The XP Series

Extreme Programming *Explored*

William C. Wake

Foreword by Dave Thomas

Extreme Programming Explored

by
William C. Wake

William.Wake@acm.org
804-934-8194 (w)

To May, Tyler, and Fiona,
with love.

CONTENTS

Section 3: Process

Preface

Extreme Programming (XP) defines a process for developing software: it addresses the problem from early exploration to delivery.

We'll explore XP from the inside to the outside.

First, XP is a programming discipline. We'll look at a core innovation: how "test-first" changes the programming process itself. We'll also discuss refactoring, the way XP programmers improve their code.

Second, XP is a team discipline. It has evolved a number of practices that help produce a high-performing team. We'll compare XP to alternative practices, and see some of them in action.

Finally, XP is a discipline for working with customers. XP has specific processes for planning and daily activity. We'll see how a team might schedule a release or iteration, and what the team does all day.

Why read this book?

If you've heard anything about Extreme Programming, you have probably had a number of questions about the mechanics or the purposes of various aspects of XP. I've tried to capture the questions I had, along with answers I've found.

Several things about XP were surprises to me, particularly the tight cycle of test-first programming (only a couple minutes long), the use of a metaphor, and the starkness of the division of labor between customer and programmer. We'll look at these, and many other topics.

You, the reader, may have several areas of interest that bring you to this book:

- Java and object-oriented programming. The first section of the book uses Java examples to focus on test-first programming and refactoring. Programmers may find the discussion of team practices useful as well, particular the ideas about metaphors and simple design.
- Extreme programming, from the perspectives of programmer, customer, and manager. We'll explore several areas more deeply or from a different perspective than the rest of the XP literature, especially the team-oriented practices, the metaphor, the planning process, and daily activities.
- Software process in general. XP is one of a number of so-called "lightweight" or "adaptive" processes that have been introduced in the last few years. By looking at XP's process more deeply, we can more clearly delineate where XP fits in with these related processes.

Philosophy of this book

Be concrete. Use real (or at least realistic) examples. When there's code, it will be Java.

Answer questions. Many of the chapters were originally written as essays for myself as I learned or taught others. Thus, each

chapter starts with a question and a short answer, and many chapters include a Q&A (question and answer) section as well.

Be focused. Make each chapter focus on one topic. Tie it to other chapters where possible.

Be precise but informal. I'll use "I," "we," and "you" a lot. For the most part, "you" is typically a programmer, but may be addressed to managers or customers in some sections.

Bring experiences to bear. I'll relate this material to real experiences.

Acknowledgements

I've always seen books with an "acknowledgements" section listing all the people who helped the author directly or otherwise. Now I'm in the position of creating such a list, and realizing I've certainly forgotten the names of many who helped get me here. So, my apologies to those I've forgotten, and my thanks to all.

People in the XP community, for conversations small and large: Ann Anderson, Ken Auer, Tom Ayerst, Serge Beaumont, Kent Beck, Bill Caputo, Ward Cunningham, Quenio dos Santos, Joi Ellis, Chris Fahlbusch, Ed Falis, Martin Fowler, Steve Freeman, Peter Gassmann, Dan Green, Chet Hendrickson, Jim Highsmith, Michael Hill, Kari Hoijarvi, Andy Hunt, Dwight Hyde, Andrey Khavryutchenko, William Kleb, Malte Kroeger, Bob Martin, Duncan McGregor, Jim Mead, Zohar Melamed, Chris Morris, Miroslav Novak, Shinichi Omura, Christian Pekeler, Rekha Raghu, Don Roberts, Gilbert Semmer, Sinan Si Alhir, Dave Thomas, Philippe Vanpeperstraete, Joseph Vlietstra, Doug Wake (brother), Daniel Weinreb, Don Wells, Frank Westphal, Roger Whitney, Laurie Williams, Mark Windholtz, Torben Wölm, Diane Woods, Park Sung Woon, Tilak Yalamanchili, Fred Yankowski, Jason Che-han Yip, and others. Not everybody agreed with me on everything, of course, but I appreciate the interaction. Ken, Ward, Martin, Bob, Frank, and others encour-

aged me to publish this; and Kent Beck and Dave Thomas in particular have given me inspiration, encouragement, and critique.

Past and present co-workers at Capital One, for discussions and the opportunity to teach and learn: Paul Given, Harris Kirk, Michele Matthews, Steve Metsker, Tim Snyder, Steve Wake, Joe Wetzel, and others. Harris and Steve Metsker have read at least a couple drafts and given me good feedback; Steve Wake (brother) has read through and critiqued more drafts than anyone, and he helped brainstorm the metaphor catalog.

Those who have let me borrow some of their words: Harris Kirk, Bob Koss, Steve Metsker, Ron Jeffries, and Christopher Painter-Wakefield.

At Addison-Wesley: Mike Hendrickson, and especially Ross Venables, my editor.

My family: my parents, Caroline and Bill; my siblings, Becky, Lynn, Doug, and Steve; and especially May, Tyler, and Fiona, who support my "always scribbling."

—William C. Wake

William.Wake@acm.org

http://www.xp123.com

Chapter 1

Introducing XP

Programming, team practices, and processes.

Extreme Programming ("XP") is a new, lightweight approach to developing software. XP uses rapid feedback and high-bandwidth communication to maximize delivered value, via an on-site customer, a particular planning approach, and constant testing.

Let's consider a traditional approach to software development:

> The user's (customer's) group arranges with the development group to have an analyst assigned for a project. Over a series of weeks and months, the analyst meets with the users for several hours a week. The analysts produce a set of documents, perhaps including things like a Vision Statement or Use Cases. The users and the project manager (and perhaps the programming team as well) review these documents and negotiate a release.
>
> The programmers take the specifications, and several months later produce a system that more or less does what it was intended to. It's often a close call at the

end, as people find out what they missed, and realize what's changed since the documents were written. In the end, the customers come in to do a user acceptance test, and the system is released.

Often the whole process took longer than anybody had expected, several features are missing, and the quality is not where the users want it. Furthermore, the documents are no longer up to date.

Some teams are more iterative:

During development, the team builds a full version of the system, perhaps every 6-8 weeks.

In the best case, analysis and development proceed in parallel, supporting each other.

An XP approach emphasizes customer involvement and testing:

The customer contacts an XP development group to start a project. The team asks that the customer sit with their team during development. The project has three phases:

- a release planning phase, where the customer writes stories, the programmers estimate them, and the customer chooses the order in which stories will be developed;

- an iteration phase, where the customer writes tests and answers questions, while the programmers program; and

- a release phase, where the Programmers install the software, and the customer approves the result.

The customer in XP has frequent opportunities to change the team's direction if circumstances change: the iteration phase is providing ready-to-go software

every two weeks. Because testing is so prominent, the customer is aware of the project's true status much earlier in the cycle.

In Extreme Programming, there is a fundamental divide in the roles of Customer and Programmer. Both are on the same team, but they have different decisions to make. The Customer owns "what you get" while the Programmer owns "what it costs." This shows up in who gets to make which decisions:

The Customer gets to decide:

- Scope: what the system must do
- Priority: what is most important
- Composition of Releases: what must be in a release to be useful
- Dates of Releases: when the release is needed.

The Programmers get to decide:

- Estimated time to add a feature
- Technical consequences: Programmers explain the consequences of technical choices, but the Customer makes the decision
- Process: how the team will work
- Detailed Schedule (within an iteration)

[summarized from *Extreme Programming Explained,* p. 55]

The issue of technical consequences comes up because the customer has to live with the result. For example, an object-oriented database might let the team work faster, but the customer might still prefer a relational database for reasons such as risk management and minimizing disruption of their support process. The developers might be 100% right: "Yes, it would be faster to develop that way," but development time is not the customer's only consideration.

The practices of XP help enforce this split between customer and programmer responsibilities. This division of labor helps keep the whole team on track by making the consequences of decisions be very visible. For example, if a customer wants the software to generate a new report this week, the XP team is happy to provide

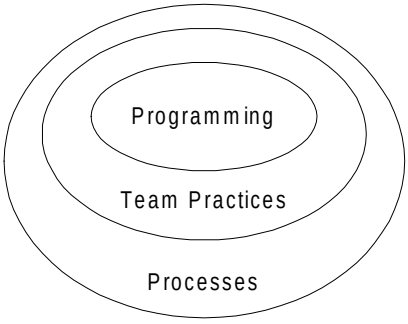
it. They'll report the technical risk (if any), and estimate what it will cost. Then the customer gets to pick what will be dropped to make time for the development.

What happens if there's a conflict? What if the customer wants these features by this date, but the programmers estimate that it will take longer than that? XP provides several options: the customer can accept less scope, the customer can accept a later date, the customer may spend time or money exploring an alternative, or the customer can find a different programming team. What XP doesn't let you do is say, "Let's try anyway—we'll catch up later."

XP is an Onion

You can think of XP as an onion.

The innermost layer is programming. XP uses a particular programming style, usable even by a solo developer. (But as we'll see later, XP discourages solo development.) The middle layer consists of a set of team-oriented practices. The outer layer defines the process by which a programming team interacts with its customer.



This book reflects the layered approach, with sections corresponding to the three layers. Each chapter answers questions pertinent to its section, and reflects questions I've had to answer in learning XP.

XP as "The Twelve Practices"

Another (related) way to look at XP is as a set of practices. In *Extreme Programming Explained*, Kent Beck lays out a set of twelve core practices that serve as a starting point for an XP team.

We'll map those practices to the layers like this:

- Programming: Simple design, testing, refactoring, coding standards.
- Team practices: Collective ownership, continuous integration, metaphor, coding standards, 40-hour week, pair programming, small releases.
- Processes: On-site customer, testing, small releases, planning game.

Notice there's some overlap as a few practices cross categories.

Underlying these practices are four values: simplicity, communication, feedback, and courage.

Section 1: XP as Programming

XP programmers write code via incremental test-first programming: unit-test a little at a time, then code just enough to make the test work. There is always a test to justify any new code. We'll develop the core of a library search system using this approach.

A program isn't done just because it happens to work. XP strives to keep the whole system as flexible as possible by keeping it as simple as possible. Refactoring improves the code while ensuring it still passes all its tests. We'll refactor a Java program that generates a web page. By cleaning up the obvious problems, we ready the code for radical design changes.

Section 2: XP as Team Practices

We'll look at the team-oriented practices and some alternatives to them: code ownership, integration, overtime, workspace, release schedule, and coding standard.

Pair programming provides on-the-spot design and code reviews for each line of production code. Many teams struggle to introduce it, but XP regards pairing as a key mechanism for ensuring quality and team learning. We'll look at a dialogue where two programmers pair on a tricky problem.

XP doesn't emphasize "architecture" as a driving mechanism, but XP programs do have an architecture. The metaphor, the design, and the development process contribute to an XP program's architecture.

The metaphor provides a conceptual framework for a system. We'll finish the section by exploring how metaphors can drive the conceptualization and realization of a couple different types of systems.

Section 3: XP as Processes

The on-site customer and automated acceptance tests are important aspects of XP, but we won't delve into them much beyond this section. We will address the two types of planning, and the day-to-day activities of the team.

On-Site Customer

A customer of XP software sits with the team full-time to write tests, answer questions, and set priorities. Having the customer with the team is crucial in helping the team go as fast as possible.

I once worked with a cross-functional team that had moved into one large room for a key project. In the "lessons learned" session, a manager of the marketing group said he thought the experience was great, that he was answering questions right when

they came up and he could see the progress resulting from his answers. He admitted that when he was back in his office, he'd get a voicemail, but put off answering for a day or two. (Meanwhile, the developers either made a guess, or worked on something less important.)

It takes many decisions to develop software. If a team can get its answers or decisions in minutes instead of hours or days, its speed will be much higher.

XP is not the only approach that recognizes the value of an onsite customer. Tom Peters says: "Make clients an integral part of every project team." [*The Professional Service Firm 50,* p. 105] and "If the client won't give you full-time, top-flight members, beg off the project. The client isn't serious." [ibid., p. 106].

Testing

XP uses testing two ways: customers develop acceptance tests that determine the overall behavior of the system, and programmers create unit tests while programming. (We'll cover unit testing in the programming section.)

The customer creates tests for each story they request. XP teams measure the progress in developing the system by running these tests. The tests form an existence proof for the testability of the features requested.

Finally, the team should strive to have its tests automated. This may mean the customer will want to spend some development time on a testing infrastructure, but it is worthwhile. These tests don't need a fancy screen, just a way to specify the inputs and the expected result. Some teams have been able to use spreadsheets for this: the user has a familiar interface, and the programmers can write a small program to read the spreadsheet and run the test.

Planning Game

XP uses the notion of a planning game for planning the release (weeks to months) and for planning iterations (one to three

weeks). The planning game uses two types of players, Customer and Programmer, and defines which type of player can make which move. In this way, the process maintains a critical division of labor: Customer determines value, Programmer determines cost.

In the *release planning game*, the goal is to define the set of features required for the next release. It is centered around user stories. The Customer writes stories, the Programmers estimate the stories, and the Customer plans the overall release. We'll discuss the release planning process, and show examples of stories a customer might write.

The *iteration planning game* is similar. The Customer chooses the stories for the iteration, and the Programmers estimate and accept the corresponding tasks. We'll describe this process as a board game.

Finally, we'll describe the activities that the customer, programmer, and manager will take part in during the day-to-day work of an iteration.

XP Resources

For more information, see references at the end of each chapter, and the bibliography at the end of the book.

Section 1: Programming

Chapter 2

How do you program in XP?

XP uses incremental, test-first programming.

Incremental, because we program in a tight cycle: not even a whole class at a time, but rather a few lines of code or a method at a time.

Test-first, because we write automated, repeatable unit tests before writing the code that makes them run.

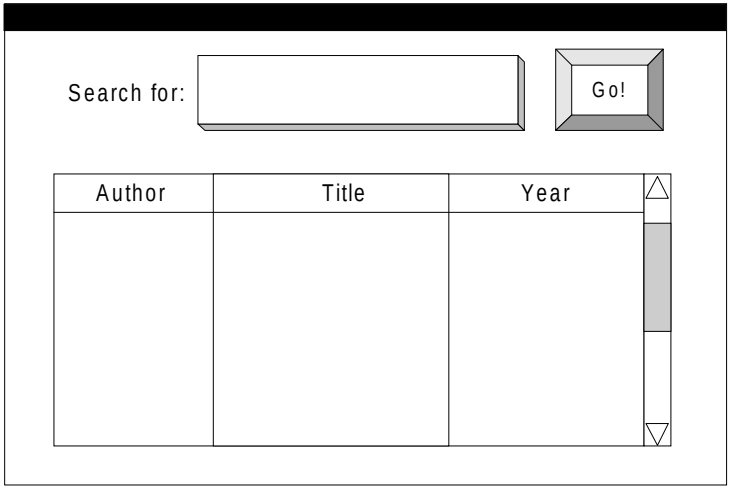
This approach yields several benefits:

- The code is testable: it was built to be so.
- The test is tested: we saw the test fail when the code to support it wasn't there, and saw it pass when the code was added.
- The tests are captured, as executable code.
- The tests help document the code: anybody who needs to see how the object works, or needs to make changes, can see and run the tests.
- We get "just enough design" to support the tests we've written.

We will develop a small bibliographic system, test-first. We will see how unit tests and simple design work together. The coding process occurs in small steps, with just enough new code to make each test run. There's a rhythm, like a pendulum on a clock: test a little, code a little, test a little, code a little.

Design

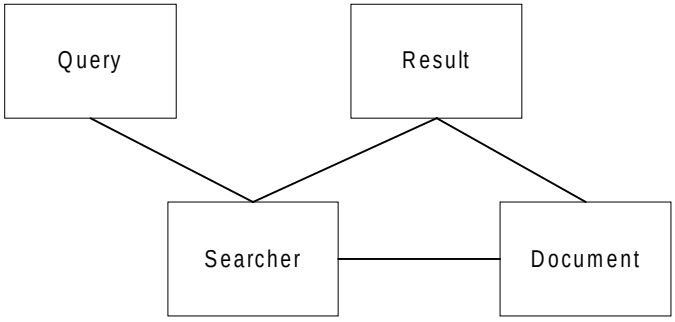
Suppose we have bibliographic data with author, title, and year of publication. Our goal is to write a system that can search that information for values we specify. We have in mind an interface something like this:



We'll divide our work into two parts: the model and the user interface. This chapter demonstrates test-first programming in development of the model. For a quick note on how the test-first approach can be applied to user interfaces, see the sidebar at the end of this chapter.

We'll begin with a brief design session. Initially, we know we have a collection of Documents. Documents know their attributes (author, title, and year).

A Searcher knows how to find Documents: given a Query, it will return a Result (the set of matching Documents).



Notice that our design fits onto a handful of cards. As we'll discuss in the chapter on metaphors, our design uses the "naive metaphor": objects are based on domain objects rather than some other metaphor.

Document, Result, and Query

Following the design above, we'll create our unit tests (and our classes) bottom-up: Document, Result, Query, and Searcher.

Document

A Document needs to know its author, title, and year. We'll begin by creating a "data bag" class, beginning with its test:

```
public void testDocument() {
   Document d = new Document("a", "t", "y");
   assertEquals("a", d.getAuthor());
   assertEquals("t", d.getTitle());
```

```
    assertEquals("y", d.getYear());
}
```

This test doesn't compile (as we haven't created the Document class yet). Create the class with stubbed-out methods (just enough to compile). (This test uses the JUnit testing framework developed by Kent Beck and Erich Gamma, available at www.junit.org.)

Run the test again to make sure it fails. This may seem funny—don't we want the tests to pass? Yes, we do. But by seeing them fail first, we get some assurance that the test is valid. And once in a while, a test passes unexpectedly: "that's interesting!"

Fill in the constructor and the methods to make the test pass.

Let's highlight this mini-process:

The Test/Code Cycle in XP

- Write one test.
- *Compile the test.* It should fail to compile, as you haven't yet implemented the code that the test calls.
- Implement just enough to compile. (Refactor first if necessary.)
- *Run the test and see it fail.*
- Implement just enough to make the test pass.
- *Run the test and see it pass.*
- Refactor for clarity and to remove duplication.
- Repeat from the top.

(We'll look at refactoring in the next chapter.)

