your needs, you could be in trouble.

To prevent this problem, isolate your third-party components behind an interface that you control. Ask yourself, "When I need to upgrade or change this component, how hard will it be?" In object-oriented languages, consider using the *Adapter pattern* [Gamma et al.] rather than instantiating third-party classes directly. For frameworks that require that you extend their classes, create your own base classes that extend the framework classes, rather than extending the classes directly.

NOTE

Isolating third-party components also allows you to extend the features of the component and gives you a convenient interface to write tests against if you need to.

Create your adapter incrementally. Instead of supporting every feature of the component in your adapter, support only what you need today. Write the adapter's interface to match your needs, not the interface of the component. This will make it easier to use and to replace when necessary.

Isolating third-party components reduces duplication at the cost of making your code slightly more complex. Some components, such as Java's J2SE or the .NET framework, are so pervasive that isolating them makes little sense. Make the decision to isolate common components according to the risk that you'll need to replace or augment that component. For example, I would use the Java or .NET `String` class directly, without an adapter, but I might consider isolating .NET's cryptography libraries or elements of the J2EE framework.

Limit Published Interfaces

Published interfaces reduce your ability to make changes. Once the public interface to a class or other module is published so that people outside the team may use it, changing it requires great expense and effort. Because other people might be using the interface, changing it can sabotage their efforts.

Some teams approach design as if every public interface were also a published interface. This *internal publication* assumes that, once defined, a public interface should never change. This is a bad idea—it prevents you from improving your design over time. A better approach is to change your nonpublished interfaces whenever you need, updating callers accordingly.

If your code is used outside your team, then you do need published interfaces. Each one, however, is a commitment to a design decision that you may wish to change in the future. Minimize the number of interfaces you expose to the outside world, and ask if the benefit of having other teams use your code is really worth the cost.* (Sometimes it is, but don't automatically assume so.) Postpone publishing interfaces as long as possible to allow your design to improve and settle.

In some cases, as with teams creating a library for third-party use, the entire purpose of the project is to create a published interface. In that case, your API is your product. Still, the smaller the interface, the better—it's much easier to add new elements to your API than to remove or change incorrect elements. Make the interface as small as is practical.

As Erich Gamma said in the interview, "When it comes to exposing more API [in Eclipse, the open source Java IDE], we do that on demand. We expose API gradually... when we see the need, then we say, OK, we'll make the investment of publishing this as an API, make it a commitment. So I really think about it in smaller steps, we do not want to commit to an API before its time."

> NOTE
> **When developing a library, you can develop your interface incrementally and then freeze it only when you release. Still, it's probably a good idea to think ahead to future changes and consider whether anything about the API you're about to publish will make those changes difficult.**

* [Brooks] estimated that making code externally reusable increases costs threefold. That estimate probably doesn't apply to modern development, but there's still a nontrivial cost associated with creating reusable components. "Object-oriented" doesn't mean "automatic reuse," despite early claims to the contrary.

Fail Fast

One of the pitfalls of simple design is that your design will be incomplete. Some elements won't work because no one has needed them before.

To prevent these gaps from being a problem, write your code to *fail fast*. Use assertions to signal the limits of your design; if someone tries to use something that isn't implemented, the assertion will cause his tests to fail.

> NOTE
> **Sometimes you can have more expressive assertions by writing your own assertion facility, rather than using your language's built-in facility. I like to create a class called `Assert` (`Assume` and `Require` are good synonyms if `Assert` is unavailable) and implement class (static) methods such as `notNull(object)`, `unreachableCode()` and `impossibleException(exception)`.**

Questions

What if we know we're going to need a feature? Shouldn't we put in a design hook for it?

In XP, the plan can change every week. Unless you're implementing the feature that very week, don't put the hook in. The plan could change, leaving you stuck with unneeded code.

Plus, if you're using incremental design and architecture properly, your code will actually be easier to modify in the future than it is today. Saving the change for later will save time and money.

Ally

Incremental Design and Architecture (p. 323)

What if ignoring a feature will make it harder to implement in the future?

A simple design should make arbitrary changes possible by reducing duplication and limiting the scope of changes. If ignoring a potential feature could make it more difficult, you should look for ways to eliminate that risk without explicitly coding support for the feature. "Incremental Design and Architecture"," later in this chapter, has more about risk-driven architecture.

Results

When you create simple designs, you avoid adding support for any features other than the ones you're working on in the current iteration. You finish work more quickly as a result. When you use simple design well, your design supports arbitrary changes easily. Although new features might require a lot of new code, changes to existing code are localized and straightforward.

Contraindications

Simple design requires continuous improvement through refactoring and incremental design and architecture. Without these, your design will fail to evolve with your requirements.

Allies

Refactoring (p. 306)
Incremental Design and Architecture (p. 323)

Don't use simple design as an excuse for poor design. Simplicity requires careful thought. As the Einstein quote at the beginning of this section says, it's a lot easier to create complex designs than simple ones. Don't pretend "simple" means "fastest" or "easiest."

Pair programming and collective code ownership, though not strictly necessary for simple design, will help your team devote the brainpower needed to create truly simple designs.

Allies

Pair Programming (p. 74)
Collective Code Ownership (p. 191)

Alternatives

Until recently, the accepted best practice in design followed the advice Erich Gamma now disavows: "The key to maximizing reuse lies in anticipating new requirements and changes to existing requirements, and in designing your systems so they can evolve accordingly."

A team can have success with this approach, but it depends on how well they anticipate new requirements. If the team's expectations are too far off, they might need to rewrite a lot of code that was based on bad assumptions. Some changes may affect so much code that they're considered impossible. If you follow this approach, it's best to hire designers who have a lot of experience in your specific industry. They're more likely to correctly anticipate changes.

Further Reading

Martin Fowler has a collection of his excellent IEEE Design columns online at *http://www.martinfowler.com/articles.html#IDAOPDBC*. Many of these columns discuss core concepts that help in creating a simple design.

The Pragmatic Programmer: From Journeyman to Master [Hunt & Thomas] contains a wealth of design information that will help you create simple, flexible designs. *Practices of an Agile Developer* [Subramaniam & Hunt] is its spiritual successor, offering similarly pithy advice, though with less emphasis on design and coding.

Prefactoring [Pugh] also has good advice for creating simple, flexible designs.

"Fail Fast" [Shore 2004b] discusses that concept in more detail. It is available at *http://www.martinfowler.com/ieeeSoftware/failFast.pdf*.

Incremental Design and Architecture

Audience

Programmers

We deliver stories every week without compromising design quality.

XP makes challenging demands of its programmers: every week, programmers should finish 4 to 10 customer-valued stories. Every week, customers may revise the current plan and introduce entirely new stories—with no advance notice. This regimen starts with the first week of the project.

Allies

Iterations (p. 42)
Stories (p. 255)

In other words, as a programmer you must be able to produce customer value, from scratch, in a single week. No advance preparation is possible. You can't set aside several weeks for building a domain model or persistence framework; your customers need you to deliver completed stories.

Fortunately, XP provides a solution for this dilemma: *incremental design* (also called *evolutionary design*) allows you to build technical infrastructure (such as domain models and persistence frameworks) incrementally, in small pieces, as you deliver stories.

How It Works

Incremental design applies the concepts introduced in test-driven development to all levels of design. Like test-driven development, programmers work in small steps, proving each before moving to the next. This takes place in three parts: start by creating the simplest design that could possibly work, incrementally add to it as the needs of the software evolve, and continuously improve the design by reflecting on its strengths and weaknesses.

Ally

Test-Driven Development (p. 287)

To be specific, when you *first* create a design element—whether it's a new method, a new class, or a new architecture—be completely specific. Create a simple design that solves only the problem you face at the moment, no matter how easy it may seem to solve more general problems.

Ally

Simple Design (p. 316)

This is difficult! Experienced programmers think in abstractions. In fact, the ability to think in abstractions is often a sign of a good programmer. Coding for one specific scenario will seem strange, even unprofessional.

Do it anyway. The abstractions will come. Waiting to make them will enable you to create designs that are simpler and more powerful.

> Waiting to create abstractions will enable you to create designs that are simple and powerful.

The *second* time you work with a design element, modify the design to make it more general—but only general enough to solve the two problems it needs to solve. Next, review the design and make improvements. Simplify and clarify the code.

Ally

Refactoring (p. 306)

The *third* time you work with a design element, generalize it further—but again, just enough to solve the three problems at hand. A small tweak to the design is usually enough. It will be pretty general at this point. Again, review the design, simplify, and clarify.

