UNIT-5

Developing

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.

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.

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

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. 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 Many of the requirements for your software will be intuitively obvious to the right customers.

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.

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"**

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

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.

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.

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.

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

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. 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, the purpose of the review is to bring customers' expectations and programmers' work into alignment.

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 unnecessary.

Customer Tests

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. 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. 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. 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.

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	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

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 testdriven 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.

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

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.

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.

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. 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 :. 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.

Questions

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. 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. 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 and a variety of other techniques.

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.

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. 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. 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.

Test-Driven Development

We produce well-designed, well-tested, and wellfactored 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.

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.

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.

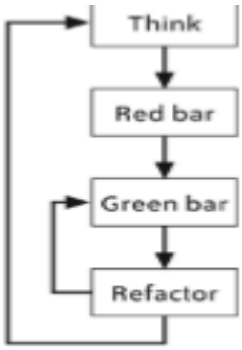

Fig:The TDD cycle

To use TDD, follow the “red, green, refactor” cycle illustrated in Figure . 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.

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.

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. 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.

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. 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. 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.

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.

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.

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.
```

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.

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`.

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. 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.

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.

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.

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. 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,
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.

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 welltested 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.

Refactoring

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.

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.

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 . 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.

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.

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. 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. There are a wide variety of individual refactorings. 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.

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. 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 hardwon 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.

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.

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.

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.

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. 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. 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.

Incremental Design and Architecture

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. 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. 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. 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. 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. 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.

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.

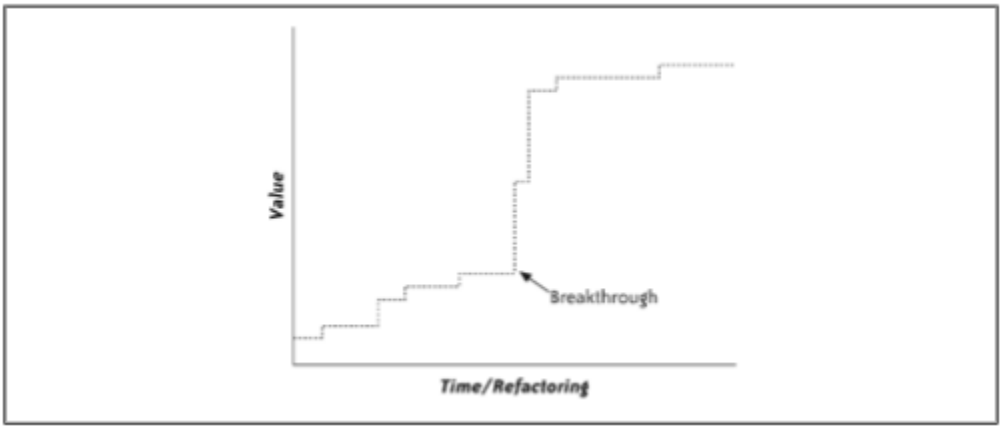

Fig: Breakthrough

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.
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. 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.
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.

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, 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. 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.

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
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. 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.

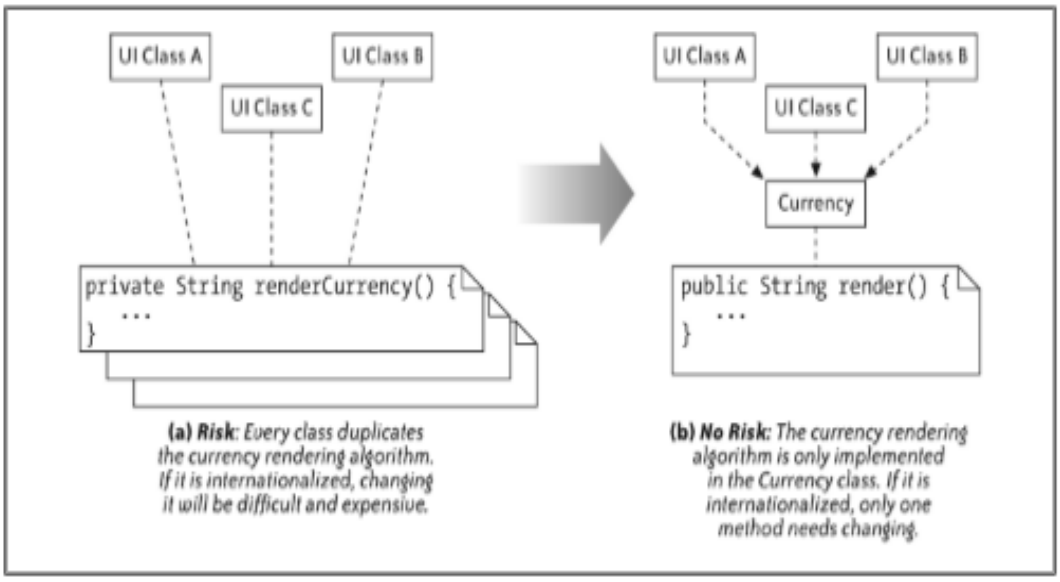

Fig: Use risk to drive refactoring

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 . 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.

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. 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. 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. 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. 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. 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.

Spike Solutions

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
```

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.

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. 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.

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 shortterm 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.

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

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.

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.

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. 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. 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.

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. 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. 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.

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.

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 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 is appropriate. Run your performance tests asynchronously, as a separate build from your standard 10-minute build (see "Ten-Minute Build"), 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. 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 String Buffer 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.