This process ensures that you've seen the test both fail and pass, which gives you assurance that the test did test something, that your change made a difference, and that you've added valuable functionality.

Kent Beck and others will tell you not to bother writing tests for simple setters, getters, and constructors: you only need to "test everything that could possibly break" (and those can't break). In spite of this, I have used some setter and getter tests in the example, but my own style is moving away from bothering with them. The only remnant of this is that I try to be aware that a large number of setters and getters can be a sign that the class may not be pulling its weight, and there may be a better re-distribution of class responsibilities.

How long does this cycle take? From one to five minutes, maybe ten on the outside.

What if it's taking longer? Move to smaller tests.

Really—five minutes?? Yes.

Isn't this bouncing around between test and code kind of high overhead? Not really. For example, in VisualAge for Java, you write the test and click "save"; the environment tells you of a compiler error, so you write the stub and "save"; then you click the "Run" button of JUnit (which just stays running); this re-loads the new classes and runs the tests again, which show red; then write the real code, "save," and run the tests one last time for a green bar.

If you were planning to write unit tests anyway, all you've added is the burden of writing a failing stub and running the tests an extra time to see them fail. Even if you're working from a command line, it's just not that bad, as you can keep windows open for compilers and so on.

Won't all this test code slow you down later during maintenance? If things change, you have to update tests as well as code. No, the tests actually speed you up in maintenance, because they give you the confidence to change things, knowing that a test will warn you if you've done something wrong. If interfaces change, you do have to change tests as well, but that's not really that hard. Furthermore, you may find that test-first programming tends to drive you toward a more stable design in the first place.

Result

A Result needs to know two things: the total number of items, and the list of Documents it contains. First we'll test that an empty result has no items.

```
public void testEmptyResult() {
   Result r = new Result();
   assertEquals (0, r.getCount());
}
```

Create the Result class and stub out its `getCount()` method. See it fail until you add `"return 0;"` as its implementation. Notice how we've put the simplest possible solution in place.

Next test a result with two documents.

```
public void testResultWithTwoDocuments() {
    Document d1 = new Document("a1", "t1", "y1");
    Document d2 = new Document("a2", "t2", "y2");
    Result r = new Result(new Document[]{d1, d2});
    assertEquals (2, r.getCount());
    assert (r.getItem(0) == d1);
    assert (r.getItem(1) == d2);
}
```

Add the `getItem()` method (returning null) and watch the test fail. (I'm going to stop mentioning that, but keep doing it. It takes a few seconds, but gives you that extra bit of reassurance that your test is valid.) Implementing a simple version of Result will give:

```
public class Result {
   Document[] collection = new Document[0];

   public Result() {}

   public Result(Document[]collection) {
      this.collection = collection;
   }

   public int getCount() {return collection.length;}

   public Document getItem(int i) {return collection[i];}

}
```

The test runs, so we're done with this class.

Notice how we've taken a simple approach to representing the set of documents: an array. This is an example of the XP principle "Do The Simplest Thing That Could Possibly Work" (sometimes abbreviated DTSTTCPW). The current system doesn't require anything fancier than arrays. We "know" the day will come when it requires more, but we won't address it until it does.

Is that really how you do it, or are you just playing along for the example? That's really how you do it.

Query

We can represent the Query as just its query string.

```
public void testSimpleQuery() {
    Query q = new Query("test");
    assertEquals("test", q.getValue());
}
```

Create the Query class with a constructor, so that it remembers its query string and reports it via `getCount()`.

Searcher

The Searcher is the most interesting class. The easy case is first: we should get nothing back from an empty collection of Documents.

```
public void testEmptyCollection() {
    Searcher searcher = new Searcher();
    Result r = searcher.find(new Query("any"));
    assertEquals(0, r.getCount());
}
```

This test doesn't compile, so stub out the Searcher class.

```
public class Searcher {
    public Searcher() {}
    Result find(Query q) {return null;}
}
```

The test compiles, but fails to run correctly (because `find()` returns null). We can fix this with this change: `"public Result find(Query q) {return new Result();}"`.

Things get more interesting when we try real searches. Then we face the issue of where the Searcher gets its documents. We'll begin by passing an array of Documents to the Searcher's constructor. But first, a test.

```
public void testOneElementCollection() {
    Document d = new Document("a", "a word here", "y");
    Searcher searcher = new Searcher(new Document[]{d});
    Query q1 = new Query("word");
    Result r1 = searcher.find(q1);
    assertEquals(1, r1.getCount());

    Query q2 = new Query("notThere");
    Result r2 = searcher.find(q2);
    assertEquals(0, r2.getCount());
}
```

This test shows us that we have to find what *is* there, and not find what's *not* there.

To implement this, we have to provide the new constructor that makes the test compile (though it still fails). Then we have to get serious about implementation.

First, we can see that a search has to retain knowledge of its collection between calls to `find()`, so we'll add a member variable to keep track, and have the constructor remember its argument:

```
Document[] collection = new Document[0];

public Searcher(Document[] docs) {
    this.collection = docs;
}
```

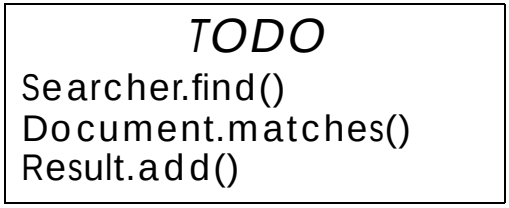
Now, the simplest version of `find()` can iterate through its documents, adding each one that matches the query to a Result:

```
public Result find(Query q) {
    Result result = new Result();
    for (int i = 0; i < collection.count; i++) {
        if (collection[i].matches(q)) {
```

```
            result.add(collection[i]);
        }
    }
    return result;
}
```

This looks good, except for two problems: Document has no `matches()` method, and Result has no `add()` method.

By now, we have several things going on. I often find it helpful to make a "todo" card where I capture a list of things I intend to do:



Let's add a test: we'll check that each field can be matched, and that a document doesn't match queries it shouldn't:

```
public void testDocumentMatchingQuery() {
    Document d = new Document("1a", "t2t", "y3");
    assert(d.matches(new Query("1")));
    assert(d.matches(new Query("2")));
    assert(d.matches(new Query("3")));
    assert(!d.matches(new Query("4")));
}
```

There are three situations for queries that we should deal with eventually: empty queries, partial matches, and case sensitivity. For now, we'll assume empty strings and partial matches should match, and that the search is case-sensitive. In the future we might change our mind.

This is enough information to let us implement `matches()`:

```
public boolean matches(Query q) {
    String query = q.getValue();

    return
       author.indexOf(query) != -1
    || title.indexOf(query) != -1
```

```
    || year.indexOf(query) != -1;
}
```

This will enable `testDocumentMatchingQuery()` to work, but `testOneElementCollection()` will still fail, because Result has no `add()` method yet. So, add a test for the method `Result.add()`:

```
public void testAddingToResult() {
    Document d1 = new Document("a1", "t1", "y1");
    Document d2 = new Document("a2", "t2", "y2");

    Result r = new Result();
    r.add(d1);
    r.add(d2);

    assertEquals(2, r.getCount());
    assert ("First item ",r.getItem(0) == d1);
    assert ("Second item", r.getItem(1) == d2);
}
```

This test fails. Result remembers its list via an array, but that is not the best choice for a structure that needs to change its size. We'll change to use a Vector:

```
Vector collection = new Vector();

public Result(Document[] docs) {
    for (int i = 0; i < docs.length; i++) {
        this.collection.addElement(docs[i]);
    }
}

public int getCount() {return collection.size();}

public Document getItem(int i) {
    return (Document)collection.elementAt(i);
}
```

Make sure the old unit tests `testEmptyResult()` and `testResultWithTwoDocuments()` still pass. Add the new method:

```
public void add(Document d) {
    collection.addElement(d);
}
```

Let's consider the `Result(Document[])` constructor. It was introduced to support the `testResultWithTwoDocuments()` test, because it was the only way to create Results containing documents. Later, we introduced `Result.add()`, which is what the Searcher needs. The array constructor is no longer needed. So, we'll put on a testing hat and revise that test. Instead of

```
Result r = new Result(new Document[]{d1,d2});
```

we'll use:

```
Result r = new Result();
r.add(d1);
r.add(d2);
```

We verify that all tests still pass, so it is now safe to remove the array-based constructor. We also see that `testAddingToResult()` is now essentially a duplicate of `testResultWithTwoDocuments()`, so we'll remove the latter.

At last, all our tests pass for Document, Result, Query, and Searcher.

Initialization

Loading Documents

Where does a searcher get its documents? Currently, you call its constructor from the main routine, passing in an array of documents. Instead, we'd like the searcher to own the process of loading its documents.

We begin with a test. We'll pass in a Reader, and be prepared to see exceptions. We've also postulated a `getCount()` method, used by the tests to verify that something was loaded. An advantage of having the tests in the same package as the class under test is that you can provide non-public methods that let tests view an object's internal state.

```
public void testLoadingSearcher() {
    try {
                    // \t=field, \n=row
        String docs = "a1\tt1\ty1\na2\tt2\ty2";
        StringReader reader = new StringReader(docs);
        Searcher searcher = new Searcher();
        searcher.load(reader);
        assert("Loaded", searcher.getCount() == 2);
    } catch (IOException e) {
        fail ("Loading exception: " + e);
    }
}
```

Notice that Searcher still uses an array (the simplest choice at the time). We'll do as we did for Result, refactor to convert the array to a Vector.

```
package search;
import java.util.*;

public class Searcher {
    Vector collection = new Vector();
    public Searcher() {}

    public Searcher(Document[] docs) {
        for (int i = 0; i < docs.length; i++) {
            collection.addElement(docs[i]);
        }
    }


    public Query makeQuery(String s) {
        return new Query(s);
    }

    public Result find(Query q) {
        Result result = new Result();
        for (int i = 0; i < collection.size(); i++) {
            Document doc = (Document)collection.elementAt(i);
            if (doc.matches(q)) {
                result.add(doc);
            }
        }
        return result;
    }
}
```

(Verify that the old tests pass.) Now we're in a position to do the loading:

```
// Searcher:
public void load(Reader reader) throws IOException {
    BufferedReader in = new BufferedReader(reader);
    try {
        String line = in.readLine();
        while (line != null) {
            collection.addElement(new Document(line));
            line = in.readLine();
        }
    } finally {
        try {in.close();} catch (Exception ignored) {}
    }
}

int getCount() {
    return collection.size();
}


// Document:
public Document(String line) {
    StringTokenizer tokens = new StringTokenizer(line, "\t");
    author = tokens.nextToken();
    title = tokens.nextToken();
    year = tokens.nextToken();
}
```

Searcher's array-based constructor is no longer needed. We'll adjust the test and delete the constructor:

```
public void testOneElementCollection() {
    Searcher searcher = new Searcher();
    try {
        StringReader reader = new StringReader(
                                    "a\ta word here\ty");
        searcher.load(reader);
    } catch (Exception ex) {
        fail ("Couldn't load Searcher: " + ex);
    }

    Query q1 = searcher.makeQuery("word");
    Result r1 = searcher.find(q1);
    assertEquals(1, r1.getCount());
```

```
    Query q2 = searcher.makeQuery("notThere");
    Result r2 = searcher.find(q2);
    assertEquals (0, r2.getCount());
}
```

SearcherFactory

Where does a Searcher come from? Currently, that's left up to whoever calls its constructor. Instead of letting clients depend on the constructor, we'd like to introduce a factory method responsible for locating the Searcher. (For the test, we'll put a file "test.dat" in the directory for testing. If we wanted to be less lazy, we'd have the test create and delete the file as well.)

```
public void testSearcherFactory() {
    try {
        Searcher s = SearcherFactory.get("test.dat");
        assertNotNull (s);
    } catch (Exception ex) {
        fail ("SearcherFactory can't load: " + ex);
    }
}
```

We can implement:

```
public class SearcherFactory {
    public static Searcher get(String filename)
                                    throws IOException {
        FileReader in = new FileReader(filename);
        Searcher s = new Searcher();
        s.load(in);
        return s;
    }
}
```

Now, a client obtains a Searcher by asking a SearcherFactory to give it one.

Looking Back

Put a design hat on, and look at the methods we've developed from two perspectives: the search client and the Searcher class. Who uses each public method?

Search Client	Searcher class
Document.getAuthor() Document.getTitle() Document.getYear() new Query() Result.getCount() Result.getItem() Searcher.find()	new Document() Document.matches() Query.getValue() new Result() Result.add()

Looking at the Document and Query classes, I still have twinges that say they may not be doing enough (each being not much more than a "data bag"). But both seem like good, meaningful "near-domain" classes, so we'll hold off on any impulse to change them. The Result and Searcher classes feel like they have the right balance.

Doesn't this development process generate a lot of blind alleys? For example, we had to change data structures from arrays to vectors (twice!). Is this a flaw in our process? No, it's not a flaw. The array was an adequate structure when it was introduced, and it was changed when necessary. We don't mind blind alleys, as long as they're never one-way dead ends. We are not omniscient, so there will be times we need to change our minds; the key is making sure we never get stuck with a bad or over-complex design.

How many tests should we expect to have? Don't be surprised if your unit test code is anywhere from one to three times as big as the code under test.

Moving Forward: Interfaces

The implementation we've derived above is a good starting point, but is not in final form. In real systems, the bibliographic information is often kept elsewhere, perhaps in a database, XML file, on another network, etc. We don't want our clients to know which alternative they're using.

The methods in the "Search Client" column of the table above show the interfaces required by clients. "Query" is probably OK as a class (since clients have to be able to construct them), but we would like to introduce interfaces for Searcher, Result, and Document. We'll apply the Extract Interface refactoring (from Fowler's book).

Unfortunately, the names we'd like for our interfaces are the same as the ones we already use for the classes. Since we'd like things to be better from the client point of view, and the classes so far are based on strings, we'll rename Searcher to StringSearcher, etc. and reserve the shorter names for the interfaces.

So, move Searcher.java to StringSearcher.java. Fix each call site and reference. Run the tests to verify that we've renamed correctly.

Introduce the interface:

```
public interface Searcher {
    public Result find(Query q);
}
```

(Run the tests.) Make StringSearcher implement the interface. (Run the tests.) Now, the only place that references StringSearcher by name is the SearcherFactory interface. (We could remove that dependency, and perhaps also put the StringXxx objects in a different package, but we won't do that here for reasons of space.)

Apply the same process to Result, renaming the old Result to StringResult, and introducing the interface:

```
public interface Result {
    public Document getItem(int i);
    public int getCount();
}
```

The StringSearcher class should still construct a StringResult object, but its return type should remain Result. (We don't mind if the StringXxx classes depend on each other, but we don't want to make clients aware of that fact.)

Finally, introduce the interface for Document:

```
public interface Document {
    public String getAuthor();
    public String getTitle();
    public String getYear();
}
```

We're left with two concrete classes that clients will depend on: SearcherFactory and Query. Clients depend on the interfaces for Searcher, Result, and Document, but not directly on the classes implementing those interfaces.

Conclusion

We've developed the bibliographic system's model in a typical XP style, emphasizing simple design and a process of alternately testing and coding. The unit tests supported us in designing, coding, and refactoring. The resulting system could have either a simple command line or a graphical user interface attached to it.

References

Kent Beck, *Extreme Programming Explained*, Addison-Wesley, 1999.

Martin Fowler et al., *Refactoring: Improving the Design of Existing Code,* Addison-Wesley, 1999.

William C. Wake, "A Java Perspective," in *Extreme Programming Installed* by Ron Jeffries et al., Addison-Wesley, 2000. A sequel to this chapter.

www.junit.org, JUnit's home.

Sidebar: The Test/Code Cycle for User Interfaces

Not many XP groups use incremental, test-first programming for graphical user interfaces (GUIs), but it's useful to know how to do so.

This sidebar summarizes the approach detailed in my chapter "A Java Perspective," in *Extreme Programming Installed* by Jeffries et al.

- The test/code cycle can apply to many, but not all, aspects of the GUI.
- GUI tests can be robust against many changes in the layout of a screen.
- In Java, widgets such as fields and buttons can be simulated using methods such as `getText()`, `setText()`, and `doClick()`.
- In Java, relative positions can be tested using `getLocationOnScreen()`.
- The GUI can make good use of stubbed-out versions of the model for fine-grained control of tests.

Even with these techniques, you will not be able to unit-test all aspects of a GUI. (In particular, interaction with dialogs and multiple screens is hard.) As in all GUI programming, it is best to drive as much functionality into the model as possible.

Chapter 3

What is refactoring?

"Refactoring: Improving the design of existing code."
—Martin Fowler

"Refactoring" is the process of improving the design of code, without affecting its external behavior. We refactor so that our code is kept as simple as possible, ready for any change that comes along.

See Martin Fowler's book *Refactoring* for a full discussion of the subject.

In this chapter, we'll start with some realistic code, and work our way through several refactorings. Our code will become more clear, better designed, and higher quality.

What do we need for refactoring?

- Our original code
- Unit tests (to assure us we haven't unwittingly changed the code's external behavior)
- A way to identify things to improve
- A catalog of refactorings to apply
- A process to guide us

Original Code

The purpose of the following working code is to generate a web page, by substituting strings for "%CODE%" and "%ALT-CODE%" in a template read from a file. The code was found to create too many string temporaries. (It's been slightly altered for purposes of the example.)

```
import java.io.*;
import java.util.*;

/** Replace %CODE% with requested id, and %ALTCODE% w/"dashed"
version of id.*/
public class CodeReplacer {
    public final String TEMPLATE_DIR = "templatedir";
    String sourceTemplate;
    String code;
    String altcode;

/**
* @param reqId java.lang.String
* @param oStream java.io.OutputStream
* @exception java.io.IOException The exception description.
*/
    public void substitute(String reqId, PrintWriter out)
throws IOException
{
           // Read in the template file
    String templateDir = System.getProperty(TEMPLATE_DIR, "");
    StringBuffer sb = new StringBuffer("");
    try {
      FileReader fr = new FileReader(templateDir +
                                     "template.html");
      BufferedReader br = new BufferedReader(fr);
      String line;
      while(((line=br.readLine())!="") && line!=null)
         sb = new StringBuffer(sb + line + "\n");
      br.close();
      fr.close();
    } catch (Exception e) {
    }
    sourceTemplate = new String(sb);
```

```
    try {
      String template = new String(sourceTemplate);
          // Substitute for %CODE%
      int templateSplitBegin = template.indexOf("%CODE%");
      int templateSplitEnd = templateSplitBegin + 6;
      String templatePartOne = new String(
               template.substring(0, templateSplitBegin));

      String templatePartTwo = new String(
              template.substring(templateSplitEnd,
                                 template.length()));
        code = new String(reqId);
        template = new String(
            templatePartOne+code+templatePartTwo);

            // Substitute for %ALTCODE%
        templateSplitBegin = template.indexOf("%ALTCODE%");
        templateSplitEnd = templateSplitBegin + 9;
        templatePartOne = new String(
            template.substring(0, templateSplitBegin));
        templatePartTwo = new String(
            template.substring(templateSplitEnd,
                               template.length()));
        altcode = code.substring(0,5) + "-" +
                    code.substring(5,8);
        out.print(templatePartOne+altcode+templatePartTwo);
    } catch (Exception e) {
        System.out.println("Error in substitute()");
    }
    out.flush();
    out.close();
}
}
```

Unit Tests

The first step in refactoring is to create unit tests that verify the basic functionality. If you're doing XP, with incremental, test-first programming, those tests exist already as a by-product of that process.