Continue this pattern. By the *fourth* or *fifth* time you work with a design element—be it a method, a class, or something bigger—you'll typically find that its abstraction is perfect for your needs. Best of all, because you allowed practical needs to drive your design, it will be simple yet powerful.

NOTE

You can see this process in action in test-driven development. TDD is an example of incremental design at the level of methods and individual classes. Incremental design goes further than TDD, however, scaling to classes, packages, and even application architecture.

INCONCEIVABLE!

I have to admit I was very skeptical of incremental design when I first heard about it. I felt that up-front design was the only responsible approach. The first time my team tried incremental design, we mixed up-front design with incremental design. We designed the architecture up-front, feeling it was too important to leave until later. Over time, however, experience showed us that many of those initial design decisions had serious flaws. We used incremental design not only to fix the flaws, but to produce far better alternatives. When I left the project eighteen months later, it had the best design of any code I've ever seen.

That project taught me to trust incremental design. It's different from traditional design approaches, but it's also strikingly effective—and more forgiving of mistakes. Software projects usually succumb to *bit rot* and get more difficult to modify over time. With incremental design, the reverse tends to be true: software actually gets *easier* to modify over time. Incremental design is so effective, it's now my preferred design approach for any project, XP or otherwise.

Continuous Design

Incremental design initially creates every design element—method, class, namespace, or even architecture—to solve a specific problem. Additional customer requests guide the incremental evolution of the design. This requires continuous attention to the design, albeit at different timescales. Methods evolve in minutes; architectures evolve over months.

No matter what level of design you're looking at, the design tends to improve in bursts. Typically, you'll implement code into the existing design for several cycles, making minor changes as you go. Then something will give you an idea for a new design approach, which will require a series of refactorings to support it. [Evans] calls this a b*reakthrough* (see Figure 9-2). Breakthroughs happen at all levels of the design, from methods to architectures.

Breakthroughs are the result of important insights and lead to substantial improvements to the design. (If they don't, they're not worth implementing.) You can see a small, method-scale breakthrough at the end of "A TDD Example"" earlier in this chapter.

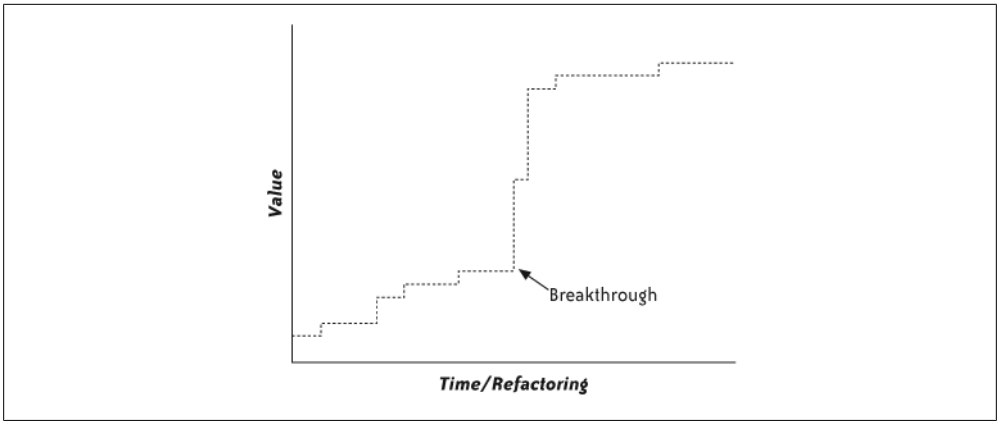


Figure 9-2. Breakthrough

Incrementally Designing Methods

You've seen this level of incremental design before: it's test-driven development. While the driver implements, the navigator thinks about the design. She looks for overly complex code and missing elements, which she writes on her notecard. She thinks about which features the code should support next, what design changes might be necessary, and which tests may guide the code in the proper direction. During the refactoring step of TDD, both members of the pair look at the code, discuss opportunities for improvements, and review the navigator's notes.

Allies

Test-Driven Development (p. 287)
Pair Programming (p. 74)

> NOTE
> **The roles of driver and navigator aren't as cut-and-dried as I imply. It's OK for drivers to think about design and for navigators to make implementation suggestions.**

Method refactorings happen every few minutes. Breakthroughs may happen several times per hour and can take 10 minutes or more to complete.

Incrementally Designing Classes

When TDD is performed well, the design of individual classes and methods is beautiful: they're simple, elegant, and easy to use. This isn't enough. Without attention to the interaction between classes, the overall system design will be muddy and confusing.

> Nothing clarifies a design issue like working code.

During TDD, the navigator should also consider the wider scope. Ask yourself these questions: are there similarities between the code you're implementing and other code in the system? Are class responsibilities clearly defined and concepts clearly represented? How well does this class interact with other classes?

When you see a problem, jot it down on your card. During one of the refactoring steps of TDD—usually, when you're not in the middle of something else—bring up the issue, discuss solutions with your partner, and refactor. If you think your design change will significantly affect other members of the team, take a quick break to discuss it around a whiteboard.

> NOTE
> **Don't let design discussions turn into long, drawn-out disagreements. Follow *the 10-minute rule*: if you disagree on a design direction for 10 minutes, try one and see how it works in practice. If you have a particularly strong disagreement, split up and try both as spike solutions. Nothing clarifies a design issue like working code.**

Class-level refactorings happen several times per day. Depending on your design, breakthroughs may happen a few times per week and can take several hours to complete. (Nonetheless, remember to proceed in small, test-verified steps.) Use your iteration slack to complete breakthrough refactorings. In some cases, you won't have time to finish all the refactorings you identify. That's OK; as long as the design is better at the end of the week than it was at the beginning, you're doing enough.

Ally
Slack (p. 247)

> NOTE
> **Avoid creating `TODO` comments or story/task cards for postponed refactorings. If the problem is common enough for you or others to notice it again, it will get fixed eventually. If not, then it probably isn't worth fixing. There are always more opportunities to refactor than time to do it all; `TODO`s or refactoring cards add undue stress to the team without adding much value.**

Incrementally Designing Architecture

Large programs use overarching organizational structures called *architecture*. For example, many programs segregate user interface classes, business logic classes, and persistence classes into their own namespaces; this is a classic *three-layer architecture*. Other designs have the application pass the flow of control from one machine to the next in an *n-tier architecture*.

These architectures are implemented through the use of recurring patterns. They aren't design patterns in the formal Gang of Four* sense. Instead, they're standard conventions specific to your codebase. For example, in a three-layer architecture, every business logic class will probably be part of a "business logic" namespace, may inherit from a generic "business object" base class, and probably interfaces with its persistence layer counterpart in a standard way.

These recurring patterns embody your application architecture. Although they lead to consistent code, they're also a form of duplication, which makes changes to your architecture more difficult.

Fortunately, you can also design architectures incrementally. As with other types of continuous design, use TDD and pair programming as your primary vehicle. While your software grows,

* The "Gang of Four" is a common nickname for the authors of *Design Patterns*, a book that introduced design patterns to the mainstream.

be conservative in introducing new architectural patterns: introduce just what you need to for the amount of code you have and the features you support at the moment. Before introducing a new pattern, ask yourself if the duplication is really necessary. Perhaps there's some language feature you can use that will reduce your need to rely on the pattern.

In my experience, breakthroughs in architecture happen every few months. (This estimate will vary widely depending on your team members and code quality.) Refactoring to support the breakthrough can take several weeks or longer because of the amount of duplication involved. Although changes to your architecture may be tedious, they usually aren't difficult once you've identified the new architectural pattern. Start by trying out the new pattern in just one part of your design. Let it sit for a while—a week or two—to make sure the change works well in practice. Once you're sure it does, bring the rest of the system into compliance with the new structure. Refactor each class you touch as you perform your everyday work, and use some of your slack in each iteration to fix other classes.

Ally
Slack (p. 247)

Keep delivering stories while you refactor. Although you could take a break from new development to refactor, that would disenfranchise your customers. Balance technical excellence with delivering value. Neither can take precedence over the other. This may lead to inconsistencies within the code during the changeover, but fortunately, that's mostly an aesthetic problem—more annoying than problematic.

> Balance technical excellence with delivering value.

NOTE

Introducing architectural patterns incrementally helps reduce the need for multiiteration refactorings. It's easier to expand an architecture than it is to simplify one that's too ambitious.