The following test requires a file named `template.html` that contains the text `"xxx%CODE%yyy%ALTCODE%zzz"`.

```
import java.io.*;
import junit.framework.*;

public class CodeReplacerTest extends TestCase {
  CodeReplacer replacer;


  public CodeReplacerTest(String testName){super(testName);}

  protected void setUp() {replacer = new CodeReplacer();}

  public void testTemplateLoadedProperly() {
    try {
      replacer.substitute("ignored,"
                    new PrintWriter(new StringWriter()));
    } catch (Exception ex) {
      fail("No exception expected, but saw:" + ex);
    }

    assertEquals("xxx%CODE%yyy%ALTCODE%zzz\n,"
                 replacer.sourceTemplate);
  }

  public void testSubstitution() {
    StringWriter stringOut = new StringWriter();
    PrintWriter testOut = new PrintWriter (stringOut);
    String trackingId = "01234567";

    try {
      replacer.substitute(trackingId, testOut);
    } catch (IOException ex) {
      fail ("testSubstitution exception - " + ex);
    }

    assertEquals("xxx01234567yyy01234-567zzz\n,"
                 stringOut.toString());
  }
}
```

This code again uses the JUnit unit-testing framework.

Code Smells

Martin Fowler and Kent Beck use the metaphor of "code smells" to describe what you sense when you look at code. Code smells tend to be things that don't indicate that something is necessarily wrong, but they're a "bad sign." You may have heard a similar idea described as "anti-patterns" (after Brown et al.) or "Spidey-sense" (after Stan Lee's Spider-man).

What potential danger signs might you see in code?

- Classes that are too long
- Methods that are too long
- Switch statements (instead of inheritance)
- "Struct" classes (getters and setters but not much functionality)
- Duplicate code
- Almost (but not quite) duplicate code
- Over-dependence on primitive types (instead of introducing a more domain-specific type)
- Useless (or wrong!) comments
- Many more...

Some smells are obvious right away; you may not detect others until you're in the middle of refactoring.

Look at the original code above, and see what problems you can identify. (Don't restrict yourself to this list!)

A Catalog of Refactorings

About half of Martin Fowler's *Refactoring* book is devoted to a catalog of refactorings. Each of these is a relatively simple transformation; Fowler explains the mechanics of the change, and provides examples.

For example:

Extract Method

Before

```
// Assume all are instance
// variables

void f() {
    ...
    // Compute score
    score = a * b + c;
    score -= discount;
}
```

After

```
void f() {
    ...
    computeScore();
}

void computeScore() {
    score = a * b + c;
    score -= discount;
}
```

Another example, not in Fowler's book:

Replace String with StringBuffer

Before

```
String a, b, c;
  :
return a + b + c;
```

After

```
String a, b, c;
  :
StringBuffer sb = new
  StringBuffer(a);
sb.append(b);
sb.append(c);
return sb.toString();
```

This refactoring lets us replace the easy-to-read String version with a more efficient StringBuffer version (or vice versa: many refactorings are appropriate to use "backwards").

Process

Work in an environment that lets you alternate testing and changing your code.

Apply one refactoring, then run the unit tests. Repeat this process until your code expresses its intent clearly, simply, and without duplication. At first, it may feel awkward to run the tests so often, but it will speed you up to do so. (It takes a few seconds to run the tests, but it reassures you that several minutes worth of changes are ok.)

When are we done? When the code:

1. Passes its tests (works)
2. Communicates everything it needs to
3. Has no duplication
4. Has as few classes and methods as possible

These goals are in priority order: if duplication is required to communicate, then the code is duplicated. They are often described as "once and only once" ("once" to work; "only once" to avoid duplication).

Going to Work

When you considered the sample code, what smells did you find? Here is what I saw:

- Long class
- Long method
- Variables could be local to method
- Useless method comment
- Could we read template once, instead of each time?
- The code is tied to using the file system
- Questionable use of "!=" for string compare
- Use of StringBuffer without append
- Re-allocating StringBuffer in loop
- The `close()` methods are not in `catch` or `finally` clause
- Inconsistent / unclear exception handling
- Lots of string addition
- Almost-duplicate code in handling `"%CODE%"` and `"%ALTCODE%"`

- ✧ Lots of extraneous "`new String()`"s
- ✧ Magic numbers (6 and 9) and symbols
- ✧ Lots of temporary variables

The worst smell is that long `substitute()` method, so use Extract Method to break it up. (Note that we capitalize the name of the refactoring used; many of these are available online in the catalog at www.refactoring.com.)

First, pull out `readTemplate()` as a new method:

```
String readTemplate() {
  String templateDir = System.getProperty(TEMPLATE_DIR, "");
  StringBuffer sb = new StringBuffer("");
  try {
    FileReader fr=new FileReader(templateDir+"template.html");
    BufferedReader br = new BufferedReader(fr);
    String line;
    while(((line=br.readLine())!="")&&line!=null)
      sb = new StringBuffer(sb + line + "\n");
    br.close();
    fr.close();
  } catch (Exception e) {
  }
  sourceTemplate = new String(sb);
  return sourceTemplate;
}
```

Even though the change is so simple it could not possibly fail, *run the test!* (And of course, the first time I ran the test, it failed because I forgot to call my new function, highlighting the importance of the mechanics.)

Notice also that we're not immediately chopping the routine into three or four pieces all at once; we're working one step at a time. We'll develop a steady rhythm: change some code, run the test, change some code, run the test. We never go far without verifying what we've done. If we make a mistake, it must be in the last thing we did.

Let's get the template name in one place (via Introduce Explaining Variable, replacing "`templateDir`" with

```
String templateName = System.getProperty(TEMPLATE_DIR,"")
    + "template.html";
```

(*Run the test.*) Then eliminate the "`fr`" variable (Inline Temp):

```
BufferedReader br = null;
    ...
br = new BufferedReader(newFileReader(templateName));
```

(and drop "`fr.close()`".) (*Run the test.*)

Now we're in a position to fix one of the bugs we noticed: the stream is not properly closed in case of errors.

```
try {
  ...
} catch (Exception ex) {
} finally {
  if (br != null) try {br.close();}
      catch (IOException ioe_ignored) {}
}
```

(*Run the test.*)

Next look at another potential problem, the "`!=`" string test. We verify (by looking around and asking around) that the template reader was not intended to stop at blank lines or anything like that, so this condition is meaningless.

```
String line = br.readLine();
while (line != null) {
    sb = new StringBuffer(sb + line + "\n");
    line = br.readLine();
}
```

(*Run the test.*) Martin Fowler points out that it is safer to keep bug-fixing and refactoring separate. He would create a new test case to demonstrate the bug, complete refactoring, and only then go back to fix it. In this small example, we'll just proceed with the fixed code.

Consider the assignment to "`sb`". It is redundant: there's no reason to create a new StringBuffer each time, when we can just add to the one we already have. Also, we can use "`append()`" to eliminate the string addition (Replace String with StringBuffer).

```
    sb.append(line);
    sb.append('\n');
```

(*Run the test.*)

Instead of "`sourceTemplate = new String(sb);`" let's push the work onto the StringBuffer: "`sourceTemplate = sb.toString();`". (*Run the test.*)

The routine both assigns to sourceTemplate, and returns it. Let's move the responsibility for the assignment to the caller, and just "`return sb.toString()`" instead. (Declare the return type as String.) (Reapportion Work Between Caller and Callee.) (*Run the test.*)

For exceptions, let's declare the routine as throwing IOException, and delete the empty catch clause; the caller will have to deal with any exceptions. This causes an (untested! hmm...) change in behavior as no partial template will be returned in case of error; we'll confirm by looking and asking that this is OK. (Run the test.)

Our routine now looks like this:

```
String readTemplate() throws IOException {
    String templateName = System.getProperty(TEMPLATE_DIR, "")
                          + "template.html";
    StringBuffer sb = new StringBuffer("");
    BufferedReader br = null;
    try {
        br = new BufferedReader(new FileReader(templateName));
        String line = br.readLine();
        while (line != null) {
            sb.append(line);
            sb.append('\n');
            line = br.readLine();
        }
    } finally {
        if (br != null) try {br.close();}
                        catch (IOException ioe_ignored) {}
    }
    return sb.toString();
}
```

The next thing I don't like is that the routine both decides where to find the template and how to read it, leaving it coupled to the file system. This may be a problem in the future (if templates were to come from somewhere else), but it's also a problem now: our test has to use an external file. I'm not sure of the right approach, so we'll defer this problem.

Substitute for %CODE%

The routine is still too long. We also still have the near-duplicate code for substitutions.

So, next we'll Extract Method for replacing "`%CODE%`".

```
String substituteForCode(String template, String reqId) {
  int templateSplitBegin = template.indexOf("%CODE%");
  int templateSplitEnd = templateSplitBegin + 6;
  String templatePartOne = new String(
    template.substring(0, templateSplitBegin));
  String templatePartTwo = new String(
     template.substring(templateSplitEnd, template.length()));
  code = new String(reqId);
  template = new String(templatePartOne+code+templatePartTwo);
  return template;
}
```

Adjust the variable declarations left in `substitute()`. (Run the test.)

The first thing I notice is the string "`%CODE%`" and the value "`6`" (the length of the pattern). Pull out the pattern and use it: (Replace Magic Number with Calculation)

```
  String pattern = "%CODE%";
  int templateSplitBegin = template.indexOf(pattern);
  int templateSplitEnd = templateSplitBegin+pattern.length();
```

(*Run the test.*)

We create a lot of new Strings too: all those "`new String()`" constructions are redundant, since their arguments are Strings already. (Remove Redundant Constructor Calls)

```
String templatePartOne =
      template.substring(0,templateSplitBegin);
  String templatePartTwo =
      template.substring(templateSplitEnd, template.length());
  code = reqId;
  return templatePartOne + code + templatePartTwo;
```

(*Run the test.*)

We'll eventually want to address the remaining string addition (on the return statement), but let's take care of "`%ALTCODE%`" first.

Substitute for %ALTCODE%

Let's do the same Extract Method and simplification for the other case, and see where we are:

```
void substituteForAltcode(String template, String code,
                          PrintWriter out) {
  String pattern = "%ALTCODE%";
  int templateSplitBegin = template.indexOf(pattern);
  int templateSplitEnd = templateSplitBegin+pattern.length();

  String templatePartOne = template.substring(
               0, templateSplitBegin);
  String templatePartTwo = template.substring(
              templateSplitEnd, template.length());
  altcode = code.substring(0,5) + "-" + code.substring(5,8);
  out.print(templatePartOne + altcode + templatePartTwo);
  }
```

This code is *so* similar to `substituteForCode()`, it's clear we should be able to unify the two routines. But there are three differences: they look for different patterns, they substitute different values, and they write to different streams. Drive them toward duplication by passing in the patterns and replacements as arguments: (Parameterize Method)

```
void substituteForAltCode(String template, String pattern,
String replacement, PrintWriter out) {
  int templateSplitBegin = template.indexOf(pattern);

  int templateSplitEnd = templateSplitBegin+pattern.length();
```

```
  String templatePartOne = template.substring(
        0, templateSplitBegin);
  String templatePartTwo = template.substring(
        templateSplitEnd, template.length());
  out.print(templatePartOne + replacement + templatePartTwo);
}
        ...
  altcode = reqId.substring(0,5) + "-" + reqId.substring(5,8);
    substituteForAltCode(template, "%ALTCODE%", altcode, out);
```

(*Run the test.*)

We can address the excessive string addition by using a series of `print()` calls (Replace String Addition with Output), and we'll pull the buffer flush up as well:

```
    out.print(templatePartOne);
    out.print(replacement);
    out.print(templatePartTwo);
    out.flush();
```

(*Run the test.*)

The big difference now is that "`%CODE%`" results in a String, and "`%ALTCODE%`" is written to a PrintWriter. These can be reconciled via the class `java.io.StringWriter`. So, make `substituteForCode()` take an argument "`PrintWriter out`," and create and pass in a "`new PrintWriter(new StringWriter())`" as its stream. (Unify String and I/O). (*Run the test.*)

How did I even know to look for a class such as StringWriter? First, I've used enough systems to know that many have a way to let you treat strings as I/O, and vice versa. Second, when you learn a new language it's worthwhile to read once through the core APIs. Third, it *is* documented. Fourth, I could have talked to a co-worker and found it; there's usually someone who knows the odd corners you don't.

The two routines are now identical: eliminate `substituteForAltcode()`, and just call `substituteForCode()` twice. (Merge Identical Routines) (*Run the test.*)

Back to `readTemplate()`

We were able to verify (by asking our customer) that it would be acceptable to read the template once at startup, rather than once per call to `substitute()`. We can declare the constructor to throw IOException, and make the call to `readTemplate()` there:

```
sourceTemplate = readTemplate(
        System.getProperty(TEMPLATE_DIR, ""));
```

(*Run the test.*) This helps eliminate some strings we would otherwise create.

At last, we can address that old thorn: direct reading of a file to load the template. The `readTemplate()` routine currently takes a directory name and constructs a file. Instead, we'll pass it a Reader and let that do the work. First, pull up construction of the FileReader into the constructor: (Reapportion Work Between Caller and Callee)

```
public CodeReplacer() throws IOException {
    String templateName =
System.getProperty(TEMPLATE_DIR, "") + "template.html";
    sourceTemplate = readTemplate(
                          new FileReader(templateName));
}

public String readTemplate(Reader reader)
                      throws IOException {
    ...
}
```

(*Run the test.*)

Next, introduce a constructor that takes a Reader, which the caller is responsible for forming. (They will probably use the `getProperty()` code that was there before.)

```
public CodeReplacer(Reader reader) throws IOException {
  sourceTemplate = readTemplate(reader);
}
```

(*Run the test.*)

I'll put a testing hat back on, and modify my test. We can simplify `testTemplateLoadedProperly()`, as we no longer need to do a substitution to see the template just to load it. Also, instead of setting up with the file "`template.html`," we'll test using a StringReader. This helps decouple our tests from the environment.

```
final static String templateContents =
                      "xxx%CODE%yyy%ALTCODE%zzz\n";
        ...
  replacer = new CodeReplacer(new StringReader(
                               templateContents));
        ...
public void testTemplateLoadedProperly() {
  assertEquals(templateContents, replacer.sourceTemplate);
}
```

(*Run the test one more time.*)

Final Result

Here's the complete new version of CodeReplacer.java:

```
import java.io.*;
import java.util.*;
/** Replace %CODE% with requested id, and %ALTCODE% w/"dashed"
version of id.*/

public class CodeReplacer {
    String sourceTemplate;

    public CodeReplacer(Reader reader) throws IOException {
        sourceTemplate = readTemplate(reader);
    }

    String readTemplate(Reader reader) throws IOException {
        BufferedReader br = new BufferedReader(reader);
        StringBuffer sb = new StringBuffer();
        try {
            String line = br.readLine();
            while (line!=null) {
```

```
                sb.append(line);
                sb.append("\n");
                line = br.readLine();
            }
        } finally {
            try {if (br != null) br.close();}
           catch (IOException ioe_ignored) {}
        }
        return sb.toString();
    }

    void substituteCode (String template, String pattern,
                         String replacement, Writer out)
                           throws IOException {
        int templateSplitBegin = template.indexOf(pattern);
        int templateSplitEnd =
             templateSplitBegin + pattern.length();
        out.write(template.substring(0, templateSplitBegin));
        out.write(replacement);
        out.write(template.substring(
                   templateSplitEnd, template.length()));
        out.flush();
    }

    public void substitute(String reqId, PrintWriter out)
                    throws IOException {
        StringWriter templateOut = new StringWriter();
        substituteCode(sourceTemplate, "%CODE%,"
                       reqId, templateOut);

        String altId = reqId.substring(0,5) + "-" +
                          reqId.substring(5,8);
        substituteCode(templateOut.toString(), "%ALTCODE%,"
                       altId, out);
       out.close();
    }
}
```

Here's CodeReplacerTest.java:

```
import java.io.*;
import junit.framework.*;

public class CodeReplacerTest extends TestCase {
    final static String templateContents =
                    "xxx%CODE%yyy%ALTCODE%zzz\n";
```

```
    CodeReplacer replacer;

    public CodeReplacerTest(String testName)
{super(testName);}

    protected void setUp() {
        try {
            replacer = new CodeReplacer(new StringReader(
                                        templateContents));
        } catch (Exception ex) {
            fail("CodeReplacer couldn't load");
        }
    }

    public void testTemplateLoadedProperly() {
        assertEquals(templateContents,
                     replacer.sourceTemplate);
    }

    public void testSubstitution() {
        StringWriter stringOut = new StringWriter();
        PrintWriter testOut = new PrintWriter (stringOut);
        String trackingId = "01234567";

        try {
            replacer.substitute(trackingId, testOut);
        } catch (IOException ex) {
            fail ("testSubstitution exception - " + ex);
        }

        assertEquals("xxx01234567yyy01234-567zzz\n,"
                     stringOut.toString());
    }

}
```

Analysis

We now have a much better routine. We've fixed a few bugs and ambiguities on the way. It's clearly much easier to read, as we've squeezed out the duplicate code. We replaced string addition with cheaper operations in a number of places. We've iso-

lated our templates from the file system; instead they use Readers to load themselves.

The new code embodies three basic things:

1. How a template is represented (currently a string)
2. How substitution is made
3. What codes and values to substitute (`%CODE%` and `%ALTCODE%`)

The first two items are part of the nature of how a template works; the third is the "business logic" that tells how it's used. A stand-alone Template class would address the first two points. That was not obvious from the monster routine we started with. This shows how our sense of the code smells can change over time: I'd now argue that an over-dependence on the String type keeps us working at too low a level.

Pulling the template into a separate class would allow us to address performance even more, as we would be free to change the representation of templates. The current implementation scans the template (twice) to locate the patterns that need substitution. We might be able to pre-process a template to identify the potential substitutions. We could change the interface to the substitution process to take a list (or perhaps a Hashtable) of substitutions, so we could do them all in one pass.

Conclusion

What have we seen?

- ✧ We can make very small, incremental improvements to the code. At each point, the latest version is better than the one before. We didn't have to take the risk inherent in "Just scrap it and start over."
- ✧ The unit tests acted as a constant safety net. We never go more than a few minutes without the reassurance they provide.
- ✧ Some improvements allow further improvements. The need for a template class was far less obvious in the initial code.

- While it's possible that some refactorings may interfere with performance, they will often open up possibilities for dramatic improvements. Extracting the `readTemplate()` method allowed us to notice it was called for every substitution when it only needed to be called once; a future version of the Template class might be able to do all substitutions in one pass. These possibilities are a much bigger factor than the "extra" procedure call that Extract Method originally introduced.
- In a handful of moves (about 20), we've substantially improved our code.

Make refactoring (including testing!) a part of your normal programming practice. Sensitize yourself to code smells, and learn refactorings that can address them. It will pay off in the simplicity and flexibility your system will have.

Resources

Kent Beck, *Extreme Programming Explained*, Addison-Wesley, 1999.

Jon L. Bentley, *Writing Efficient Programs,* Prentice Hall, 1982.

Jon L. Bentley, *Programming Pearls 2/e* (1999), and *More Programming Pearls* (1988), Addison-Wesley.

William J. Brown et al., *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis,* John Wiley & Sons, 1998.

Martin Fowler et al., *Refactoring: Improving the Design of Existing Code,* Addison-Wesley, 1999. A catalog of code smells and refactorings.

Martin Fowler, www.refactoring.com. Online catalog.

Stan Lee (editor), *The Ultimate Spider-Man,* Boulevard, 1996.

www.junit.org, JUnit home.

Section 2: Team Practices

Chapter 4

What are XP's team practices?

We'll explore these practices and their alternatives.

Code Ownership: Who can change an object that needs to change?

Integration: How and when does a team ensure everybody's code works together?

Overtime: What do you do when you run out of time?

Workspace: How should a team be physically arranged?

Release Schedule: How often should a team release their product?

Coding Standard: How should the code be written?

These are "team" practices because it's no use trying to do them alone; it does no good to be the only person integrating often or the only one following a coding standard.

Code Ownership

When code needs to be changed, who gets to (or has to!) change it?

Nobody ("Orphan")

In some groups, nobody owns the code. For closed-source systems, maintained by a third party, "nobody" just means "nobody here." There are other programs that are untouchables: there's no source code (it was lost years ago), or the system is too brittle or too complicated to safely change.

When code isn't owned, developers will either design around it or treat it as a black box. That latter approach can be especially painful: the programmer has to format data going in and out; they may have to mediate between two paradigms (e.g., using screen-scraping technology); they may be unable to do things you would normally expect to be possible; there may be data denormalization and coordination problems because information has to be stored both inside and outside the system.

Avoid the situation of "nobody owns it (and nobody wants to)."

Last One Who Touched It ("Tag" or "Musical Chairs"); and Whoever's Newest ("Trial by Fire")

You may have systems that are reasonably stable, but still a pain to maintain. In these situations, there's typically a local cultural rule that kicks in: whoever touched it last owns the next problem (which can make it frustratingly hard to escape), or perhaps the problem is given to the least powerful person (often with the instructions, "just make it work").

This model seems very common in maintenance organizations.

One Owner Per Class (or Package) ("Monogamy")

This approach is very common in new development: the author owns the code they write, until somebody explicitly takes it over.

The key is that there's a well-defined person in charge of a section of code. If you want it changed, you negotiate with them. They may even let you change it yourself, but they'll retain approval rights on what you do. Sometimes the ownership will go stale: the person who did one part is gone, and it's not actively changing, so there won't be a real owner until someone needs something done, the team will assign an owner when it needs to.

The benefit of this approach is that it provides a clear method for deciding who does what work, and it lets a person develop expertise in a particular area.

There are downsides, though. If the owner isn't available to make a change when needed, the team is slowed down. (Some teams mitigate this by having emergency backups.) Sometimes the owner won't agree with a proposed change, perhaps completely vetoing it. Then clients may design around it or wrapper it in a way that is worse for the system overall. Another problem is that a single owner can be in trouble and hide the fact. (The code may be deeply in trouble by the time their lack of expertise is noticed.)

The single-owner approach can require that interfaces be frozen too early, before they're really understood, because of the need to let clients "make progress." Refactoring can be more difficult; when interfaces are public and require a lot of cooperation to change, there is political pressure not to change them (even if they need it).

Sequential Owners ("Rental Property" or "Serial Monogamy")

Some groups have a single owner for a package at any given time, but that owner will be decided on a task-by-task or iteration-by-iteration basis.

This can be good in that it gives people variety, but they may not get a chance to build deep enough expertise to be particularly effective. But at least there's an owner, so everybody knows who to talk to.

The biggest problem is that the code is like rental property: the current "owner" is really a "tenant"; they know the situation is temporary, so they have less incentive to worry about the long-term value of the place (provided they can get out of their lease before it gets too bad).

Ownership by Layer ("Tribalism")

Some groups organize their software into distinct layers, or even distinct applications, and have ownership at that level. Anybody in the tribe can make any change they need to—but they keep the tribe informed about what they're doing. However, nobody from another layer would dare to jump in. (You see this with user interface vs. business logic vs. database, or application 1 vs. application 2 vs. database.)

This method overcomes some flaws of "monogamy": the "bus number" is higher, there's no chance bad code will be hidden from the sub-team, etc. (The "bus number" is the number of people that, if they were hit by a bus, would cause the project to fail.) The subteams can develop a very strong esprit de corps, which further boosts their productivity.

However, interfaces *between* layers are elevated to near "golden" status, making them even harder to change. This interferes with refactoring intended to improve the overall system design.

Communication within the subteam is usually good, but it's harder to talk across layers. A change that requires convincing one person in the "one owner" model now requires "How about our subteams meet and discuss this; is everybody open next week?" This slows down the team as whole.

Collective Code Ownership ("Everybody")

In collective ownership, the whole team owns all code. Anybody can change any part they need to.

The key benefit of this approach is that it is the least obstructive to getting things done quickly. The ability to refactor is

improved, as there are fewer published interfaces. For the right team, this can be an effective mode. (See the next section, which describes how XP tries to retain its benefits without falling victim to its risks.)

There are potential downsides as well:

- Some people take personal pride in their code, and don't want others to touch it.
- You risk the tragedy of the commons: "everybody is responsible" can come to mean "nobody is responsible."
- This resembles "musical chairs" carried even further, perhaps aggravating the risk that nobody will be expert enough, and that nobody will care for the long-term value.
- It can be hard to read and work with someone else's code. You may get a mish-mash of styles and approaches.
- People may step on each others' toes more, as they need access to the same objects.

XP Uses Collective Code Ownership

Extreme Programming recognizes the risks in collective code ownership, but uses it anyway. XP's practices attempt to mitigate these risks:

- Personal pride: XP doesn't really address this, other than perhaps encouraging a shift to pride at the team level.
- "Tragedy of the Commons": Pair programming, and unit tests running at 100%, help ensure that no person can "pollute" in secrecy. The shuffling of pairs helps make all code visible to the whole team. Refactoring can clean up any trouble that does occur.
- Not enough expertise: Pair programming spreads knowledge through the team. The open workspace gives others a chance to speak up when one pair seems stuck or worried. Simple design can require less "deep" expertise. Finally, tests help ensure that functionality won't diverge from the requirements.
- Reading others' code: A coding standard helps reduce this problem, as does the shared culture that pair programming engenders.

- ✧ Stepping on toes: Continuous integration ensures that people won't go far without rejoining the main line. The tests ensure that there is no regression.

Integration

How and when does a team ensure everybody's code works together?

Just before delivery

Some groups I've worked with have typically had developers working in private areas, picking up what they need from others as they need it. Before delivery, there's an attempt to do a "code freeze" (usually a "code slush" at that point): everybody makes sure everything is checked in, and they resolve any conflicts due to changing interfaces (especially those that obviously break the compile). (This is similar to annealing in metallurgy: at first there's a lot of activity, and as the temperature is lowered, everything settles into a low energy state.)

The big problem with this approach is that it lets people go off in drastically wrong directions, with nothing forcing them to test (or even integrate) for days or weeks. The integration process becomes a lot of work for whoever must resolve all the problems.

Daily Builds

One way forward is to go to daily builds. Every night, the system is compiled, and a "smoke test" is run.

The developer's motto becomes "Don't break the build." Developers are supposed to do their own integration testing as they check in, so there should be no surprises. (The groups I've been in with this process had a project manager who actively checked that people were checking in as they finished tasks.)

Groups evolve different mechanisms to deal with the code breaking. I've heard of groups that say "Whoever breaks the build must be in by 8:00 each day to check the latest build (until somebody new breaks it)." One group I was in relied on peer

pressure by publicizing who broke the build. Eventually the team hired a manager who had a set start time each day, and whose first responsibility of the day was to identify any compilation or integration problems, and get the relevant programmer(s) to fix them.

Continuous Integration

Continuous integration, as XP uses, is not literally continuous, but it does occur several times per day. Each time a developer (pair) finishes a work session, they integrate what they have done. Typically, there's a single machine for the team's integration efforts, so it's serialized "first-come, first-serve."

Integration in XP is supported by tests. Developers must keep all unit tests running at 100%. To integrate, they take their code to a machine already running at 100%. After they've integrated: if the tests are still at 100%, they're done. If not, only one thing has changed: the code they introduced. They're obligated to fix the code (perhaps with help from others), or back out the changes and throw them away. They're *not* permitted to leave the integration system in a deteriorated state.

Continuous integration is possible in XP because of the "togetherness" of the group, because it's supported by tests, and because XP provides for simpler design via refactoring.

Overtime

What do you do when you run out of time?

Work overtime

For many teams, overtime is their first response to a schedule crunch. First, the team's hours start to stretch, then the team is asked to work weekends. Some groups go so far as to have mandatory overtime.

This path can be counter-productive. The job can turn into a "death march" (in Yourdon's phrase). People find themselves

working at less capacity because they're too tired to think straight. Or, it takes a toll on their family life, or they put in the hours physically but not mentally.

40-Hour Week

XP pushes a different view, saying that a 40-hour week is more appropriate (for some value of 40). XP teams realize there may be an occasional long week, but that even two weeks of overtime in a row is a sign of other problems. What does "40" mean? It doesn't mean exactly 40; for some people it will be 35, for others 45, etc. "40" puts a number out, to say, "there is a limit."

XP teams want a sustainable level of productivity. If the team can't do all they promised in an iteration, they hand back stories. The "Yesterday's Weather" rule ensures that next iteration is better balanced to the team's productivity level. ("Yesterday's Weather" says to estimate that next iteration will accomplish about as many days work as this one did.) If a team is doing 2- or 3-week iterations, and it requires overtime to finish the promised tasks for this iteration, odds are good it would take overtime next iteration as well. It's better to figure out the team's natural pace.

My manager, Steve Metsker, says, "A professional should be willing to spend five hours a week (outside of work) improving themselves." There are XP teams that reserve half a day per week as "play time" (where members can do things like learn a new programming language). This uncommitted time pays off in the main work week.

In *Planning Extreme Programming*, Kent Beck has a great story about a team that moved from thinking "We don't have enough time" to "We have too much to do." He points out that the shift is empowering: you can't do anything about time, but you*can* do something about your tasks.

Workspaces

How should a team be organized physically?

Geographically Separate (Including Telecommuting)

Geographically separate groups are based in different locations, and face the communication difficulties that implies. For example, there may be differences in language (in the worst case) and differences in time zone. Even groups nominally in the same time zone can evolve different hours: I worked with a group where one site tended to get in between 8:00 and 9:00, and the other group around 10-10:30. With varying lunch and departure schedules, the core hours were 10:30-11:30 and 1:30-4:00.

Communication problems can make this approach more costly, even though it appears cheaper on the surface. ("We can hire the best people from both locations," "We'll start another site where it's cheaper to hire.") There are travel costs as well as communication costs (including resentment from those who travel). Most groups must co-locate once in a while. I know of a group with members located on three continents; they felt that things fell apart if they didn't meet in the same room at least quarterly.

Telecommuting takes the case of separate workspaces to the extreme, breeding more communication difficulty, more isolation, and a perception of less opportunity for advancement. If tasks can be split in a way to minimize the need for communication, telecommuting may be a welcome approach. I haven't known any developers to stick with it for more than a year or so, though.

Offices (1- or 2-Person)

Offices offer the greatest level of privacy and quiet to the developer. In *Peopleware*, DeMarco and Lister reported that those with offices did the best in their "coding wars," which they attributed to the need for programmers to reach a state of "flow."

Offices are apparently the most expensive choice, because few companies seem willing to provide them for developers. (I've found a 2-person office to be far more productive than a cubicle.)

Cubicles

"Cubes" seem to be the form of office least liked by developers and most liked by the managers who put them there.

Companies I've worked with seem to be making them smaller: ten years ago, the standard size was 100 square feet (including for the president of the company); today (at a smaller, different company), the standard is 54 square feet (and I'm bigger than I was then!). My current cubicle doesn't allow for two people at the desk, and even if it did, the computer doesn't fit anywhere but the corner.

Cubes also seem to get smaller only, never bigger. A friend of mine worked for a company that would have occasional "moving weekends": you'd pack up all your stuff, and return Monday to the same office only 6" smaller. By gaining 6" per cube across a football-field-sized room, they could add even more cubicles.

Cubes don't allow much privacy or quiet, but they still interfere with communication. (I've seen wall heights anywhere from 4 to 8 feet, but they always seem to be open on top and never have a door.)

Open Workspace

XP specifies an open workspace, usually with small private spaces around a central area. The fast machines are in the center, for pairing. XP would like the team to hit a state where programmers can focus on their problem, but still hear enough to jump in if they can help.

The same physical structure without an XP focus can degenerate into a "bullpen": everybody is stuffed into the same space, with no private space at all. You aren't on the same project or phase, so the rhythms of your communication clash with those of others. The bullpen has been the worst office type I've been in.

Some groups starting toward XP also take smaller steps toward open workspace, by arranging for pairing machines in a spare niche, or treating a wing of cubes as a public area. Beware that

some places have "hot" walls: the cubicle walls are dangerous to move as they're wired for electricity.

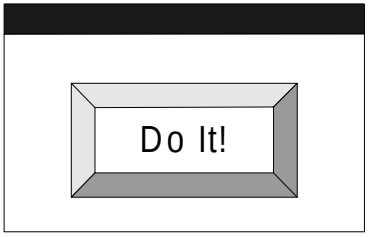
Release Schedule: Small Releases

How often should a team release their product? XP pushes for small but frequent releases. An XP team is trying to learn, and the more feedback they get (especially from actual use), the better. Projects that don't release for months, or even years, accumulate a lot of risk: technology will change, the business environment will change, the team will change. Small releases reduce this risk.

A release should be small, but it needs to make sense: only whole features should be included. The first release needs at least primitive versions of all major components or subsystems.

Failing to meet the goal of having small releases is the biggest mistake I've made in developing systems. When I focus on it, I find that "small" can be even smaller than I would have thought.

I've resolved to start every project with this question: "Will this interface work?" (Mimic pushing the button with your nose or forehead.)



The answer is often "yes, if you knew these parameters." In that case we can start without a GUI. Where this interface isn't enough, it can get us focused on understanding what is the minimal useful system. (I've also found that it can be the developers who resist starting small; they have the vision of what the system could be and hate to back off from there.)

In the first XP immersion class, Michael Hill described his practice of producing a ZFR ("ziffer"): Zero Feature Release. It was an end-to-end release that *did* nothing, but it established the architecture and the deployment strategy in the first iteration. It's hard to get a system into production the first time; updating it is usually a lot easier.

Coding Standard

A coding standard improves people's ability to read each other's code. In XP, it supports refactoring as well: consistently "shaped" code can be refactored more confidently.

There are several approaches to a team's coding standard:

- None. Ugh.
- Each programmer chooses their own style and uses it everywhere. ("Then I can tell where I added code.") Ugh again.
- Each programmer chooses their own style for their own code, but uses the existing style on existing code. This is the minimum I can live with. I've worked with many teams that intended to have a standard, but evolved toward this as there was no "enforcement" pressure. At least with this, you can read an individual module. This approach does interfere with refactoring: when you move code between modules, you have to reformat it as well.
- The team has a style and sticks to it. This is the ideal for XP. A team coding standard supports the XP practices of collective code ownership, pair programming, constant refactoring, and continuous integration.

I've seen two group styles in determining their standard. The first type group has a huge battle at the beginning of the project with heated arguments over spaces and braces, but converges on a group style. The second group has no fight, but builds no group ethic either; they end up with "whatever's right for the code you're in." I'm not sure how to lead the second type along.

What coding style should be used? In a recent project where I participated in defining this (for Java) we had a 6-page summary

of Sun's Java coding conventions, extended with the JavaBean naming conventions.

Now, I use a one-page standard. Perhaps even that wouldn't be needed with the peer pressure of constant pairing and refactoring.

Conclusion

We've discussed several team practices of XP, and some of the alternative choices possible.

Code Ownership: Who can change an object that needs to change? XP says "collective code ownership." A non-XP team may find this to be problematic.