Risk-Driven Architecture

Architecture may seem too essential not to design up-front. Some problems do seem too expensive to solve incrementally, but I've found that nearly everything is easy to change if you eliminate duplication and embrace simplicity. Common thought is that distributed processing, persistence, internationalization, security, and transaction structure are so complex that you *must* consider them from the start of your project. I disagree; I've dealt with all of them incrementally [Shore 2004a].

NOTE

Two issues that remain difficult to change are choice of programming language and platform. I wouldn't want to make those decisions incrementally!

Of course, no design is perfect. Even with simple design, some of your code will contain duplication, and some will be too complex. There's always more refactoring to do than time to do it. That's where *risk-driven architecture* comes in.

Ally
Simple Design (p. 316)

Although I've emphasized designing for the present, it's OK to *think* about future problems. Just don't *implement* any solutions to stories that you haven't yet scheduled.

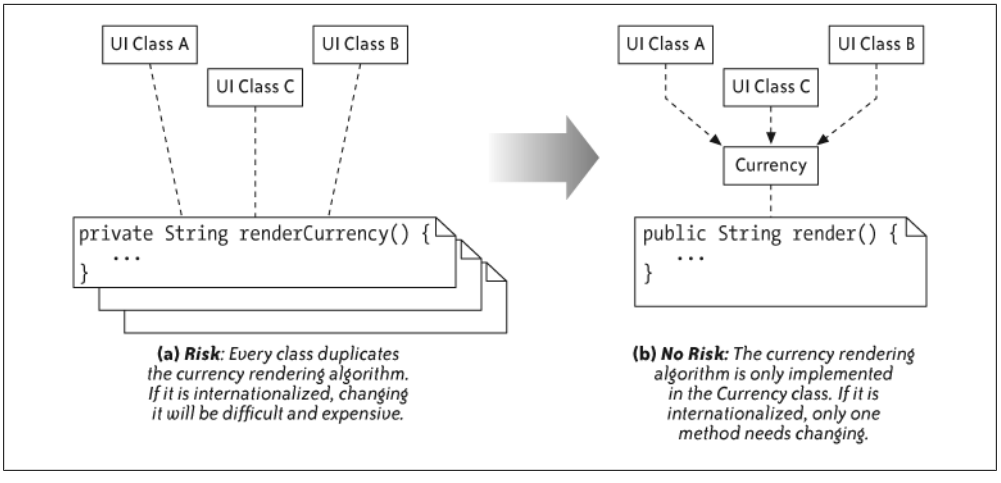


Figure 9-3. Use risk to drive refactoring

What do you do when you see a hard problem coming? For example, what if you know that internationalizing your code is expensive and only going to get more expensive? Your power lies in your ability to choose which refactorings to work on. Although it would be inappropriate to implement features your customers haven't asked for, you *can* direct your refactoring efforts toward reducing risk. Anything that improves the current design is OK—so choose improvements that also reduce future risk.

To apply risk-driven architecture, consider what it is about your design that concerns you and eliminate duplication around those concepts. For example, if your internationalization concern is that you always format numbers, dates, and other variables in the local style, look for ways to reduce duplication in your variable formatting. One way to do so is to make sure every concept has its own class (as described in "Once and Only Once"" earlier in this chapter), then condense all formatting around each concept into a single method within each class, as shown in Figure 9-3. If there's still a lot of duplication, the Strategy pattern would allow you to condense the formatting code even further.

Limit your efforts to improving your *existing* design. Don't actually implement support for localized formats until your customers ask for them. Once you've eliminated duplication around a concept—for example, once there's only one method in your entire system that renders numbers as strings—changing its implementation will be just as easy later as it is now.

> NOTE
>
> **A team I worked with replaced an entire database connection pooling library in half a pair-day. Although we didn't anticipate this need, it was still easy because we had previously eliminated all duplication around database connection management. There was just one method in the entire system that created, opened, and closed connections, which made writing our own connection pool manager almost trivially easy.***

* We did have to make it thread-safe, so it wasn't entirely trivial.

USING STORIES TO REDUCE RISK

Another great way to reduce technical risk is to ask your customers to schedule stories that will allow you to work on the risky area. For example, to address the number localization risk, you could create a story such as "Localize application for Spain" (or any European country). This story expresses customer value, yet addresses an internationalization risk.

Your customers have final say over story priorities, however, and their sense of risk and value may not match yours. Don't feel too bad if this happens; you can still use refactorings to reduce your risk.

It's Not Just Coding

Although incremental design focuses heavily on test-driven development and refactoring as an enabler, it isn't about coding. When you use TDD, incremental design, and pair programming well, every pairing session involves a lot of conversation about design. In fact, that's what *all* the (relevant) conversations are about. As Ron Jeffries likes to say, design is so important in XP that we do it *all the time*. Some of the design discussions are very detailed and nitpicky, such as, "What should we name this method?" Others are much higher-level, such as, "These two classes share some responsibilities. We should split them apart and make a third class."

Allies

Test-Driven Development (p. 287)
Refactoring (p. 306)
Pair Programming (p. 74)

Have design discussions away from the keyboard as often as you think is necessary, and use whatever modelling techniques you find helpful. Try to keep them informal and collaborative; sketches on a whiteboard work well. Some people like to use *CRC* (Class, Responsibility, Collaborator) cards.

Some of your discussions will be *predictive*, meaning you'll discuss how you can change your design to support some feature that you haven't yet added to the code. Others will be *reflective*, meaning you'll discuss how to change your design to better support existing features.

> NOTE
> **Beware of getting trapped in *analysis paralysis* and spending too much time trying to figure out a design. If a design direction isn't clear after 10 minutes or so, you probably need more information. Continue using TDD and making those refactorings that are obvious, and a solution will eventually become clear.**

Reflective design (discussed in more detail in "Refactoring"" earlier in this chapter) is always helpful in XP. I like to sketch UML diagrams on a whiteboard to illustrate problems in the current design and possible solutions. When my teams discover a breakthrough refactoring at the class or architecture level, we gather around a whiteboard to discuss it and sketch our options.

Predictive design is less helpful in XP, but it's still a useful tool. As you're adding a new feature, use it to help you decide which tests to write next in TDD. On a larger scale, use predictive design to consider risks and perform risk-driven architecture.

Ally

Test-Driven Development (p. 287)

The trick to using predictive design in XP is keeping your design simple and focusing only on the features it currently supports. Although it's OK to predict how your design will change when you add new features, you shouldn't actually implement those changes until you're working on the stories in question. When you do, you should keep your mind open to other ways of implementing those features. Sometimes the act of coding with test-driven development will reveal possibilities you hadn't considered.

Given these caveats, I find that I use predictive design less and less as I become more experienced with incremental design. That's not because it's "against the rules"—I'm perfectly happy breaking rules—but because working incrementally and reflectively has genuinely yielded better results for me.

Try it yourself, and find the balance between predictive and reflective design that works best for you.

Questions

Isn't incremental design more expensive than up-front design?

Just the opposite, actually, in my experience. There are two reasons for this. First, because incremental design implements just enough code to support the current requirements, you start delivering features much more quickly with incremental design. Second, when a predicted requirement changes, you haven't coded any parts of the design to support it, so you haven't wasted any effort.

Even if requirements never changed, incremental design would still be more effective, as it leads to design breakthroughs on a regular basis. Each breakthrough allows you to see new possibilities and eventually leads to another breakthrough—sort of like walking through a hilly forest in which the top of each hill reveals a new, higher hill you couldn't see before. This continual series of breakthroughs substantially improves your design.

What if we get the design absolutely wrong and have to backtrack to add a new feature?

Sometimes a breakthrough will lead you to see a completely new way of approaching your design. In this case, refactoring may seem like backtracking. This happens to everyone and is not a bad thing. The nature of breakthroughs—especially at the class and architectural level—is that you usually don't see them until after you've lived with the current design for a while.

Our organization (or customer) requires comprehensive design documentation. How can we satisfy this requirement?

Ask them to schedule it with a story, then estimate and deliver it as you would any other story. Remind them that the design will change over time. The most effective option is to schedule documentation stories for the last iteration.

Ally

Documentation (p. 195)

If your organization requires up-front design documentation, the only way to provide it is to engage in up-front design. Try to keep your design efforts small and simple. If you can, use incremental design once you actually start coding.