Integration: How and when does a team ensure everybody's code works together? XP says "continuous integration." A non-XP team should strive toward as frequent an integration as possible.

Overtime: What do you do when you run out of time? XP says "maintain a 40-hour week." A non-XP team should strive for this as well.

Workspace: How should a team be physically arranged? XP says "open workspace." If a team is not XP, I favor offices, but with attention to the team's communication paths.

Release Schedule: How often should a team release their product? XP says "small releases." If a team is not XP, I think this is still a useful target, but probably harder to hit.

Coding Standard: How should the code be written? XP says "team must share a standard." If a team is not XP, I think a team standard is still appropriate.

Resources

Kent Beck, *Extreme Programming Explained*, Addison-Wesley, 1999.

Kent Beck and Martin Fowler, *Planning Extreme Programming*, Addison-Wesley, 2000.

Alistair Cockburn, *Surviving Object-Oriented Projects*, Addison-Wesley, 1998. Discusses various models; recommends "Owner per Deliverable."

Jim Coplien, "Code Ownership";
http://www1.bell-labs.com/user/cope/Patterns/Process/section18.html.

Tom DeMarco and Timothy Lister, *Peopleware*, Dorset House, 1987.

Ron Jeffries, "Code Ownership";
http://www.xprogramming.com/Practices/PracOwnership.html.

Sun, "Sun's Java Coding Conventions," http://java.sun.com/docs/codeconf/

Sun, "JavaBeans Conventions," http://java.sun.com/beans/docs/beans.101.pdf

Sun, "JavaDoc conventions," http://java.sun.com/products/jdk/javadoc/writingdoccomments/

Edward Yourdon, *Death March*, Prentice Hall, 1999.

Chapter 5

What's it like to program in pairs?

Pair programming is exhausting but productive.

Pair programming is the practice of having two people working together on all production code. They do this as full partners, taking turns typing and watching, to provide constant design and code review.

Some Counter-Examples

Solo

Bob is feeling great—he got in at 6:30 AM today. He picks up the next task card, does a quick design, and starts writing tests and code. In a couple hours, he integrates his code to the main system, and all tests pass on the first try, just as the rest of the team shows up for work that day.

Bob may be a great programmer, but as you might expect, this is not an example of pair programming. Even a team that's fully committed to pairing will occasionally face the temptation to skip it "just this once." Steve Metsker of Capital One says, "If it doesn't pinch sometimes, it's not a real methodology."

Solo programming can cause several problems. In the example, Bob may not be having as good a day as he thought; he may be passing tests, but not refactoring enough, or vice versa. The design may work but not be as simple as it could be. Also, we've avoided some potential cross-team learning—Bob is the only person familiar with the code he created.

Crowd

Janna picks her task card and asks Jay for help. Jay says, "Hey everybody—we're hooking up the joystick to the spaceship—come see!" The whole team (all ten) crowd around to watch and make suggestions. In an hour (or so), everything is working. Everybody takes a turn playing with it, and they all agree it's pretty cool.

Again, this may be good programming (and it may be fun) but it's not pair programming. The code may be fine, but it cost five times as much as it should have, and other work was delayed meanwhile. This is not to say that all pairs and all partners will always be able to handle all problems, but it ought to be rare to require the whole team for an extended time. The celebration at the end may be fine—a team needs to recognize its successes—but let's recognize it for what it is.

Disengaged

Pat asks Ann to be her partner for the morning. Pat starts in, but Ann just sits behind her, arms and legs folded, physically present but not paying any attention. Her attitude projects, "You can make me sit here, but you can't make me help."

It takes more than physical presence to be a partner; it takes engagement in the task. In this case, if Ann is consistently a problem partner, the team will need to work to bring Ann into alignment with the team's values.

True Pairing

Don asks Lee to be his partner on the next task. Here's the card:

> ### *Task: Assign Usernames*
>
> Assign a unique username to each person in the system. Usernames need to be 6-14 characters long, and must start with a letter. If it hasn't been taken already, the username should be the first initial plus full last name.

Don and Lee talk to Carol, the customer, and find out a little more. This is a one-shot update; new entries will come with a username. Also, usernames should only have lower-case letters and digits.

Don says, "Let's start with the simplest test case: ignore the problem of duplicates, and generate 'bsmith' for 'bob smith'". He starts to write code for the test, and says, "What are the parameters to `substring()`?" While he does the mechanics of the code, Lee looks up the method, and says, "It's one-based, like this..." and explains.

Don is focused on the typing, so Lee is free to support the task at a higher level. Instead of making a guess about the method, they've checked the real description, and avoided a potential bug.

Don says, "It gets harder when we have the second 'bob smith'. We'll need to keep track of the names somehow—we can generate them in order, I guess, and add a number field to put on the end. Let's assume they call our routine with names in order. Let me add a variable for lastBaseName, and then..." Lee interjects, "Wait—we need to write the test first!" Bob blushes and says, "Oh, right."

The partner can help make sure team values aren't ignored.

Don starts again, "Hey, Carol! If you have two 'Bob Smith's do you want the second one to be 'bsmith1' or 'bsmith2'?" Carol says, "I guess 'bsmith2', unless that's a lot harder." Don finishes the test. He makes a typo, but Lee holds off a second before pointing it out, and sure enough he fixes it before she says anything.

We see the value of an on-site customer. Again, the partner has focused on potential quality problems. Notice also the holding back; if Don had missed the typo, Lee would have spoken up, but she didn't want to interrupt his flow unnecessarily.

Don gets the code going and the test working. Lee says, "We still need to cover the case of a name that's too short. Can I drive?" Don passes the keyboard, and she creates a test for "ye wu". After a quick consult with Carol, they decide this should be "yewu01". Lee gets the code working, and Don says, "We also have to worry about mixed case." Lee says, "Do you want to drive?"

The partner thinks strategically, making sure key cases aren't omitted. Also notice the protocol for passing the keyboard: "Can

I drive?" With practice, pairs will develop a sense of knowing when to switch.

Lee says, "We should refactor to make all this work together cleanly." They move code around until it's clear and concise. Bob then easily drops in the new feature.

Lee says, "I've been thinking. We really have three problems: the real name, the username, and the mapping between them. The real name could have many problems: case, length, and so on. Do we understand all these problems? The username has its restrictions too: length, case, and use of digits. Finally, we need to know the mapping for each type of name. Let's sit with Carol again."

More strategic thinking, and learning as well. They've learned more about the problem structure from what they've done so far.

They snag Carol for a real sit-down. They have a good handle on the requirements for the username, but names have a lot of subtlety. Carol says, "I've been looking into it this morning, and our data is actually pretty dirty. You might see any of these:

- |Mary and Bob Smith
- Bob|D'Santos
- _|John Smith
- Socrates|
- |Plato
- Aristotle|A. Smith
- |JOHN SMITH
- |John Smith
- John|Smith
- |'()$#"@
- |(none)

and so on.

You might see short or long names:

- Wu|Wei
- Archimides|Apollonniganymedea

and you might see some tough duplicates:

- John|Roberstoniton (20 of these, followed by)
- John|Robertsonitony

You'd have to be careful not to assign two "jrobertsoniton"s.

The good news is: we're really not all that picky about what names you assign. If people really hate their username, they can call and we'll change it."

Lee asks, "What's the worst case for number of duplicates?" Carol says, "If you assume 99 or fewer, that's OK except for one crucial case: the dirty data with no alphabetic characters. There are a lot of those. You can assign them something like 'x00001', or anything really."

This is not the best news, since they've already started working, but it's better to find out now than later.

Don and Lee go back to their code. Don says, "It looks like we're pretty far down the wrong track; should we toss this code?" Lee says, "Yup, it's the right thing to do."

The partner gives you permission to do what's needed. Sometimes, that means tossing code while retaining the learning, and starting over.

They agree on a new strategy: Use a Unix pipeline:

- ✧ write out id #, first, and last name
- ✧ shift the name to lower-case and delete anything not a letter
- ✧ use an 'x' for the first and last name if they're empty
- ✧ trim to at most 12 characters
- ✧ add enough 0's to make it at least 6 characters
- ✧ sort
- ✧ build and write the names. If a name duplicates the previous one, append the counter.

They set up some test cases:

1|John|Smith
2|John|Smith
3|Wu|Wei
4||
5|(504)|555-1234

etc.

Don: "Can we *still* get a collision?" Lee: "I don't think so, but if we do, we'll just report an error, and do those few by hand."

They write their pipeline, trading back and forth as before. Once the tests run and they integrate, they take a well-deserved break.

Analysis

This was a hard session, perhaps harder than typical. (Most sessions won't have the "restart" we saw in the middle.) We saw many interactions typical of pairs, though:

- ✧ Asking a partner for help (and receiving it)
- ✧ The partner helps with both strategic and detailed thinking
- ✧ There's a protocol for changing hands: "Can I drive?"

- ✧ The partner provides an ongoing quality boost: review, refactoring, unit testing, new ideas.
- ✧ The partner provides permission to do the right thing, even if it's starting over.
- ✧ If one partner forgets something, the other can remind them.

Pairing is a skill. It takes practice, and it doesn’t start out easy for everybody. Pair programming is a crucial skill in XP, so it’s worth cultivating the habit to take advantage of the benefits.

Sidebar: Why Pair Programming Works

The Nature of the Brain

The first reason pair programming works is that a pair can perform concurrent tactical and strategic work which reduces the overall time to complete a task. There is something about the brain that makes it very difficult to think at a high level at the same time that you are doing a lot of hand-eye coordination. I notice that if I try to think about why I'm typing something or what I should do next my typing speed slows down. On the flip side, if I'm rapidly entering code or working rapidly in an IDE, the quality of my high level thinking suffers a great deal. Pair Programming creates strong synergy by filling in the other roles' weaknesses.

The second reason is potentially a quantum productivity leap and flows from the first reason. There is a great deal of design quality that only manifests when you are working rapidly. If the delay between steps is too long I can't recall the needed information. This point is subtle but very important. Although a process may linearly reduce the time to perform a set of tasks, the quantum increases come when the speed increase allows the information to remain concurrently in memory so that the brain can now suggest dramatic new designs.

The Nature of the Heart

The desire to appear competent to one's peers helps one drive toward results. In addition, new technologies are often demanding, and the availability of a partner who can either share knowledge (or at least share your frustration) is very valuable. On the other hand, knowing that you have knowledge to share increases your confidence and self-esteem. Finally, there is a sense of lowered competition: when you both accomplish the same tasks, there are no feelings of being left out or anxiety that someone else just coded more in less time.

Ensuring Pair Programming Success

Reaping the benefits of pair programming requires one to confront a few inner demons. First, there is the inner critic, the constant chatterbox that relentlessly questions our abilities. For most

of us, comments such as "What's wrong with me, why did she see that mistake and I didn't?" are always present in our heads as long as we're breathing. Pair programming asks us to accept our humanity and continue.

The second demon is our unwillingness to talk and to listen. The first principle of XP is communication and being able to communicate our needs to programming partner requires some bravery. Perhaps we need more breaks; perhaps fewer. Maybe we want to drive more. Maybe we need more silence. I've often found a need to say, "I need a moment to absorb what you just said."

Finally, pair programming requires us to be highly aware and discerning of the software, while being very respectful of our partner. Pausing before jumping in with criticisms allows you more time to understand your partner's thinking.

Pair Programming Experiences in a Nutshell

Exhausting

Four hours of pair programming feels like a full day's work. I find the XP process to be intense and exhausting. It is a major mental and psychological challenge.

Immensely satisfying

There is strong feeling of accomplishment that comes from knowing a task was accomplished in less time and of higher quality than could be done solo. The experience is the mental equivalent of the feeling after an exercise session. Pair programming creates a synergy in that we are able to improve our inter-personal skills while delivering superior products.

—Harris Kirk, July 19, 2000

Sidebar: Pair Programming

On our team we've found that everyone contributes differently to design (and analysis, and good code, and...). One major axis seems to be between the people who are prolific in design (code, etc.) and those who perfect design (code, etc.). The prolific developers are great for getting things started and produce large amounts of imaginative and (typically) adequate to better than average work. The perfecters produce smaller quantities of design/code but the resulting product typically has fewer flaws from the beginning and stands up well over time. Of course, people are all along this continuum.

What I've observed since we've been trying out Pair Programming is that working in pairs pretty much brings out the best in everyone. When two prolific people pair, they tend to slow down and produce higher quality work (but still faster than the average developer). When two perfecters pair up, they tend to get started more quickly (it's hard to sit and stare at your monitor dreaming up the perfect design when someone is sitting next to you asking questions) and are a little more imaginative and prolific. When a perfecter and a prolific developer pair up, the prolific person typically runs the keyboard or whiteboard and starts producing, while the perfecter asks questions and suggests improvements, and adds final polish. Everyone wins.

—Christopher Painter-Wakefield, September 6, 2000

Chapter 6

Where's the architecture?

Architecture shows up in spikes, the metaphor, the first iteration, and elsewhere.

What is architecture? The IEEE definition (P1471) is "An architecture is the highest-level concept of a system in its environment." In the Rational Unified Process (RUP), architecture is supported by multiple views, usually including logical view, implementation view, process view, deployment view, and use-case view. The use-case view is critical as it ties the others together. See "The 4+1 View Model of Architecture," by Philippe Kruchten, for more details.

Architecture is often described as the spine or skeleton of the system, embodying the decisions that have global impact and/or would be difficult to change. Extreme Programming has less emphasis on up-front architecture than other methods because architecture has less impact in XP. XP says "embrace change," while architecture-driven approaches say, "Some things are hard to change, so plan the skeleton first."

On the other hand, architecture in the sense of "the essential structure" has meaning even in XP. XP addresses architecture through several mechanisms:

- Spike
- Metaphor
- First Iteration
- Small Releases
- Refactoring
- Team Practices

Spike

Early on, during the release planning game, the team has the opportunity to do spikes: quick throw-away (and thrown away!) explorations into the nature of a potential solution.

Based on the stories and the spikes, you'll decide on an approach to the system's structure. For example, suppose you have stories about managing orders over the Internet, and the security that requires. You might quickly converge on a solution with a web server, an application server, a database, and a pair of firewalls.

Another story might tell you performance constraints. To investigate performance, you might do a spike simulating many browser clients simultaneously sending queries to the system. This can help tell you how big your boxes have to be.

Another story might give you availability requirements. You'll explore replicating servers, database failover, and so on, to make sure you technically know how to manage high availability, and to help the customer assess costs and tradeoffs.

You'll also do spikes for functionality, to explore how to implement various parts of the system.

Thus, the spikes you do in the exploration phase can guide you to a deployment architecture. Since the spikes begin early on, you

have some lead time for ordering and installing hardware and software. Don't underestimate the value of having this part dealt with early on; I've seen many projects stuck in a panic as they try to get end-to-end communications all at the last minute during final installation. Addressing this early addresses a high-risk part of many projects.

What if the customer just doesn't want *stories about performance, availability, etc.?* That's their prerogative. You have an obligation to let them know your perception of the risks, but the call is theirs to make.

What if the structure has to change later? If it has to, it has to. All you can do is give it your best analysis. With a solution in place, you'll be able to identify any problems sooner than you otherwise would, giving you more time to fix the problem.

Metaphor

During the exploration phase, and all along the way after that, you'll seek an effective *metaphor* that helps guide your solution. For example, in the online ordering system mentioned earlier, the metaphor might be centered around forms and orders.

The metaphor acts partly as the logical design in the Rational Unified Process. It identifies the key objects and their interactions. This helps orient you when you're trying to understand the functionality at the highest levels.

The metaphor may change over time as you learn more about the system, and as you get inspired in your understanding of it. This is an example of the fluidity of the architecture in XP: if the conceptualization *can* be improved, it *will* be changed.

We'll look at the metaphor more in the next chapter.

First Iteration

The first iteration is key in making the system come together. From *Extreme Programming Explained,* p. 113:

> ...the first iteration must be a functioning skeleton of the system as a whole... For the first iteration, pick a set of simple, basic stories that you expect will force you to create the whole architecture. Then narrow your horizon and implement the stories in the simplest way that can possibly work. At the end of this exercise you will have your architecture.

Page 134:

> The first iteration puts the architecture in place. Pick stories for the first iteration that will force you to create 'the whole system,' even if it is in skeletal form.

As we mentioned earlier, Michael Hill talks about using the first iteration to produce a ZFR ("ziffer," Zero Feature Release). The team produces a system that does exactly nothing, but does it in a way that the whole architecture is in place for the system.

There are several things you can do in the first iteration to make it easy to put the system in place:

Be skinny. You don't need all the features of any component. For example, in a web application, make sure the web server, application server, database, and firewalls are all present and can talk to each other in the simplest way possible.

The book *The C Programming Language*, by Kernighan and Ritchie, had many good features, but the one that's stayed with me the longest is the use of a "Hello, world" program: a minimal program that printed that phrase on the console:

```
int main()
{
    printf ("Hello, world!\n");
}
```

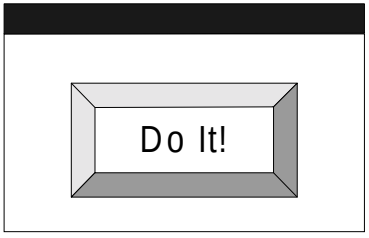
It's a small program, but notice what it takes to get it to run: obtaining access to an account, with the account set up so tools can run; running the compiler and the linker; and running the program itself. This is often the key: if you can get this program working, you can modify it as far as you need to. If you can't get even that simple program working, what chance do you have of getting a realistic program to work?

Configure everything. Major pieces of software like web servers and firewalls have many ways they can be configured. By focusing the first release on an end-to-end version of the system, you help ensure that the configuration is right while your system is still simple to understand.

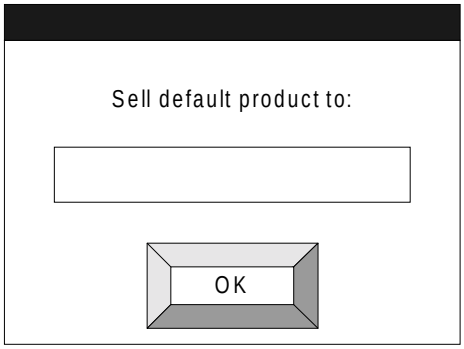
Establish a tricky configuration early while the system is still simple, and keep it working. This is much easier than doing it at the end, when the configuration is just as tricky but the system as a whole is more complicated too.

Shed Functionality. The focus is on an end-to-end solution. If the first iteration gets in trouble, drop any functionality you need to. The first iteration doesn't have to be useful; it has to be installable.

One-Button Interface. We looked at the "Do It!" interface before:



To my regret, I've violated this approach too many times. There is always a lot of pressure to add more features "while we wait for the hardware to get here." I did a web project that should have taken this approach:



If we'd gotten that working (through dozens of configuration issues), nothing else would have been very hard.

Small Releases

XP's small releases help gel the architecture quickly. Since you're only installing at most a few months' work, you are forced to get the essential structure in place quickly. You're delivering the stories the customer values most. Because you get feedback quickly (from real use), weak areas in the architecture show up so you can correct them.

Refactoring

The spike, metaphor, and first iteration go a long way toward defining the initial architecture. Refactoring lets us manage the design going forward. Since refactoring changes the design—the architecture—without changing the system's behavior, we don't risk the functionality of our program while we improve or alter its architecture.

Refactorings often work "in the small" to clean up object interfaces and interactions, but there are "big refactorings" as well.

These can drastically change the form of the system. For example, Martin Fowler and Kent Beck describe refactorings such as "Separate Domain from Presentation," which splits presentation logic from business logic.

Team Practices

There's a level at which architecture dictates things "on the ground." It's lovely to have a software architecture document that explains wonderful things about how the code should work, but it's only valuable if it reflects what people actually do.

XP forgoes the Software Architecture Document that RUP values, but still has an architecture. Pair programming (including switching pairs) helps ensure that the people know and use the approach the team is using. If a better approach is found, the team can share and learn that. An open workspace makes it easy to communicate. The coding standard ensures that code is written consistently

Questions and Answers

Who defines the deployment (technical) architecture? When? The team defines the architecture. This is done during the exploration phase of release planning. As the team understands the stories and spikes potential solutions, it learns what it needs to.

How is the architecture documented? As it will for all documents, the team produces enough to do the job at hand, plus whatever else the customer wants beyond that—wants badly enough to pay for. XP is light on documentation, but works hard to ensure that the team understands and communicates important information within the team.

Why produce a "skinny" version of the system? Wouldn't it be better to get, say, persistence "done right" on the first iteration, then concentrate on business logic in the next iteration, and so on? By

doing a skinny version, you're exercising all parts of the system, so you find out early on if any part has a problem. If you don't deliver a vertical slice of useful functionality in a release, you run the risk that the customer gets stuck with a well-designed part of a system that can't do anything. It's better to have 10% of the features be 100% done, than to have 100% of the features be 10% done.

If everybody is the architect, isn't nobody the architect? In a sense, that's right. In XP, the team is responsible for the architecture. Alistair Cockburn describes XP as a high-discipline process, and this is one of the places that shows up. If the team makes no effort to maintain the metaphor, and doesn't bother converging to a consistent error-handling style, and doesn't focus on a functional system, it will not deliver what it needs to. (The presence of a coach can help a team maintain its discipline.)

So do we want architects on the team or not?
Architecture skills are still valuable; so valuable that we want the entire team cross-trained in them.

Where is the XP approach going to have trouble? I worry about two places in particular, addressed by RUP but not explicitly by XP: the process / thread structure, and the data model. Threading and synchronization are hard to get right, and plagued by subtle bugs and sensitivity to change. Isolate them as much as possible, but they're still risky.

The data model is risky because it tends to be less fluid than the application code. (Consider migrating a 50-million-row table to a new structure, or converting data through a process that takes days.) This may reflect my comfort level more than inherent limitations of XP, but I struggle to manage these areas through the XP philosophy.

My team isn't XP; how much of this can I use? You probably can't rely on the intense communication and oral history in XP: you'll need to document more.

Take advantage of spikes: explore to learn what your architecture needs.

Strive for a skinny implementation as your first iteration. So many problems are related to system configuration, that an early iteration with minimal functionality can help explore these areas thoroughly.

Summary

Although XP doesn't focus on architecture, it has several mechanisms that ensure it is addressed:

- Spike
- Metaphor
- First Iteration
- Small Releases
- Refactoring
- Team Practices

We talked about each of these areas, and explored some questions about how they work in practice.

References

Alistair Cockburn, "Just in Time Methodology Construction"; http://members.aol.com/humansandt/papers/jitmethy/jitmethy.htm.

Ellis et al., "Toward a Recommended Practice for Architecture Description"; http://www.pithecanthropus.com/~awg/ieee_white_paper.pdf.

Martin Fowler et al., *Refactoring*, Addison-Wesley, 1999.

Philippe Kruchten, "The 4+1 View Model of Architecture," *IEEE Software*, November 1995, 12 (6), pp. 42-50; http://www.rational.com/products/whitepapers/350.jsp.

Chapter 7

What is the system metaphor?

"The system metaphor is a story that everyone—customers, programmers, and managers—can tell about how the system works."
—Kent Beck

Why seek a metaphor?

We seek a system metaphor for several reasons:

Common Vision: To enable everyone to agree on how the system works. The metaphor suggests the key structure of how the problem and the solution are perceived. This can make it easier to understand what the system is, as well as what it could be.

Shared Vocabulary: The metaphor helps suggest a common system of names for objects and the relationships between them. This can become a jargon in the best sense: a powerful, specialized, shorthand vocabulary for experts. Naming something helps give you power over it.

Generativity: The analogies of a metaphor can suggest new ideas about the system (both problems and solutions). For example, we'll look at the metaphor, "Customer service is an assembly line." This suggests that a problem is handed from group to group to be worked on, but it also raises the question, "What happens when the problem gets to the end of the line? Does it just fall off?" This can bring out important issues that might otherwise lurk and fester.

Architecture: The metaphor shapes the system, by identifying key objects and suggesting aspects of their interfaces. It supports both static and dynamic aspects of the system.

Choosing a metaphor takes work. Your team should explore several possibilities, looking at the system through their worldviews. If the best metaphor is a combination of two, that's OK.

What if the metaphor is "wrong"? You obviously don't want this problem, but all you can do is your best. If you later realize there's a better metaphor, evolve your system in that direction. This can be a very good sign: it may mean you've made a breakthrough learning in understanding your problem.

What if you can't think up any good metaphors? There's always the "naive metaphor": let objects be themselves. For example, a bank model might have Customer and Account objects. This may not give you any extra insights (since it's based on what you already know), but at least it lets you get started.

What are some system metaphors that have been used?

- A pension tool combining double-entry bookkeeping and a spreadsheet. [*Extreme Programming Explained,* p. 56]
- Desktop metaphor for graphical user interfaces. [Ludolph et al., "The Story Behind the Lisa (and Macintosh) Interface."]
- Bill of materials in VCAPS. [http://c2.com/cgi/wiki?VcapsProject] (VCAPS is a parts management system.)
- Lines, buckets, parts, and bins in C3 payroll. [http://c2.com/cgi/wiki?SystemMetaphor] (C3 is a payroll system, and was the first XP project.)

- ✧ Shopping carts for e-commerce sites. [Discussed in the Extreme-Programming group at egroups.com]

For more information on metaphors in system design, see *Designing Information Technology in the Postmodern Age*, by Richard Coyne, and "A Guide to Metaphorical Design," by Kim Halskov Madsen.

Example: Customer Service

Suppose we need to develop an application to support customer service representatives. For example: a customer calls in to complain that long distance doesn't work on their phone. The rep takes some information from them, and ensures that a technician works on the problem.

We'll explore several metaphors that a team might evaluate, and see the tradeoffs we might make in letting each of these guide the system's development.

The naive metaphor. Customer Service Representatives create Problem Reports on behalf of Customers, and assign them to Technicians. In the naive metaphor, we have to carefully think through the implications of the metaphor, as it doesn't give us much guidance.

Assembly line. Think of problem reports and solutions as an Assembly, and the technicians and customer service representatives as Workers at Stations. This metaphor suggests that it might take a number of steps or people to solve a problem.

Notice how the assembly line metaphor helps carry our understanding. For example, we might wonder how many items per hour come off the end of the line. This helps us think about the "capacity" of our operation. We might wonder what happens to an item once it hits the end of the line. Perhaps we'll want a policy of notifying each customer of the problem's resolution. We might wonder what happens when a station gets backed up. This might help us identify bottlenecks, and might lead us to think of

workers as multiply skilled people who can pitch in where they're most needed.

Some aspects of the assembly line metaphor don't work so well. An assembly line tends to have a relatively fixed structure, while we might want something more flexible. The steps from place to place are typically pre-defined; we might want something that decides routing based on the situation.

Issues like these don't necessarily invalidate the metaphor, but rather help us to explore and understand its limits.

Problem-solving chalkboard. An Expert (CSR or technician) puts a Problem on the Board. There are a number of Experts sitting around: when anyone sees a problem they can solve (or know how to break into easier sub-problems), they do so. There's a protocol that defines, "Who gets the chalk next?" and "When are we done?"

This metaphor suggests a few considerations we face in using such a system. Experts have different skills, and they may not necessarily agree on how to solve a particular problem. The chalkboard may become a scarce resource. The most knowledgeable person may find they're doing all the work. We may have "experts" who aren't as good as they think they are. And so on.

Subcontractors. The customer service representative is the General Contractor, with control over the whole Job. They can let work out to Subcontractors (who can delegate to others).

This model treats the representative as a critical part of the process: the customer's advocate. Commitments and sub-commitments are tracked. Time bounds (service level agreements) can be built into the contracts. Someone is always responsible overall, ensuring that problems don't fall through the cracks.

Workflow. Workflow is a generalization of the assembly line, to support an arbitrary graph of possible transfers, along with the idea of dynamically determining what step is next. It is in effect a semi-standardized version of the naive model.

The concept of workflow may not be a part of some peoples' mental framework. Even if it seems like a perfect fit based on the problem, it may not be the best choice if it won't give people any intuition about the system. (I find it a little abstract for a metaphor, but it might be just right for the right group of people.)

Conclusion. We've discussed several potential metaphors: naive, assembly line, chalkboard, subcontracting, and workflow. Each of these brings a different perspective on the interplay of people and problems. The team needs to consider the strengths and weaknesses of each possible model, and select the best fit they can.

Example: Editors

Editors have used a number of metaphors and models over the years.

Card. An early editor style was to imitate punched cards: some number of lines, with 80 characters each. This was easy to implement with a simple array of characters. Users had to confront the question, "What happens when your text is longer than a line?"

Array of Lines. Other editors used another model: make each line stand on its own, with any length it needs. Again, implementation was simple: an array of pointers to variable-length lines.

String. This model was initially used by TECO, and inherited by EMACS: the text is a giant string (with some character as a line separator). Lines can easily vary in length. This metaphor easily supports many operations (such as multi-line searches) that could be tricky in the preceding models. See *The Craft of Text Editing*, by Craig Finseth.

Sequence of Runs. The previous models are adequate for simple text, but something more complicated is needed to support styles. One solution is to regard the text as a sequence of strings, each with the same style. So,

This is a **bold and *bold underlined*** text string.
might be encoded as:

This is a	(Regular)
bold and	(Bold)
bold underlined	(Bold and underlined)
text	(Underlined)
string.	(Regular)

Notice that each combination of styles is treated as a different run, and that runs don't overlap. This metaphor gives a natural view to the meaning of selecting text and applying a style.

Tree. Another editor style is tree-based, for hierarchical data. This was common in some Lisp editors (years ago), and also used in "structure-aware" or "syntax-directed" editors. It shows up as an outline view in some editors. Today, it's a common model for editing HTML and XML: since they require properly nested tags, the nodes form a hierarchy. Tree editors have powerful operators for locating and re-arranging hierarchies.

Combination. Editors often combine several modes: a tree view may be built out of a string view, and vice versa. A purely tree-based editor may have simple text editing inside, and so on. A combination metaphor may be best, but make sure the combined power is worth the possible confusion it may cause.

Conclusion. Each of these models has been used in real editors. The right metaphor provides the explanatory power you need, in a model that all parties understand.

More Examples

Just to give some more ideas, here are a number of metaphors brainstormed in a 2-hour session, where we considered every system we could think of (including non-XP systems).

Without further ado:

Mail: letter, envelope, stamp, address, address book, mailbox, bank of mailboxes.

Desktop: file, folder, filing cabinet, trash can.

Randomness: dice, wheel, cards; sampling.

Controls and gauges: control panel, dashboard, knob, gauge, thermometer, probe points (special test points on a circuit board), TV remote, calibration.

Broadcast: radio tower, station / channel, tuning, program.

Time: Clock, calendar, timer.

Data structures: queue, stack, tree, table, directed graph, chain.

Patterns: decorator, facade, factory, state, strategy, visitor.

Information objects: Book, tombstone (final report on dead process), tape recorder, bill-of-materials, knitting needle cards (cards with holes punched indicating areas of interest, collected together by sticking knitting needles through them), money sorter, music score.

Identity: Passport, ticket, license plate, password, lock, key, keyring.

Surfaces: Window, scroll; canvas, pen, brush, palette; clipboard; overlaid transparencies; menu; Lite Brite® screens for bitmap pictures (®Lite Brite is a trademark of Hasbro Inc.); typewritten page, style sheet; blackboard (AI system); map; mask, skin; mirror; form; file card.

Dynamic objects: outline, spreadsheet, Gantt chart, highway network.

Connection: plumbing, pipe, filter; bridge; standardized connectors, adapters; plug-in modules; breadboard (components and wires); LEGO® pieces (®LEGO is a trademark of the LEGO Group); puzzle.

Other Objects: air mattress (compression); gate (monitor); checkpoint; glue (ala TeX word formatting); engine; toolbox; stacked camera lenses.

Shopping: shopping cart, checkout.

Notification: alert, alarm; lease; reservation, appointment; heartbeat.

Processes: walkthrough / drive-by / fly-by; slide show; auction, bid; registration; data mining; cooking, recipe (algorithm); voting; experiment; assembly line; double-entry bookkeeping, accounting; architecture, design; pruning; balance; garbage collection.

People: Agent, proxy, concierge (recommendations), servant, tutor, tour guide, subcontractor.

The "data structures" and "patterns" metaphors in particular may be a little weak to base a system around, but we included them anyway. Some of the window-oriented metaphors are probably so folded in to our consciousness as to no longer be so useful. (Twenty years ago they weren't so obvious.)

The metaphor affects how you perceive the system. Consider some of the identity metaphors: a password is changed often, while a key seems more permanent. Both passwords and keys are fairly anonymous. A passport, on the other hand, carries your identity, and a record of where you've been.

Using Metaphors

Determine the metaphor during the exploration phase, when stories are written and spiked. Revise it as you go. Let the metaphor guide your solution. Use its names for the "uppermost" classes. Understand how those classes interact.

When I interview people, I ask them, "Tell me about the three or four key objects in your system, and how they interact." (Now

you're forewarned:) It used to amaze me how hard this was: many people just don't focus on what's key.

Make sure the whole team agrees on the key objects. Given three or four blank cards, would each person on the team write down the same objects?

Does the metaphor describe the problem or the solution? The solution: the key names and relationships.

Bob Koss of Object Mentor says:

> I've used the metaphor to conceptually talk about a system but not to implement the system.
>
> I've used the metaphor to describe the interacting objects used to implement the system, but not to talk about the system.
>
> I've used the metaphor to both describe the system and to describe the interacting objects. This is the holy grail, but isn't always easily achievable.

Limits of Metaphors

It can be hard to find the right metaphor. There are several potential problems:

- The metaphor may not be familiar enough to the group that needs to use it. Double-entry bookkeeping may be the perfect metaphor, but if the accountants understand it and the programmers don't, it won't be a good system metaphor. (This may mean you need to find a new metaphor, or work to help everyone understand it, etc.)
- We may be working with a too-weak metaphor, one not powerful to explain the critical aspects. For example, if we're developing a distributed system, a metaphor that treats it like a simple uni-processor may not be sophisticated enough to explain things like network delays or failure of remote nodes.

- A metaphor may be too strong, overshadowing but not addressing the real problem. "When all you have is a hammer, the whole world looks like a nail."
- We may be working with a metaphor that unduly limits our conception of the system. Ted Nelson, of hypertext fame, argues that a spreadsheet had to be understood as a new sort of thing; in effect he argues that the metaphors provide only a shadow of its true nature. (Cited in Coyne, *Designing Information Technology in the Postmodern Age*, p. 251.) The metaphor in this case may adequately explain what the system is, but not what it could be.
- Many metaphors may be of the form, "it's like a magic xxx" ("a magic typewriter," "magic paper," etc.). This is a two-edged sword. On the up side, it helps us capture some of the extra capabilities our system provides. On the down side, the point where we need the most help ("the magic") is exactly the part that's not covered by the metaphor. Alan Kay, Bruce Tognazzini, and Randall Smith discuss these issues (Kay in Coyne, op cit., p. 252; Tognazzini in *Tog on Interface*; Smith in "Experiences with the Alternate Reality Kit: An Example of the Tension between Literalism and Magic").

Summary

We've briefly explored the meaning and usage of the system metaphor in XP.

The metaphor provides a common vision and a shared vocabulary. It helps generate new understanding of the problem and the system, and it helps direct the system's architecture.

We demonstrated how a couple different systems might explore a variety of metaphors, by brainstorming a few and comparing their world-views.

Finally, we discussed some limitations of metaphors.

References

Anonymous, "Vcaps Project," *Portland Pattern Repository,* Dec. 12, 2000; http:/ /c2.com /cgi /wiki?VcapsProject

Anonymous, "System Metaphor," *Portland Pattern Repository,* Dec. 12, 2000; http:/ /c2.com /cgi /wiki?SystemMetaphor

Kent Beck, *Extreme Programming Explained,* Addison-Wesley, 1999.

Richard Coyne, *Designing Information Technology in the Post-modern Age,* MIT Press, 1995.

egroups.com, ExtremeProgramming list; http:/ /egroups.com.

Craig Finseth, *The Craft of Text Editing,* Springer-Verlag, 1991.

Frank Ludolph, Rod Perkins, and Dan Smith, "The Story Behind the Lisa (and Macintosh) Interface";
http:/ /home.san.rr.com /deans /lisagui.html.