Results

When you use incremental design well, every iteration advances the software's features and design in equal measure. You have no need to skip coding for an iteration for refactoring or design. Every week, the quality of the software is better than it was the week before. As time goes on, the software becomes increasingly easy to maintain and extend.

Contraindications

Incremental design requires simple design and constant improvement. Don't try to use incremental design without a commitment to continuous daily improvement (in XP terms, *merciless refactoring*). This requires self-discipline and a strong desire for high-quality code from at least one team member. Because nobody can do that all the time, pair programming, collective code ownership, energized work, and slack are important support mechanisms.

Allies

Refactoring (p. 306)
Simple Design (p. 316)
Pair Programming (p. 74)
Collective Code Ownership (p. 191)
Energized Work (p. 82)
Slack (p. 247)

Test-driven development is also important for incremental design. Its explicit refactoring step, repeated every few minutes, gives pairs continual opportunities to stop and make design improvements. Pair programming helps in this area, too, by making sure that half the team's programmers—as navigators—always have an opportunity to consider design improvements.

Ally

Test-Driven Development (p. 287)

Be sure your team sits together and communicates well if you're using incremental design. Without constant communication about class and architectural refactorings, your design will fragment and diverge. Agree on coding standards so that everyone follows the same patterns.

Allies

Sit Together (p. 113)
Coding Standards (p. 133)

Anything that makes continuous improvement difficult will make incremental design difficult. Published interfaces are an example; because they are difficult to change after publication, incremental design may not be appropriate for published interfaces. (You can still use incremental design for the *implementation* of those interfaces.) Similarly, any language or platform that makes refactoring difficult will also inhibit your use of incremental design.

Ally

Refactoring (p. 306)

Finally, some organizations place organizational rather than technical impediments on refactoring, as with organizations that require up-front design documentation or have rigidly controlled database schemas. Incremental design may not be appropriate in these situations.

Alternatives

If you are uncomfortable with XP's approach to incremental design, you can hedge your bets by combining it with up-front design. Start with an up-front design stage, and then commit completely to XP-style incremental design. Although it will delay the start of your first iteration (and may require some up-front requirements work, too), this approach has the advantage of providing a safety net without incurring too much risk.

NOTE

If you're feeling bold, use XP's iterative design directly, without the safety net. Incremental design is powerful, effective, and inexpensive. The added effort of an up-front design stage isn't necessary.

There are other alternatives to incremental design, but I don't think they would work well with XP. One option is to use another type of incremental design, one more like up-front design, that does some up-front design at the beginning of every iteration, rather than relying on simple design and refactoring to the extent that XP does.

I haven't tried other incremental design approaches with XP because they seem to interact clumsily with XP's short iterations. The design sessions could be too short and small to create a cohesive architecture on their own. Without XP's focus on simple design and merciless refactoring, a single design might not evolve.

Another alternative is to design everything up-front. This could work in an environment with very few requirements changes (or a prescient designer), but it's likely to break down with XP's adaptive plans and tiered planning horizons (see "Release Planning"" in Chapter 8).

Further Reading

"Is Design Dead?" [Fowler 2000], online at *http://www.martinfowler.com/articles/designDead.html*, discusses evolutionary design from a slightly skeptical perspective.

"Continuous Design" [Shore 2004a] discusses my experiences with difficult challenges in incremental design, such as internationalization and security. It is available at *http://www.martinfowler.com/ieeeSoftware/continuousDesign.pdf*.

Spike Solutions

Audience
Programmers

We perform small, isolated experiments when we need more information.

You've probably noticed by now that XP values concrete data over speculation. Whenever you're faced with a question, don't speculate about the answer—conduct an experiment! Figure out how you can use real data to make progress.

That's what spike solutions are for, too.

About Spikes

A *spike solution*, or *spike*, is a technical investigation. It's a small experiment to research the answer to a problem. For example, a programmer might not know whether Java throws an exception on arithmetic overflow. A quick 10-minute spike will answer the question:

```
public class ArithmeticOverflowSpike {
    public static void main(String[] args) {
        try {
            int a = Integer.MAX_VALUE + 1;
            System.out.println("No exception: a = " + a);
        }
```

```
        catch (Throwable e) {
            System.out.println("Exception: " + e);
        }
    }
}
```

```
No exception: a = -2147483648
```

> NOTE
> **Although this example is written as a standalone program, small spikes such as this one can also be written inside your test framework. Although they don't actually call your production code, the test framework provides a convenient way to quickly run the spike and report on the results.**

Performing the Experiment

The best way to implement a spike is usually to create a small program or test that demonstrates the feature in question. You can read as many books and tutorials as you like, but it's my experience that nothing helps me understand a problem more than writing working code. It's important to work from a practical point of view, not just a theoretical one.

Writing code, however, often takes longer than reading a tutorial. Reduce that time by writing small, standalone programs. You can ignore the complexities necessary to write production code—just focus on getting something working. Run from the command line or your test framework. Hardcode values. Ignore user input, unless absolutely necessary. I often end up with programs a few dozen lines long that run almost everything from `main()`.

Of course, this approach means you can't reuse this code in your actual production codebase, as you didn't develop it with all your normal discipline and care. That's fine. It's an *experiment*. When you finish, throw it away, check it in as documentation, or share it with your colleagues, but don't treat it as anything other than an experiment.

> Spike solutions clarify technical issues by setting aside the complexities of production code.

> NOTE
> **I discard the spikes I create to clarify a technical question (such as "Does this language throw an exception on arithmetic overflow?"), but generally keep the ones that demonstrate how to accomplish a specific task (such as "How do I send HTML email?"). I keep a separate *spikes/* directory in my repository just for these sorts of spikes.**

DESIGN SPIKES

Sometimes you'll need to test an approach to your production code. Perhaps you want to see how a design possibility will work in practice, or you need to see how a persistence framework will work on your production code.

In this case, go ahead and work on production code. Be sure to check in your latest changes before you start the spike, and be careful not to check in any of your spike code.

Scheduling Spikes

Most spikes are performed on the spur of the moment. You see a need to clarify a small technical issue, and you write a quick spike to do so. If the spike takes more than a few minutes, your iteration slack absorbs the cost.

Ally

Slack (p. 247)

If you anticipate the need for a spike when estimating a story, include the time in your story estimate. Sometimes you won't be able to estimate a story at all until you've done your research; in this case, create a spike story and estimate that instead (see "Stories"" in Chapter 8).

Questions

Your spike example is terrible! Don't you know that you should never catch `Throwable`?

Exactly. Production code should never catch `Throwable`, but a spike isn't production code. Spike solutions are the one time that you can forget about writing good code and focus just on short-term results. (That said, for larger spikes, you may find code that's *too* sloppy is hard to work with and slows you down.)

Should we pair on spikes?

It's up to you. Because spikes aren't production code, even teams with strict pair programming rules don't require writing spikes in pairs.

One very effective way to pair on a spike is to have one person research the technology while the other person codes. Another option is for both people to work independently on separate approaches, each doing their own research and coding, then coming together to review progress and share ideas.

Should we really throw away the code from our spikes?

Unless you think someone will refer to it later, toss it. Remember, the purpose of a spike solution is to give you the information and experience to know *how* to solve a problem, not to produce the code that solves it.

How often should we do spikes?

Perform a spike whenever you have a question about if or how some piece of technology will work.

What if the spike reveals that the problem is more difficult than we thought?

That's good; it gives you more information. Perhaps the customer will reconsider the value of the feature, or perhaps you need to think of another way to accomplish what you want.

Once, a customer asked me for a feature I thought might work in a certain way, but my spike demonstrated that the relevant Internet standard actually prohibited the desired behavior. We came up with a different design for implementing the larger feature.

Results

When you clarify technical questions with well-directed, isolated experiments, you spend less time speculating about how your program will work. You focus your debugging and exploratory programming on the problem at hand rather than on the ways in which your production code might be interacting with your experiment.

Contraindications

Avoid the temptation to create useful or generic programs out of your spikes. Focus your work on answering a specific technical question, and stop working on the spike as soon as it answers that question. Similarly, there's no need to create a spike when you already understand a technology well.

Don't use spikes as an excuse to avoid disciplined test-driven development and refactoring. Never copy spike code into production code. Even if it is exactly what you need, rewrite it using test-driven development so that it meets your production code standards.

Ally

Test-Driven Development (p. 287)

Alternatives

Spike solutions are a learning technique based on performing small, concrete experiments. Some people perform these experiments in their production code, which can work well for small experiments (such as the arithmetic overflow example), but it increases the scope of possible error. If something doesn't work as expected, is it because your understanding of the technology is wrong? Is it due to an unseen interaction with the production code or test framework? Standalone spikes eliminate this uncertainty.

For stories that you can't estimate accurately, an alternative to scheduling a spike story is to provide a high estimate. This is risky because some stories will take longer than your highest estimate, and some may not be possible at all.

Another option is to research problems by reading about the underlying theory and finding code snippets in books or online. This is often a good way to get started on a spike, but the best way to really understand what's going on is to create your own spike. Simplify and adapt the example. Why does it work? What happens when you change default parameters? Use the spike to clarify your understanding.

Performance Optimization

Audience

Programmers, Testers

We optimize when there's a proven need.

Our organization had a problem.* Every transaction our software processed had a three-second latency. During peak business hours, transactions piled up—and with our recent surge in sales, the lag sometimes became hours. We cringed every time the phone rang; our customers were upset.

* This is a fictionalized account inspired by real experiences.

We knew what the problem was: we had recently changed our order preprocessing code. I remember thinking at the time that we might need to start caching the intermediate results of expensive database queries. I had even asked our customers to schedule a performance story. Other stories had been more important, but now performance was top priority.

I checked out the latest code and built it. All tests passed, as usual. Carlann suggested that we create an end-to-end performance test to demonstrate the problem. We created a test that placed 100 simultaneous orders, then ran it under our profiler.

The numbers confirmed my fears: the average transaction took around 3.2 seconds, with a standard deviation too small to be significant. The program spent nearly all that time within a *single* method: `verify_order_id()`. We started our investigation there.

I was pretty sure a cache was the right answer, but the profiler pointed to another possibility. The method retrieved a list of active order IDs on every invocation, regardless of the validity of the provided ID. Carlann suggested discounting obviously flawed IDs before testing potentially valid ones, so I made the change and we reran the tests. All passed. Unfortunately, that had no effect on the profile. We rolled back the change.

Next, we agreed to implement the cache. Carlann coded a naïve cache. I started to worry about cache coherency, so I added a test to see what happened when a cached order went active. The test failed. Carlann fixed the bug in the cache code, and all tests passed again.

> NOTE
> ***Cache coherency* requires that the data in the cache change when the data in the underlying data store changes and vice versa. It's easy to get wrong.**

Unfortunately, all that effort was a waste. The performance was actually slightly *worse* than before, and the caching code had bloated our previously elegant method into a big mess. We sat in silence for a minute.

What else could it be? Was there a processor bug? Would we have to figure out some way to dump our JIT-compiled code so that we could graph the processor pipelines executing assembly code? Was there a problem with memory pages, or some garbage collector bug? The profiling statistics were clear. As far as we could tell, we had done everything correctly. I reverted all our changes and ran the profiler again: 3.2 seconds per transaction. What were we missing?

Over lunch that day, Carlann and I shared our performance testing woes with the team. "Why don't you let me take a look at it?" offered Nate. "Maybe a fresh pair of eyes will help."

Carlann and I nodded, only too glad to move on to something else. We formed new pairs and worked on nice simple problems for the rest of the day.

The next morning, at the stand-up meeting, Janice told us that she and Nate had found the answer. As it turned out, my initial preconceptions had blinded us to an obvious problem.

There was another method inlined within `verify_order_id()` that didn't show up in the profiler. We didn't look at it because I was sure I understood the code. Janice and Nate, however, stepped through the code. They found a method that was trying to making an unnecessary network connection on each transaction. Fixing it lopped three full seconds off each transaction. They had fixed the problem in less than half an hour.

Oh, and the cache I was sure we would need? We haven't needed it yet.

How to Optimize

Modern computers are complex. Reading a single line of a file from a disk requires the coordination of the CPU, the kernel, a virtual file system, a system bus, the hard drive controller, the hard drive cache, OS buffers, system memory, and scheduling pipelines. Every component exists to solve a problem, and each has certain tricks to squeeze out performance. Is the data in a cache? *Which* cache? How's your memory aligned? Are you reading asynchronously or are you blocking? There are so many variables it's nearly impossible to predict the general performance of any single method.

The days in which a programmer could accurately predict performance by counting instruction clock cycles are long gone, yet some still approach performance with a simplistic, brute-force mindset. They make random guesses about performance based on 20-line test programs, flail around while writing code the first time, leave a twisty mess in the real program, and then take a long lunch.

The days in which a programmer could predict performance by counting instructions are long gone.

Sometimes that even works. More often, it leaves a complex mess that doesn't benefit overall performance. It can actually make your code slower.

A holistic approach is the only accurate way to optimize such complex systems. Measure the performance of the entire system, make an educated guess about what to change, then remeasure. If the performance gets better, keep the change. If it doesn't, discard it. Once your performance test passes, stop—you're done. Look for any missed refactoring opportunities and run your test suite one more time. Then integrate.

Usually, your performance test will be an end-to-end test. Although I avoid end-to-end tests in other situations (because they're slow and fragile—see "Test-Driven Development"" earlier in this chapter), they are often the only accurate way to reproduce real-world performance conditions.

You may be able to use your existing testing tool, such as xUnit, to write your performance tests. Sometimes you get better results from a specialized tool. Either way, encode your performance criteria into the test. Have it return a single, unambiguous pass/fail result as well as performance statistics.

If the test doesn't pass, use the test as input to your profiler. Use the profiler to find the bottlenecks, and focus your efforts on reducing them. Although optimizations often make code more complex, keep your code as clean and simple as possible.

Use a profiler to guide your optimization efforts.

If you're adding new code, such as a cache, use test-driven development to create that code. If you're removing or refactoring code, you may not need any new tests, but be sure to run your test suite after each change.

Ally

Test-Driven Development (p. 287)

When to Optimize

Optimization has two major drawbacks: it often leads to complex, buggy code, and it takes time away from delivering features. Neither is in your customer's interests. Optimize only when it serves a real, measurable need.

> Performance optimizations must serve the customer's needs.

That doesn't mean you should write stupid code. It means your priority should be code that's clean and elegant. Once a story is done, if you're still concerned about performance, run a test. If performance is a problem, fix it—but let your customer make the business decision about how important that fix is.

XP has an excellent mechanism for prioritizing customer needs: the combination of user stories and release planning. In other words, schedule performance optimization just like any other customer-valued work: with a story.

Allies

Stories (p. 255)
Release Planning (p. 207)

Of course, customers aren't always aware of the need for performance stories, especially not ones with highly technical requirements. If you have a concern about potential performance problems in part of the system, explain your concern in terms of business tradeoffs and risks. They still might not agree. That's OK—they're the experts on business value and priorities. It's their responsibility, and their decision.

Similarly, you have a responsibility to maintain an efficient development environment. If your tests start to take too long, go ahead and optimize until you meet a concrete goal, such as five or ten minutes. Keep in mind that the most common cause of a slow build is too much emphasis on end-to-end tests, not slow code.

TESTERS AND PERFORMANCE

When I work with dedicated testers, they often act as technical investigators for the team. One of the ways they do this is to investigate *nonfunctional requirements* (also called *parafunctional requirements*) such as performance.

Testers should help customers articulate their nonfunctional requirements, then create and maintain test suites that evaluate the software's ability to meet those requirements. These test suites can test stability as well as performance and scalability.

How to Write a Performance Story

Like all stories, performance stories need a concrete, customer-valued goal. A typical story will express that goal in one or more of these terms:

Throughput
: How many operations should complete in a given period of time?

Latency
: How much delay is acceptable between starting and completing a single operation?

Responsiveness

How much delay is acceptable between starting an operation and receiving feedback about that operation? What kind of feedback is necessary? (Note that latency and responsiveness are related but different. Although good latency leads to good responsiveness, it's possible to have good responsiveness even with poor latency.)

When writing performance stories, think about *acceptable* performance—the minimum necessary for satisfactory results—and *best possible* performance—the point at which further optimization adds little value.

Why have two performance numbers? Performance optimization can consume an infinite amount of time. Sometimes you reach your "best" goal early; this tells you when to stop. Other times you struggle even to meet the "acceptable" goal; this tells you when to keep going.

For example, a story for a server system could be, "Throughput of at least 100 transactions per minute (1,000 is best). Latency of six seconds per transaction (one second is best)." A client system might have the story, "Show progress bar within 1 second of click (0.1 second is best), and complete search within 10 seconds (1 second is best)."