Kim Halskov Madsen, "A Guide to Metaphorical Design," *Communications of the ACM,* V37 #12, December, 1994.

Randall Smith, "Experiences with the Alternate Reality Kit: An Example of the Tension between Literalism and Magic," *IEEE Computer Graphics & Applications,* Sept., 1987.

Bruce Tognazzini, *Tog on Interface,* Addison-Wesley, 1992.

Section 3: Process

Chapter 8

How do you plan a release? What are stories like?

Write stories, estimate stories, and prioritize stories.

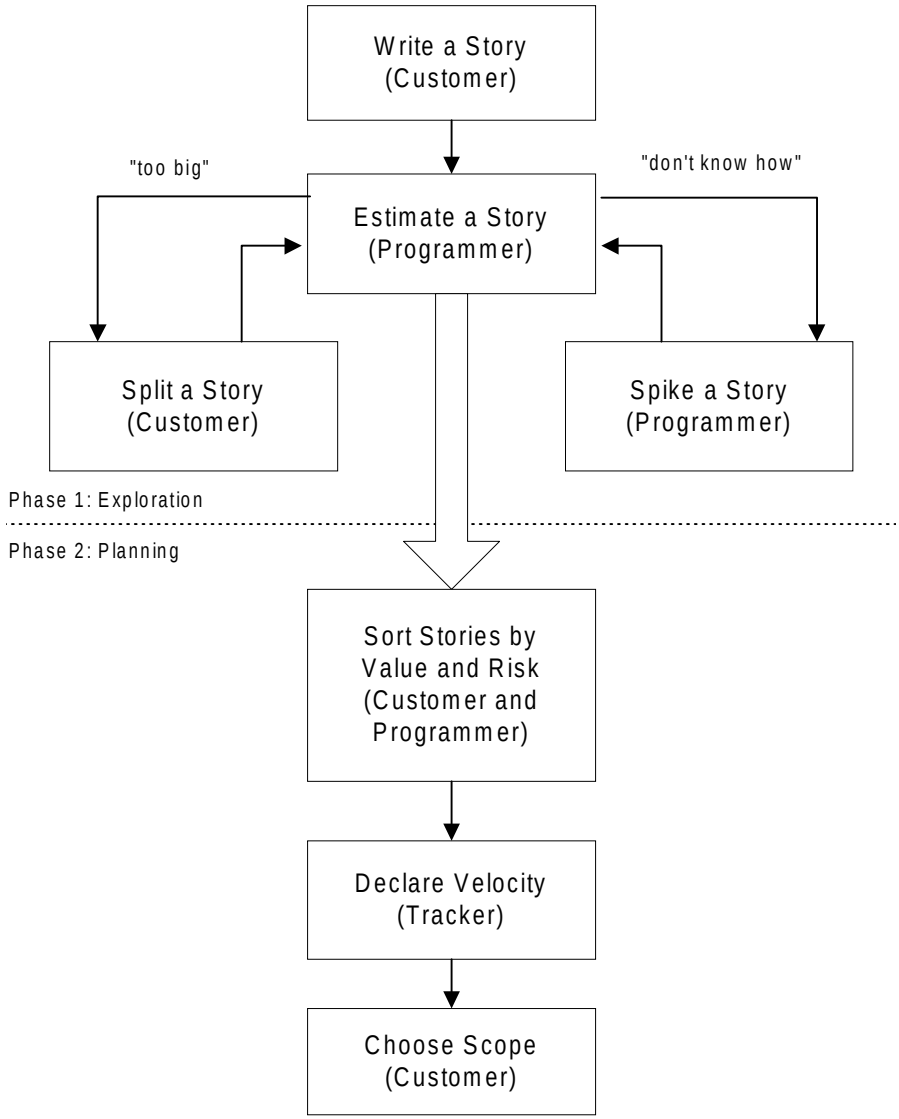
Release Planning

A release is a version of a system with enough new features that it can be delivered to users outside the development group. A release represents perhaps one to three months' work.

The goal of release planning is to help the customer identify the features of the software they want, to give the programmers a chance to explore the technology and make estimates, and to provide a sense of the overall schedule for everybody. The release planning process typically takes from one to a few weeks.

In XP, the release planning process is called the "Release Planning Game." It is a *cooperative* game, not a competitive one; the goal is for everybody to come out ahead. "Game" is meant in the

game-theoretic sense, but we'll describe the process as if it were a "real" game. There are two players, Customer and Programmer. The tokens are the user stories, written on file cards. The game has two phases: exploration and planning. There are various moves the players can make, depending on the phase. The release planning game embeds the rights of Customers and Programmers into its rules. You can think of the game as looking like this:


Write a Story
(Customer)
"too big"
"don't know how"
Estimate a Story
(Programmer)
Split a Story
(Customer)
Spike a Story
(Programmer)
Phase 1: Exploration
Phase 2: Planning
Sort Stories by
Value and Risk
(Customer and
Programmer)
Declare Velocity
(Tracker)
Choose Scope
(Customer)

Release Planning: Exploration Phase

Goal: Understand what the system is to do, well enough that it can be estimated.

Method: The Customer writes and manages stories. The Programmer estimates stories (spiking when necessary). Continue until all stories wanted for the planning phase are estimated. Expect this phase to take a few days to a few weeks.

Result: A working set of estimated stories.

Activities

Customer Writes a Story. The Customer captures an idea, possibly one suggested by the Programmers. (Only the Customer should write a story, and only if they want a feature.) To write a story, take a blank file card and write "Story: story title" on the top. Write a sentence or a short paragraph describing some feature of the system. Try to keep it small enough that the Programmers will estimate one to three story points ("weeks").

Cite or attach supporting materials, if needed. A story card doesn't need all the details though; it's as much a promise to discuss a feature as a description of it.

Remember that the story must be testable. The Customer has to specify the tests (later on), so they should have in mind some mechanism by which to test it.

Give the card to the Programmers to estimate.

Programmers Estimate a Story. The Programmers try to estimate a story. (They'll probably have to ask questions, compare to previous stories, and so on.) The estimate is in ideal programmer weeks (also known as "story points"). If the estimate says it will take more than three "weeks," the Programmers pass the story back to the Customer to split. If the Programmers don't know how to implement it, they do a Spike.

Customer Splits a Story. The Customer may want to split a story for more flexibility, or because the Programmers say it is too big for them to estimate. If the Programmers want a story split, it must be split. Smaller stories tend to be less risky, so it's worth the trouble.

Take the original story card. Write two or more new story cards, each simpler than the original, that when taken together cover the original story. Rip up and throw away the original card. Give the new stories to the Programmers to estimate.

Programmers do a Spike. If the Programmers don't know how to estimate something, they'll do a quick programming exploration of an issue. "Quick" means minutes to hours, to perhaps a couple days. A spike is a skinny, minimal solution in throw-away code. The result of a spike is enough knowledge to attempt an estimate.

In the End. The Customer has written stories for the features they want, and the Programmers have provided each story with a price tag.

Release Planning: Planning Phase

Goal: Plan the next release so it gives the most bang for the buck.

Input: Set of estimated stories.

Method: Do Actions 1 through 4 below. Expect this phase to take up to a few hours.

Result: A prioritized list of the stories currently planned to be included in the next release (and some tentative understanding by Customer of future releases).

1. *Customer Sorts Stories by Value.* From most to least valuable, or at least labeled high, medium, or low. You might think of these as "must have," "should have," and "could have."

2. *Programmers Classify Stories by Risk (Optional).* Labeled high, medium, or low.

3. *Programmers Declare the Velocity.* Declare the velocity, that is, how many story points the Customer should expect per fixed-length iteration, typically one to three weeks. The first time, a reasonable guess is 1/3 story point per programmer per week; after that, determine velocity empirically.

4. *Customer Chooses Scope.* Choose stories for the next release. To judge how long the development effort should take, divide the total story point estimates by the velocity. For the first release, the stories must exercise the whole system end-to-end, even if at a minimal level.

If the Customer doesn't like the resulting schedule, they have a few options: change the stories, release with fewer stories, accept a new date, change the team, or work with a different team entirely.

What's this bit about story points versus weeks? The idea is that programmers give their estimates in what feels like it ought to be a week's work, but never really is. What the programmers estimate as one week will probably take two or three real weeks to accomplish. Since these "ideal weeks" don't really exist, we just call them story points and let everything work out from there. The exact ratio of ideal to actual weeks isn't that important, as long as it's reasonably consistent.

How committed is the customer to having these stories in the chosen order? Not very committed. The customer can change their priorities at any time. The plan just gives a starting point.

What's the point of sorting by risk? The Customer doesn't seem to have to take it into account. The Customer doesn't *have* to use the risk assessments, but some teams feel better if the Customer is aware of what the Programmers think will be hard. On the other hand, other teams have found that the risk factor shows up in the higher estimates, and isn't worth tracking separately. This is an

area where XP's processes are evolving; I believe you'll see this tending to go away.

Example

We'll demonstrate release planning—exploration and planning phases—in an example. First, we need some sort of overall vision for the system

.

> *Vision*
>
> Produce a system able to search an electronic library card catalog.

We might hope for a little more on why we want to do this, and how we think our system will be better than others, but it's a reasonable starting point for now.

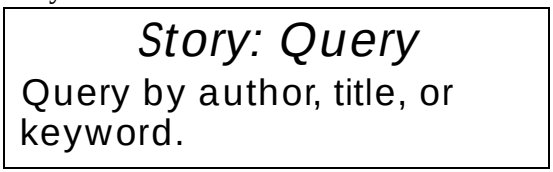
Exploration Phase: Writing and Estimating

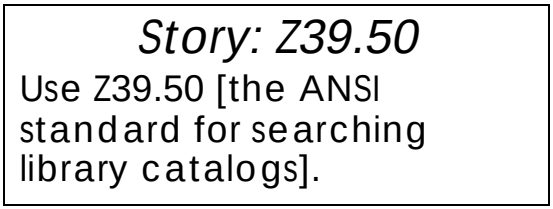
The release planning game begins when the Customer starts writing stories. They write a card:

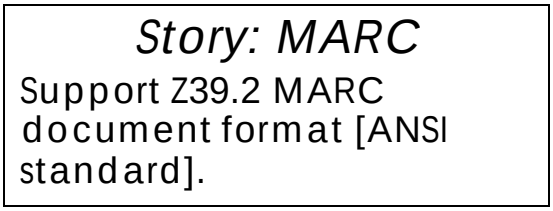
> *Story: Query=>Details*
>
> Given a query, return a list of all matching items. Look at the details of any items.

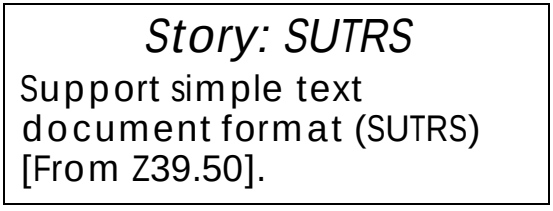
The Programmers look at this story and try to estimate it, but they come back and say "it's too big." The Customer says, "Let

me split it into a few cards," writes some new ones, and throws the original away.







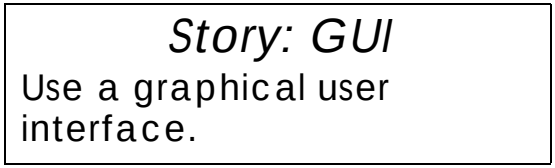


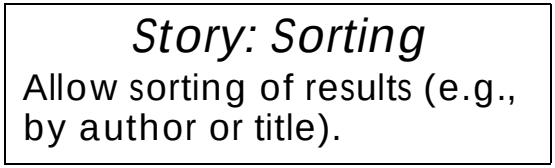
The Customer says, "There might be other document formats later, but I know we need those two."

The Programmer will estimate the stories. We'll assume they have some familiarity with Z39.50 and the ASN.1 protocol it uses. But they may decide to do a quick spike: they already have a tool to handle the protocol encoding, so one pair might attempt to do a simple "login" connection to a Z39.50 system. Another programmer might research the MARC format, and code up a quick tool to parse a record. Someone else might look at SUTRS.

Another pair might try to encode a simple query. (With luck, the programmers will have been able to do some of this investigation before the planning game, or part of the team may have been hired for expertise that can take most of the issues into account.)

After the investigations, the Programmer says: "Queries: 2 weeks; Z39.50: 3 weeks; MARC: 2 weeks; SUTRS: 1 week." (Note that these are really "story points," and may represent more than that many weeks of programmer effort.) Meanwhile, the Customer has been coming up with more stories:





Customer: "We'd like a graphical user interface so it's easy to use."

Programmer: "For what you've described so far, 2 weeks."

Customer: "Not all library systems can sort, but we'd like to take advantage of those that can."

Programmer (after some quick investigation): "2 weeks."

Story: Drill-down

Z39.50 systems can either let you look at quick information (e.g., title, author, year), or the whole record. Instead of always showing the whole record, drill down from quick results to individual items.

Programmer: "2 weeks."

Story: No-drill for 1

Don't make me drill down if the result has only one item.

Programmer: "1 week."

Story: Save Results

Save search results to a file.

Programmer: "What's this for?"

Customer: "It lets other programs see the results; the user can share the file with friends, use it in a document, etc."

Programmer: "1 week."

Story: Print

Print whole list of results or individual items.

Programmer: "2 weeks."

Story: Save Query

Allow saving of queries.

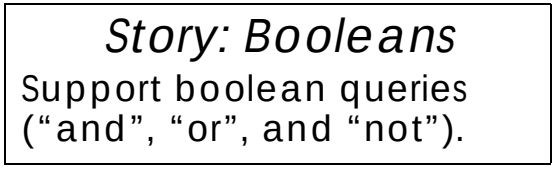
Programmer: "How is this different from saving the results?"

Customer: "You won't get the same result every time even if you search with the same query."

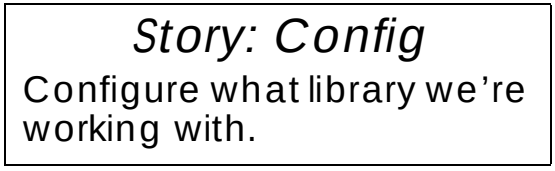
Programmer: "What?!?!"

Customer: "Libraries buy new books every day. Some people like a standing query that they can run just to see 'what's new?'."

Programmer: "Oh, OK. 1 week."



Programmer: "3 weeks."



Programmer: "What kind of configuration information do we need?"

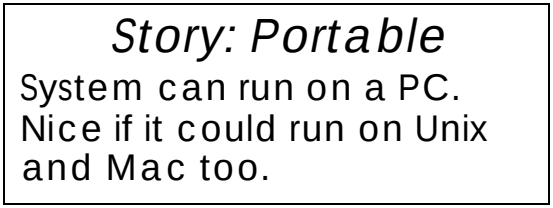
Customer: "It's similar to a URL on the web: host, port, and database name. Many users just use the same library all the time, but others use a variety. We'd like the users to be able to have a list of libraries."

Programmer: "2 weeks."

Programmer: "How about a few more stories: one about what types of systems we should run on, and one about performance?"

Customer: "Good idea. Here you go." (Notice that the Programmer can *suggest* that this type story be added, but it's up to the Customer to write it, and the Customer can reject the suggestion completely.

)

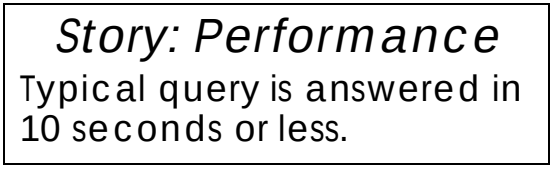


Programmer: “Our programmers are familiar with C++ and Java, and both those could be used. Here are some tradeoffs... [Mercifully omitted] If you want to go with C++, we'll need to split the story because the system-specific parts will make it take longer than 3 weeks to accommodate it. If you’re willing to go with Java, we estimate 1 week (to arrange for multiple machines, and development and testing environments).”

Customer: “We’ll take Java then.”

Programmer: “Could you put that on the card?”

Customer: “OK.” (Notice that the programmer has *recommended* a particular technology, but the final decision is the customer’s: the customer is the one who has to pay to support it going forward.)



The programmers go do a quick spike, using a “quick” GUI and a simplified system. It appears that the bulk of time is spent waiting on the network. They tell the users “1 week.” (This allows for some time to establish a performance monitoring tool, and for performance tuning.)

Planning Phase: Sorting and Selecting

The Customer sorts by value, the Programmer by risk. The estimate is in parentheses.

	High Value	Medium Value	Low Value
High Risk	Z39.50 (3)	Boolean (3) Sorting (2)	
Medium Risk	Query (2) MARC (2) GUI (2)	Drill-down (2)	Print (2)
Low Risk	Performance (1)	SUTRS (1) Config (2) Portable (1)	No-drill (1) Save Result (1) Save Query (1)

Notice how all the high-risk stories seem to have 3-week estimates. That's probably natural, but I would encourage this team to think and explore a little, to see what they could do to reduce the risk; perhaps the Customer could split the riskier stories, or the Programmers could do a Spike on a tricky aspect.

Some teams don't bother to classify the risk, trusting that they can keep the stories simple enough to not bother. Other teams use the risk within an iteration, where they might address the risky parts first. (The Customer gets to choose what goes in the iteration as a whole, regardless of the risk, though.)

The Programmers say, "The velocity is five, meaning we expect to deliver five estimated points per 3-week iteration with our team."

The Customer says, "I know we need a minimally functioning system as soon as possible. These are the stories I think we'll need for the first release."

Planned Release	
Story Points	**Story name**
3	Z39.50
2	Query
2	MARC
2	GUI
1	Performance
1	SUTRS
2	Config
1	Portable

Notice that this is all the high-value items, and select medium-value items. At five story points per iteration, this will take 14/5=2.8 iterations.

Next, development could proceed (beginning with iteration planning). We'll cover iteration planning in the next chapter.

Conclusion

We have:

- Described the Release Planning process, as embodied in the planning game, organized into exploration and planning phases.
- Written and estimated a number of stories, demonstrating the exploration phase.
- Sorted stories, and selected stories for the release plan, demonstrating the planning phase.

Chapter 9

How do you plan an iteration?

Iteration planning can be thought of as a board game.

The goal of iteration planning is to take the stories a team plans to implement in this iteration (the stories currently most valuable to the customer), break those stories into smaller tasks, and assign programmers to work on the tasks.