Also consider the conditions under which your story must perform. What kind of workstation or servers will the software run on? What kind of network bandwidth and latency will be available? What other loads will affect the system? How many people will be using it simultaneously? The answers to these questions are likely the same for all stories. Help your customers determine the answers. If you have a standard deployment platform or a minimum platform recommendation, you can base your answers on this standard.

ESTIMATING PERFORMANCE STORIES

You'll often have difficulty estimating performance stories. As with fixing bugs, the cost of optimization stories depends on how long it takes you to find the cause of the performance problem, and you often won't know how long it will take until you actually find it.

To estimate performance stories, timebox your estimate as you do with bug stories (see "Stories"" in Chapter 8). If you can't solve the problem within the timeboxed estimate, use that information to create a new optimization story with a more accurate estimate, to schedule for a subsequent iteration.

Questions

Why not optimize as we go? We know a section of code will be slow.

How can you really know until you measure it? If your optimization doesn't affect code maintainability or effort—for example, if you have a choice between sorting libraries and you believe one would be faster for your situation—then it's OK to put it in.

That's not usually the case. Like any other work, the choice to optimize is the choice *not* to do something else. It's a mistake to spend time optimizing code instead of adding a feature the customer wants. Further, optimizing code tends to increase complexity, which is in direct conflict with the goal of producing

Ally

Simple Design (p. 316)

a simple design. Although we sometimes need to optimize, we shouldn't reduce maintainability when there's no direct customer value.

If you suspect a performance problem, ask your on-site customers for a ballpark estimate of acceptable performance and run a few tests to see if there's anything to worry about. If there is, talk to your customers about creating and scheduling a story.

How do we write a performance test for situations involving thousands of transactions from many clients?

Good stress-testing tools exist for many network protocols, ranging from ad hoc shell scripts running `telnet` and `netcat` sessions to professional benchmarking applications. Your testers or QA department can recommend specific tools.

Our performance tests take too long to run. How can we maintain a 10-minute build?

Good performance tests often take a long time to run, and they may cause your build to take more time than you like. This is one of the few cases in which a *multistage build* (discussed in "Continuous Integration"" in Chapter 7) is appropriate. Run your performance tests asynchronously, as a separate build from your standard 10-minute build (see "Ten-Minute Build"" in Chapter 7), when you integrate.

Our customers don't want to write performance stories. They say we should write the software to be fast from the beginning. How can we deal with this?

"Fast" is an abstract idea. Do they mean latency should be low? Should throughput be high? Should the application scale better than linearly with increased activity? Is there a point at which performance can plateau, or suffer, or regress? Is UI responsiveness more important than backend processing speed?

These goals need quantification and require programming time to meet. You can include them as part of existing stories, but separating them into their own stories gives your on-site customers more flexibility in scheduling and achieving business value.

Results

When you optimize code as necessary, you invest in activities that customers have identified as valuable over perceived benefit. You quantify necessary performance improvements and can capture that information in executable tests. You measure, test, and gather feedback to lead you to acceptable solutions for reasonable investments of time and effort. Your code is more maintainable, and you favor simple and straightforward code over highly optimized code.

Contraindications

Software is a complex system based on the interrelationship of many interacting parts. It's easy to guess wrong when it comes to performance.

It's easy to guess wrong when it comes to performance.

Therefore, don't optimize code without specific performance criteria and objective proof, such as a performance test, that you're not yet meeting that criteria. Throw away optimizations that don't objectively improve performance.

Be cautious of optimizing without tests; optimization often adds complexity and increases the risk of defects.

Alternatives

There are no good alternatives to measurement-based optimization.

Many programmers attempt to optimize all code as they write it, often basing their optimizations on programmer folklore about what's "fast." Sometimes these beliefs come from trivial programs that execute an algorithm 1,000 times. A common example of this is a program that compares the speed of `StringBuffer` to string concatenation in Java or .NET.

Unfortunately, this approach to optimization focuses on trivial algorithmic tricks. Network and hard drive latency are much bigger bottlenecks than CPU performance in modern computing. In other words, if your program talks over a network or writes data to a hard drive—most database updates do both—it probably doesn't matter how fast your string concatenations are.

On the other hand, some programs *are* CPU-bound. Some database queries are easy to cache and don't substantially affect performance. The only way to be sure about a bottleneck is to measure performance under real-world conditions.

Further Reading

"Yet Another Optimization Article" [Fowler 2002b] also discusses the importance of measurement-based optimization. It's available online at *http://www.martinfowler.com/ieeeSoftware/yetOptimization.pdf*.

Exploratory Testing

Audience

Testers

By Elisabeth Hendrickson

We discover surprises and untested conditions.

Allies

No Bugs (p. 159)
The Whole Team (p. 28)

XP teams have no separate QA department. There's no independent group of people responsible for assessing and ensuring the quality of the final release. Instead, the whole team—customers, programmers, and testers—is responsible for the outcome. On traditional teams, the QA group is often rewarded for finding bugs. On XP teams, there's no incentive program for finding or removing bugs. The goal in XP isn't to find and remove bugs; the goal is *not to write bugs* in the first place. In a well-functioning team, bugs are a rarity—only a handful per month.

Does that mean testers have no place on an XP team? No! Good testers have the ability to look at software from a new perspective, to find surprises, gaps, and holes. It takes time for the team to learn which mistakes to avoid. By providing essential information about what the team overlooks, testers enable the team to improve their work habits and achieve their goal of producing zero bugs.

NOTE

Beware of misinterpreting the testers' role as one of process improvement and enforcement. Don't set up quality gates or monitor adherence to the process. The *whole team* is responsible for process improvement. Testers are respected peers in this process, providing information that allows the whole team to benefit.

One particularly effective way of finding surprises, gaps, and holes is *exploratory testing*: a style of testing in which you learn about the software while simultaneously designing and executing tests, using feedback from the previous test to inform the next. Exploratory testing enables you to discover emergent behavior, unexpected side effects, holes in the implementation (and thus in the automated test coverage), and risks related to quality attributes that cross story boundaries such as security, performance, and reliability. It's the perfect complement to XP's raft of automated testing techniques.

Exploratory testing predates XP. [Kaner] coined the term in the book *Testing Computer Software*, although the practice of exploratory testing certainly preceded the book, probably by decades. Since the book came out, people such as Cem Kaner, James and Jonathan Bach, James Lyndsay, Jonathan Kohl, and Elisabeth Hendrickson have extended the concept of exploratory testing into a discipline.

About Exploratory Testing

Philosophically, exploratory testing is similar to test-driven development and incremental design: rather than designing a huge suite of tests up-front, you design a single test in your head, execute it against the software, and see what happens. The result of each test leads you to design the next, allowing you to pursue directions that you wouldn't have anticipated if you had attempted to design all the tests up-front. Each test is a little experiment that investigates the capabilities and limitations of the emerging software.

Exploratory testing can be done manually or with the assistance of automation. Its defining characteristic is not how we drive the software but rather the tight feedback loop between test design, test execution, and results interpretation.

Exploratory testing works best when the software is ready to be explored—that is, when stories are "done done." You don't have to test stories that were finished in the current iteration. It's nice to have that sort of rapid feedback, but some stories won't be "done done" until the last day of the iteration. That's OK—remember, you're not using exploratory testing to guarantee quality; you're using it provide information about how the team's *process* guarantees quality.

Ally

"Done Done" (p. 155)

NOTE

This switch in mindset takes a little while to get used to. Your exploratory testing is not a means of evaluating the software through exhaustive testing. Instead, you're acting as a technical investigator—checking weak points to help the team discover ways to prevent bugs. You can also use exploratory testing to provide the team with other useful data, such as information about the software's performance characteristics.

Exploratory testers use the following four tools to explore the software.

Tool #1: Charters

Some people claim that exploratory testing is simply haphazard poking at the software under test. This isn't true, any more than the idea that American explorers Lewis and Clark mapped the Northwest by haphazardly tromping about in the woods. Before they headed into the wilderness, Lewis and Clark knew where they were headed and why they were going. President Thomas Jefferson had given them a charter:*

> The Object of your mission is to explore the Missouri river & such principal stream of it as by [its] course and communication with the waters of the Pacific ocean, whether the Columbia, Oregon, Colorado or any other river may offer the most direct & practicable water communication across this continent for the purpose of commerce.

Similarly, before beginning an exploratory session, a tester should have some idea of what to explore in the system and what kinds of things to look for. This *charter* helps keep the session focused.

NOTE
An exploratory session typically lasts one to two hours.

The charter for a given exploratory session might come from a just-completed story (e.g., "Explore the Coupon feature"). It might relate to the interaction among a collection of stories (e.g., "Explore interactions between the Coupon feature and the Bulk Discount feature"). It might involve a quality attribute, such as stability, reliability, or performance ("Look for evidence that the Coupon feature impacts performance"). Generate charters by working with the team to understand what information would be the most useful to move the project forward.

Charters are the testing equivalent of a story. Like stories, they work best when written down. They may be as informal as a line on a whiteboard or a card, but a written charter provides a touchstone the tester can refer to in order to ensure she's still on track.

Tool #2: Observation

Automated tests only verify the behavior that programmers write them to verify, but humans are capable of noticing subtle clues that suggest all is not right. Exploratory testers are continuously alert for anything out of the ordinary. This may be an editable form field that should be read-only, a hard drive that spun up when the software should not be doing any disk access, or a value in a report that looks out of place.

Such observations lead exploratory testers to ask more questions, run more tests, and explore in new directions. For example, a tester may notice that a web-based system uses parameters in the URL. "Aha!" thinks the tester. "I can easily edit the URL." Where the URL says *http://stage.example.com/edit?id=42*, the tester substitutes *http://stage.example.com/edit?id=9999999*.

* *http://www.lewis-clark.org/content/content-article.asp?ArticleID=1047*.

Tool #3: Notetaking

While exploring, it's all too easy to forget where you've been and where you're going. Exploratory testers keep a notepad beside them as they explore and periodically take notes on the actions they take. You can also use screen recorders such as Camtasia to keep track of what you do. After all, it's quite frustrating to discover a bug only to find that you have no idea what you did to cause the software to react that way. Notes and recordings tell you what you were doing not just at the time you encountered surprising behavior but also in the minutes or hours before.

Be especially careful to keep notes about anything that deserves further investigation. If you cannot follow a path of testing in the current session, you want to remember to explore that area in more detail later. These are opportunities for further exploratory testing.

Tool #4: Heuristics

Remember that exploratory testing involves simultaneously designing and executing tests. Some test design techniques are well known, such as *boundary testing*. If you have a field that's supposed to accept numbers from 0–100, you'll probably try valid values like 0, 100, and something in the middle, and invalid values like –1 and 101. Even if you had never heard the term boundary testing, you'd probably consider trying such tests.

A *heuristic* is a guide: a technique that aids in your explorations. Boundary testing is an example of a test heuristic. Experienced testers use a variety of heuristics, gleaned from an understanding of how software systems work as well as from experience and intuition about what causes software to break. You can improve your exploratory efforts by creating and maintaining a catalog of heuristics that are worth remembering when exploring your software. Of course, that same catalog can also be a welcome reference for the programmers as they implement the software.

Some of your heuristics will be specific to your domain. For example, if you work with networking software, you must inevitably work with IP addresses. You probably test your software to see how it handles invalid IP addresses ("999.999.999.999"), special IP addresses ("127.0.0.1"), and IPv6 style addresses ("::1/128"). Others are applicable to nearly any software project. The following are a few to get you started.

None, Some, All

For example, if your users can have permissions, try creating users with no permissions, some permissions, and all permissions. In one system I tested, the system treated users with no permissions as administrators. Granting a user no permissions resulted in the user having root access.

Goldilocks: too big, too small, just right

Boundary tests on a numeric field are one example of Golidlocks tests. Another example might be uploading an image file: try uploading a 3 MB picture (too big), a file with nothing in it (too small), and a file of comfortable size (50 KB).

Position: beginning, middle, end

Position can apply to an edit control: edit at the beginning of a line, middle of a line, end of a line. It can apply to the location of data in a file being parsed. It can apply to an item selected from a list, or where you're inserting an item into a list.

Count: zero, one, many

For example, you could create invoices with zero line items, one line item, and many line items. Or you might perform a search for zero items, one item, or many pages of items. Many systems have a small typographical error related to plurals—that is, they report "1 items found" instead of "1 item found."

Similarly, count can apply in numerous situations: a count of dependent data records (a customer with zero addresses, one address, many addresses, or a group with zero members, one member, many members), a count of events (zero transactions, one transaction, many simultaneous transactions), or anything else that can be counted.

CRUD: create, read, update, delete

For each type of entity, and each data field, try to create it, read it, update it, and delete it. Now try CRUD it while violating system constraints such as permissions. Try to CRUD in combination with Goldilocks, Position, Count, and Select heuristics. For example, delete the last line item on an invoice, then read it; update the first line item; delete a customer with zero, one, or many invoices, etc.

Command Injection

Wherever your software supports text coming from an external source (such as a UI or a Web Services interface), ensure it doesn't ever interpret incoming text as a command—whether an SQL, a JavaScript, or a shell/command-line command. For example, a single quote in a text field will sometimes raise SQL exceptions; entering the word `tester's` will cause some applications to respond with an SQL error.

Data Type Attacks

For each type of input, try values that push common boundaries or that violate integrity constraints. Some examples: in date fields, try February 29 and 30. Also try dates more than 100 years ago. Try invalid times, like 13:75. For numbers, try the huge (4,294,967,297 = 2^32 + 1) and the tiny (0.00000000000000001). Try scientific notation (1E-16). Try negative numbers, particularly where negative numbers should not be allowed, as with purchase price. For strings, try entering long strings (more than 5,000 characters), leaving fields blank, and entering a single space. Try various characters including common field delimiters (` | / \ , ; : & < > ^ * ? Tab). The list could go on, but you get the idea.

An Example

"Let's decide on a charter for this session," Jill said, settling into the seat next to Michael. Jill was one of the team's testers; Michael was a programmer. "What should we focus on?"

"You know us," Michael replied. "Our code doesn't have any bugs!" He grinned, knowing what was coming.

"Oooh, you'll pay for that one." Jill smiled. "Although I have to say our quality is better than any other team I've worked with. How many bugs did we have last month?"

"It's been a bit high, actually." Michael counted on his fingers. "There was that installer issue... and the networking problem...." He paused. "Four."

"Let's see how that holds up. What's new in the code?"

"Some fairly routine stuff," Michael said. "One thing that's brand-new is our support for multilingual input. But there's nothing to find there—we tested it thoroughly."

"I love a challenge!" Jill said with a wicked grin. "Besides, character-set issues are a rich source of bugs. Let's take a look at that."

"Sure." Michael wrote a card for the session: Explore Internationalization. He clipped the card to the monitor so it would remind them to keep on track.

Jill took the keyboard first. "Let's start with the basics. That will help us know what's going wrong if we see anything strange later on." As she navigated to a data entry screen, Michael wrote the date and time at the top of a new page in his notepad and prepared to take notes.

Jill opened up a character viewer and pasted together a string of gibberish: German, Hebrew, Greek, Japanese, and Arabic characters. "Let's use the CRUD heuristic to make sure this stuff gets into the database and comes back properly." Moving quickly, she saved the text, opened up a database viewer to make sure it was present in the database, then closed the application and reloaded it. Everything was intact. Next, she edited the string, saved it again, and repeated her check. Finally, she deleted the entry. "Looks good."

"Wait a minute," Michael said. "I just had an idea. We're not supposed to allow the user to save blank strings. What if we use a Unicode space rather than a regular space?"

"I'll try it." Jill went back to work. Everything she tried was successfully blocked. "It all looks good so far, but I have a special trick." She grinned as she typed `#FEFF` into her character viewer. "This character used to mean 'zero-width no-break space.' Now it's just a byte-order mark. Either way, it's the nastiest character I can throw at your input routines." She pressed Save.

Nothing happened. "Score one for the good guys," Jill murmured. "The data input widgets look fairly solid. I know from past experience that you've abstracted those widgets so that if one is working, they're probably all working." Michael nodded, and she added, "I might want to double-check a few of those later, but I think our time is better spent elsewhere."

"OK, what should we look at next?" Michael asked.

"A few things come to mind for us to check: data comparison, sorting, and translation to other formats. Unicode diacritical marks can be encoded in several ways, so two strings that are technically identical might not be encoded using the same bytes. Sorting is problematic because Unicode characters aren't sorted in the same order that they're represented in binary..."

"...and format translation is nasty because of all the different code page and character-set issues out there," Michael finished. He wrote the three ideas—data comparison, sorting, and translation—on his notepad.

"You said it," Jill agreed. "How about starting with the nasty one?"

"Sounds good." Michael reached for the keyboard. "My turn to type," he said, smiling. He handed Jill the notepad so she could continue the notetaking. "Now, where could the code page issues be lurking...?"

When You Find Bugs

Exploratory testing provides feedback about the software and also about the effectiveness of the team's process. When it reveals a bug, it indicates that the team may not be working as effectively as it could. To remedy, fix your software *and* your process as described in "No Bugs"" in Chapter 7.

Ally
No Bugs (p. 159)

If you've already used the feedback from exploratory testing to improve both the software and the process, but you're consistently finding a lot of bugs, it means the process is still broken. Don't give up: look for root causes and keep plugging away. It's often a simple case of trying to do too much in too little time.

When the bugs are rampant, you may be tempted to add a QA department to catch the bugs. This may provide a temporary fix, but it's the first step down a slippery slope. Here is an example:

Imagine a team that has been struggling with velocity because stories just never seem to be "done done," as is often the case when customers finds bugs in "done" stories. The programmers are frustrated that they can't seem to prevent or detect the problems that result in their customers rejecting stories.

"I give up," says Wilma to her partner, Betty. "This story is taking too long, and there are too many other stories in the queue. Let's ask Jeff to test this and tell us where all the problems are. He's good at finding bugs."

In fact, Jeff is great at finding bugs. The next morning he delivers a stack of sticky notes to Wilma and Betty that detail a bunch of bugs. They fix the bugs and deliver the story. The customer accepts it, smiling for the first time in weeks. "That was *great*!" says Wilma. "We finally *finished* something!"

On the next story, Betty turns to Wilma and says, "You know, Jeff was so good at finding bugs in the last story...."

"Yeah," Wilma agrees. "Even though there are some more error conditions we could think through, let's see what Jeff has to say."

The pattern continues. The more that programmers rely on testers to find bugs *for* them, the fewer bugs they find themselves. Testers find more and more bugs, but in the end, quality gets worse.* The key to having no bugs is not to get better testers, but for the team to take responsibility for producing bug-free software—*before* testers try it. Instead of relying on testers to get the bugs out, use the information that exploratory testing provides to improve your process.

Ally
No Bugs (p. 159)

Questions

Should testers pair with other members of the team?

* I wrote about this effect in "Better Testing, Worse Quality," at *http://testobsessed.com/wordpress/wp-content/uploads/2006/12/btwq.pdf*. Page past the end of the slideshow to find it.

This is up to your team. Pairing with programmers and customers helps break down some of the natural barriers between testers and other team members, and it helps information flow better throughout the team. On the other hand, programmers may not always have time to pair with testers. Find a good balance.

Won't the burden of exploratory testing keep getting bigger over the course of the project?

It shouldn't. Sometimes teams use exploratory testing as a form of manual regression testing; with each iteration, they explore the new features, and the existing features, and the interactions, and so on. They put so much on the list of things to explore that the time needed for exploratory testing during the iteration becomes unmanageable.

The flaw in this approach is using exploratory testing as a means of regression testing. Use test-driven development to create a comprehensive, automated regression test suite. Focus your exploratory testing on new features (and their interactions with existing features), particularly those features that do things differently from previous features.

Ally

Test-Driven Development (p. 287)

Just as you timebox your releases and work on just the most important features, you can timebox your explorations and test just the most important charters.

How can we get better at exploratory testing?

Exploratory testing involves a set of skills that can be learned. To improve, you should:

Practice
: You can test more than the software your team is developing. Whenever you use software for your own purposes, like word processing or email, try testing it. Also consider applying your testing skills to open source projects.

Get feedback
: Find out what surprises other people discovered, and ask yourself why you didn't see them. Sometimes it's because there were more test conditions you might have tried, so you can add new tricks to your heuristics toolbox. Other times it's because you didn't understand the significance of what you were seeing; you saw the same behavior but didn't recognize it as a bug.

Share tips and techniques
: You can share ideas within your team, and you can reach out to the broader community. Online discussion forums are a good place to start. Other options include round table–style meetings. For example, James Bach hosts WHET, the Workshop on Heuristics and Exploratory Techniques, and James Lyndsay hosts LEWT, the London Exploratory Workshop in Testing. Both are gathering places where testers share stories and experiences.

Results

When you use exploratory testing, you discover information about both the software and the process used to create that software. You sometimes discover missing test cases, incomplete or incorrect understanding of the story, and conversations that should have happened but didn't. Each surprise gives you an opportunity to improve both the software and your development practices. As a team, you use this information to improve your process and reduce the number of bugs found in the future.

Contraindications

Don't attempt to use exploratory testing as a regression testing strategy. Regression tests should be automated.

Only do exploratory testing when it is likely to uncover new information and you are in a position to act on that information. If, for example, there is already a list of known bugs for a given story, additional exploratory testing will waste time rediscovering known issues. Fix the bugs first.

Alternatives

You can use exploratory testing as a mechanism for bug hunting when working with software, particularly legacy software, that you suspect to be buggy. However, beware of relying on exploratory testing or any other testing approach to ensure all the bugs are caught. They won't be.

Some teams don't do any manual or end-to-end testing with XP. These teams are using another mechanism to confirm that they don't produce bugs—presumably, they're relying on user feedback. This is OK if you actually don't produce bugs, which is certainly possible, but it's better to confirm that *before* giving software to users. Still, you might get away with it if the software isn't mission-critical and you have forgiving users.

Other teams use testers in a more traditional role, relying on them to find bugs rather than committing to deliver bug-free code. In my experience, these teams have lower quality and higher bug rates, probably as a result of the "Better Testing, Worse Quality" dynamic.

Further Reading

"General Functionality and Stability Test Procedure for Microsoft Windows Logo, Desktop Applications Edition" [Bach 1999] is the first widely published reference on how to do exploratory testing. The WinLogo certification assures customers that the software behaves itself on Windows. But how do you create a systematic process for assessing the capabilities and limitations of an arbitrary desktop application in a consistent way? Microsoft turned to James Bach, already well known for his work on exploratory testing. The resulting test procedure was made available to independent software vendors (ISVs) and used by certifiers like Veritest. It's online at *http://www.testingcraft.com/bach-exploratory-procedure.pdf*.

"Session-Based Test Management" [Bach 2000] is the first published article that explains in detail how to use session-based test management, which James Bach and his brother Jonathan devised, and which I discussed as Charters and Sessions earlier in this chapter. One of the challenges of exploratory testing is how to manage the process: stay focused, track progress, and ensure the effort continues to yield value. This is the solution. The article is online at *http://www.satisfice.com/articles/sbtm.pdf*.

"Did I Remember To" [Hunter] is a great list of heuristics, framed as things to remember to test. The list is particularly great for Windows applications, but it includes ideas that are applicable to other technologies as well. It's online at *http://blogs.msdn.com/micahel/articles/175571.aspx*.

"Rigorous Exploratory Testing" [Hendrickson] sets out to debunk the myth that exploratory testing is just banging on keyboards, and discusses how exploratory testing can be rigorous

without being formal. The article is online at *http://www.testobsessed.com/2006/04/19/rigorous-exploratory-testing/*.

"User Profiles and Exploratory Testing" [Kohl 2005a] is a nice reminder that different users experience software in different ways, and it provides guidance on characterizing user behavior to support exploratory testing. It's online at *http://www.kohl.ca/blog/archives/000104.html*.

"Exploratory Testing on Agile Teams" [Kohl 2005b] presents a case study of using automation-assisted exploratory testing to isolate a defect on an Agile project—which flouts the belief that exploratory testing is a purely manual activity. It complements this chapter nicely. Read it online at *http://www.informit.com/articles/article.asp?p=405514&rl=1*.

"A Survey of Exploratory Testing" [Marick] provides a nice collection of published work related to exploratory testing, including [Bach 1999]'s "General Functionality and Stability Test Procedure for Microsoft Windows Logo." Online at *http://www.testingcraft.com/exploratory.html*.

"Exploring Exploratory Testing" [Tinkham & Kaner] is an in-depth discussion of how exploratory testers do what they do, including strategies such as questioning and using heuristics. Based on work funded in part by an NSF grant, this paper elaborates on exploratory testing practices missing from the then-current draft of the IEEE's *Software Engineering Body of Knowledge* (SWEBOK). Available online at *http://www.testingeducation.org/a/explore.pdf*.