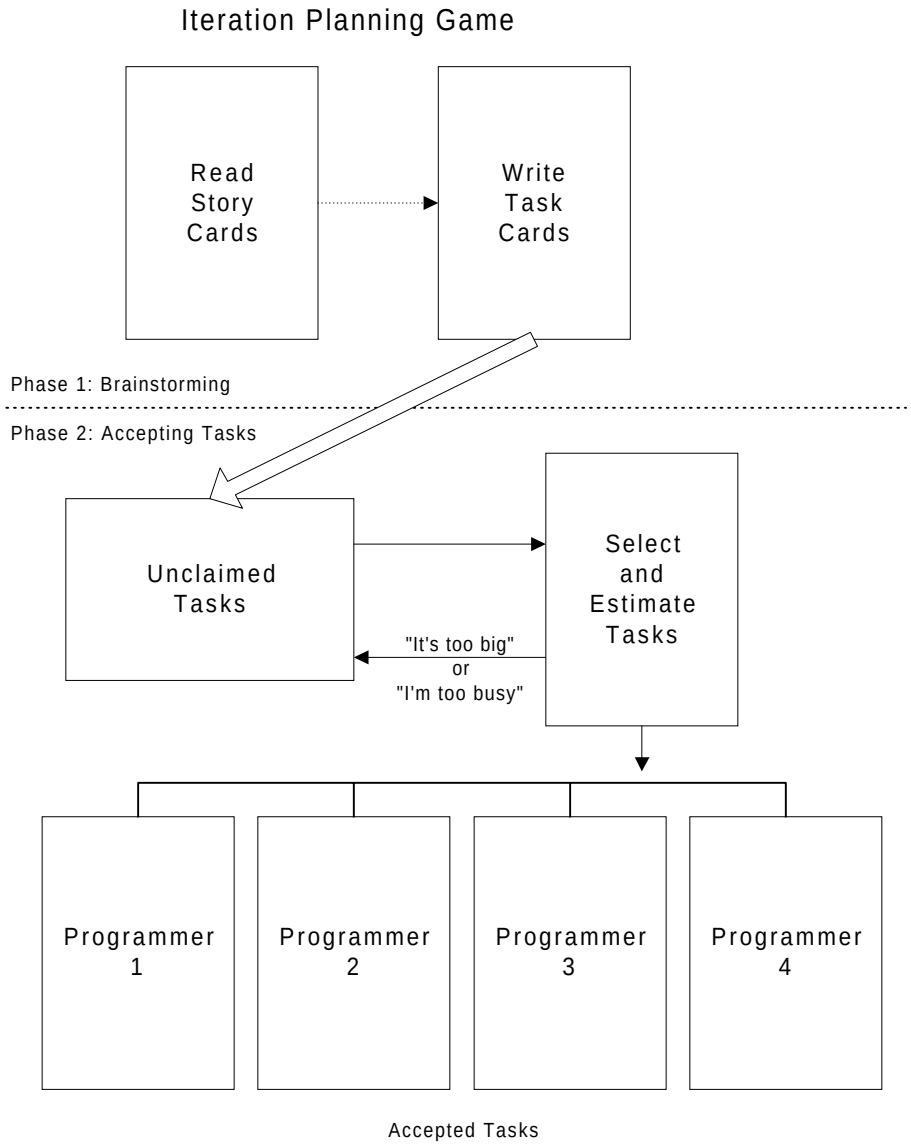
Rather than producing everything in a “big bang” at the end, an XP team uses a series of iterations. Iterations are of a fixed length, one to three weeks long. Iterations are time-boxed: if the team can't get everything done, they will drop features rather than slip the end-date of the iteration. At the end of each iteration, you should expect to see the system ready to deliver, running the acceptance tests for the stories you've chosen.

On the first day of each iteration, the team will decide which stories to focus on. Just as release planning used a game (over weeks), the iteration planning process uses a game too. It typically requires perhaps one to four hours, as this planning covers a much smaller amount of work. The iteration plan will identify

what tasks will be done over the next one to three weeks, and who will do them.

The team accomplished some number of story points in the previous iteration. (For the first iteration, expect about 1/3 story point per programmer per week.) Select as many story cards as you'd like, provided they add up to that many story points or fewer. The stories don't have to be in the order they were in the release plan; the customer can request them in whatever order they like. In fact, the customer may introduce new stories if they're willing to give the team time to estimate them.

To help understand the process, you can think of iteration planning as a board game. There are tokens (story cards, task cards, and coins), and "squares" to move them in. Here is a board and the rules:


Iteration Planning Game
Read Story Cards
Write Task Cards
Phase 1: Brainstorming
Phase 2: Accepting Tasks
Unclaimed Tasks
Select and Estimate Tasks
"It's too big"
or
"I'm too busy"
Programmer 1
Programmer 2
Programmer 3
Programmer 4
Accepted Tasks

Setup

- The Tracker reminds the team how many story points ("weeks") were completed last iteration. (On the first iteration, the team should make its best guess; as a default, assume 1/3 story point per programmer per week.)
- The Customer selects whichever unfinished stories they want, provided the total story points fits that limit.
- The Tracker reminds each Programmer how many task points ("days") they completed last iteration. (On the first iteration, assume 1/2 task point per programmer per day in the iteration.) (This is the "Yesterday's Weather" rule: the past is your best prediction for the future.)
- Each Programmer puts that many coins in "their" Accepted Tasks box. (Different programmers will have different numbers of coins.)

Phase 1: Team Brainstorms Tasks

In the brainstorming phase, read each story card, and work with the team to brainstorm the tasks it might involve. At the end of this phase, the Programmers will have a pile of task cards, corresponding to the story cards you selected. (A given task may support many stories.)

- The Customer picks a story card and reads it aloud.
- The Programmers brainstorm the tasks required to implement that story. (This will usually involve discussions with the Customer.) They write a task card for each task identified.
- Continue until each story card has been read.
- Move the stack of task cards to the Unclaimed Tasks pile.

Phase 2: Programmers Accept Tasks

The accepting phase of the game is mostly the Programmers' job. They will select the tasks, estimate them, and accept the ones they will perform. In addition to conversations to clarify what is needed, the Customer may be involved in this phase for two reasons:

- If the Programmers can't fit in all the tasks, they'll ask to defer or split a story.
- If they have more time than tasks, they may ask for another story.

So:

- For each task, some Programmer will select it, estimate it in task points ("days"), and write their estimate on the card in pencil.
- If the task is bigger than 3 days ("too big"), the team can split the task and put the new tasks on the Unclaimed Tasks pile.
- If the Programmer has fewer coins than their estimate ("too busy"), they erase their estimate and return the card to the Unclaimed Tasks pile.
- Otherwise, the Programmer can accept the task, by removing as many coins as their estimate, and adding the card to their stack.
- This all happens in parallel as various Programmers claim various tasks.
- Continue until the team gets stuck, needs more stories, or wins.

Finishing

Team Gets Stuck: (can't find a way to win)

- The Customer picks a story to split or defer.
- Remove the corresponding tasks. Brainstorm new tasks for a split story.
- Try Phase 2 again.

Team Needs More Stories:

- There are Programmers who haven't used up all their coins, but the tasks are all claimed.

- The Customer can add in another story and see if the team can fit it in.

Team Wins:

- All tasks are accepted.
- The workload is reasonably balanced (everybody has approximately 0 coins left).

Next:

By the end of the game, the team has developed an iteration plan that should enable them to implement your chosen stories.

- The Manager may record the iteration plan.
- Customers start writing acceptance tests.
- Programmers implement tasks.

Questions and Answers

How long should this take? Perhaps a few hours.

Why do we have to identify tasks anyways? Aren't the stories good enough? Some teams just keep the stories small and use them directly, but others find it helpful to identify tasks. Tasks are easier to estimate than stories since they're smaller, and one task may support multiple stories, so there are some advantages to having both.

Why should the Customer have to split or defer a story if the team gets stuck? If the team did so many story points last time, why can't they do that many this time? The estimate for the story was just that: an estimate. After the team has taken the trouble to break it up into tasks and estimate those, they have better information about the actual work involved.

But won't that make the team's schedule slip? The release plan said this story could be done. It might well affect the schedule. You hope there are other stories that come in quicker than you asked to make up for it. If not, at least you know now.

Is the sum of the task points equal to the story points? No, it's not necessarily so. It will depend how you estimate them. If you're reasonably consistent, it will all work out.

How do you know who gets which task? In XP, Programmers choose their tasks, rather than management assigning them. There may be a little jostling, but the team should be able to devise its own accommodations.

But, what if a Programmer wants to do the wrong kind of task? (e.g., a GUI programmer wants to do a database task) XP's natural instinct is to avoid specialization, so this cross-training is actually a good thing. Remember, this programmer will have a pair partner, and they can work with someone who has experience in the area. Also, their inexperience will tend to be reflected in their task estimate (higher than an expert might say for the same task).

Why doesn't the whole team estimate the tasks? Shouldn't the team's best estimate stand? (After all, that's the way you estimate stories.) This is a key difference between release planning and iteration planning. One way to think of it: "The team owns the story, the Programmer owns the task; therefore, the team owns story estimates but the Programmer owns task estimates." There's a psychological reason as well: you're more likely to accept an estimate you made for your own work than an estimate the team assigned to you.

Another reason is that personalized estimates can be more accurate. Suppose Alice finished 5 task points last iteration, and Bob finished 10. It's possible they're equally productive, but just have a different estimation style. Neither estimate, nor their average, is necessarily right for an arbitrary person on the team. If a Programmer estimates consistently and produces consistently, their estimates will converge to the right number over time.

How do we ensure that when we put the tasks back together we've covered the whole story? The Customer will be writing acceptance tests for the story; this will tell you whether the story is done. If

the team is missing tasks, the tests will show it; the team will have to cover them somehow and adjust the schedule if necessary.

How can a Programmer know how long to estimate for a task? The Programmer can talk to the Customer or other Programmers, compare the task to previous task estimates, do a quick spike (5-60 minutes), or go with a number that "feels right."

Can a Programmer make contingent estimates? (e.g., 1 day if Jan helps me, otherwise 3) Sure. Just make sure Jan is willing to play it that way, and don't let things get too complicated.

Do we really have to use the game board? No, of course not. Think of the game board as a diagram to remind you about how the iteration planning process works. Some teams have a white-board and magnets, and attach the stories and tasks there where they can be checked off in public view as they're completed.

Key Ideas

- The team brainstorms tasks, but the individual programmer estimates and accepts them.
- Use the "Yesterday's Weather" rule for both story points and task points.
- The team may have to do some juggling.

Chapter 10

Customer, Programmer, Manager: What's a typical day?

Customer: questions, tests, and steering; Programmer: testing, coding, and refactoring; Manager: project manager, tracker, and coach.

Assume we've been through release planning and iteration planning. What happens in a typical day during an iteration?

Customer

The Customer has four key jobs during the iteration:

1. Answer Questions
2. Write Acceptance Tests
3. Run Acceptance Tests

4. Steer the Iteration

and one job (after several iterations) when the release is ready:

5. Accept the Release

Answer Questions

One of the main reasons you're an on-site customer is so you can answer questions for the Programmers while they're implementing tasks.

When asked a question, answer right away if you can, or promise to find out (and do so quickly). This feedback cycle of minutes instead of hours or days is one of the key things that lets an XP team move quickly.

Many questions are not questions of fact: they require you to make a decision. Your decisions are important: they're how you get what you need. For example, if you've said you want some data cleaned up, the Programmers may present you with options, but you have to tell them which one you want.

Once you've answered a question, you'll want to keep track of your answer. The best way to do this is to write an acceptance test or (more rarely) write a story.

Write Acceptance Tests

For each story in the iteration, determine what would make you confident that the story was properly implemented, and write a test to verify it. Progress in the iteration will be measured against the tests, so the sooner they're available, the better.

In the simplest form, a test will be a card with the title: "Test: test-title FOR story-title," containing information about the test. In a better case, you may be able to specify the test in a spreadsheet; you will want to spend some story or task points to get a tool to run these tests automatically. Also, you may have a Programmer or Tester who will help: you specify the test, and they'll implement it. Be aware that if you require a Programmer to work on the tests, this will impact the number of story points they can deliver.

Software testing is a discipline to itself; a good starting point would be *Testing Computer Software*, by Cem Kaner, Hung Quoc Nguyen, and Jack Falk, or *Black-Box Testing: Techniques for Functional Testing of Software and Systems,* by Boris Beizer.

When you're designing tests, think about the input and the expected result. (You need to capture both.) Test both what should happen when things are right, and what should happen when they go wrong. For example, a word processor test might specify what happens when the disk is full. It can also be revealing to have tests that are at a boundary condition and on either side of it. For example, if you're allowing numbers 1-9999, try 0, 1, 2, 9998, 9999, and 10000 as test values. (There's certainly a lot more to testing than this brief introduction.)

Try to keep your tests fairly independent of each other: when one test breaks, you don't want a chain reaction with too many problems to look at. If tests are independent, you can run them in any order, without a lot of setup.

Acceptance tests are crucial: they're your "gauge" so you can measure how good the program is.

Run Acceptance Tests

Every day, run the acceptance tests for all stories expected to be implemented by the end of this iteration. Let the team know which tests pass and which fail. The Manager (Tracker) will keep a graph where everybody can see it. After you run these tests by hand a few days, you'll see why you will prefer automated tests.

By the end of the iteration, you would like all tests to pass, although it won't necessarily be so.

Steer the Iteration

You have three key opportunities to guide the team:

- Release planning: you choose what's in, what's out, and the release date.
- Iteration planning: you choose what stories are in each iteration.

- On the fly (within the iteration): you choose what the team does *right now.*

There will be times when you need to adjust the plan. Perhaps the development team is not progressing as quickly as planned, and they need to drop some features for the iteration. Given the team's progress, *you* are best able to decide which story is more crucial, or whether a story can be meaningfully split.

Or, there may be a change in business conditions. Perhaps marketing has identified a killer feature that would enable sales to book many orders if only it were available by the show date. In this case, you can introduce the new story, get it estimated, and re-prioritize the team's activities. You prefer not to interrupt an iteration in progress if you can avoid it, but if you ask, the team will set aside, or throw away, what they've done, and devote themselves to the new features.

Accept the Release

As you get closer to release time, the Programmers will be handling the final tasks of installing the software. Once they're ready, you'll want to run the acceptance tests one last time, along with whatever else you need to be sure the software is ready to go live.

When the software is in, and you're satisfied, take a deep breath and declare, "We accept the release." Notify anybody who needs to know.

A release is a big deal; celebrate with the team, then relax and explore a little before jumping in to the next release.

Questions and Answers

But I've got a job to do. How can I spend all my time with this team? Well, hopefully it won't have to be *all* your time. You'll probably be able to do a little work other than this, but this is definitely an investment. Fortunately or not, an on-site customer is the best person to guide an XP team, by deciding what to do, setting priorities, and answering questions.

Can't I just talk to an analyst and let them translate what I want? XP has no analyst role per se; the customer works directly with the programmers. XP strives for the high-bandwidth communication of having the person with the problem talk directly to someone who might be able to solve it.

What if there are many customers? The development process needs input that is prioritized into a single stream of requests. The potential customers need to find a way to trade off their particular requirements.

What if there are many, many *customers?* "Customer" is a role; there may be product managers or marketing people who can stand up and make the necessary *business* decisions.

Programmer

XP is unusual as methodologies go, in that it prescribes activities at the day-by-day (and even at the minute-by-minute) level. We can think of XP's programming activities as shown below.

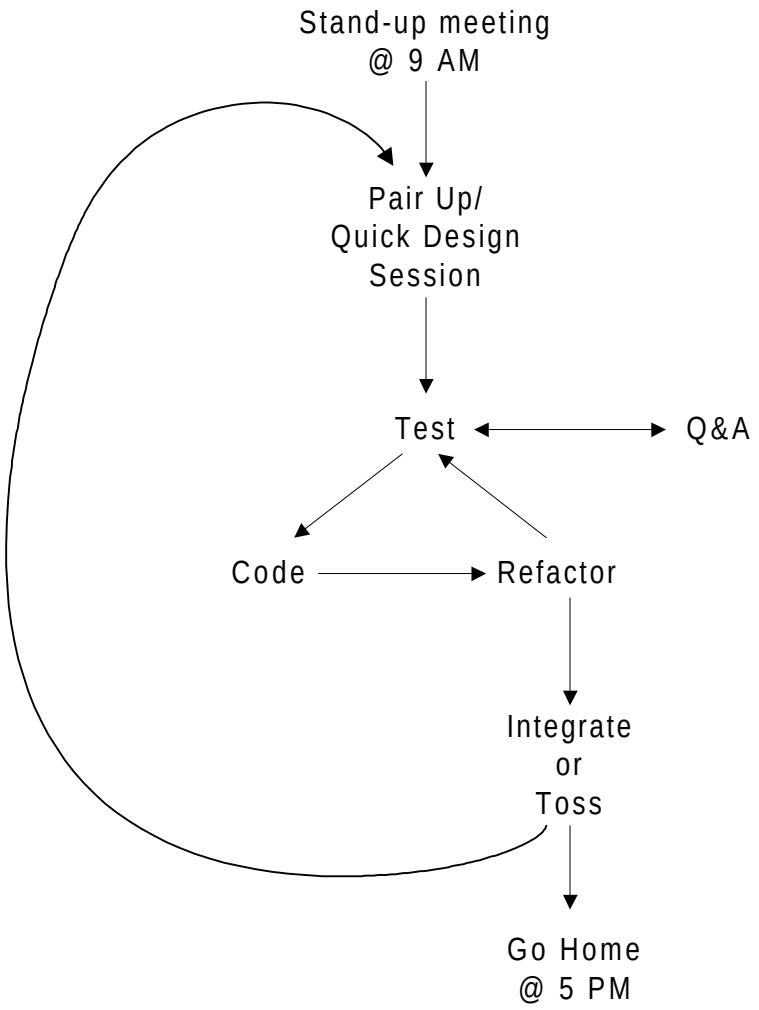
The triangle in the middle captures the idea of test-first programming, a key feature of XP:

- test, then code, then refactor

This is the opposite of traditional programming, which does:

- design, then code, then test

In traditional programming, the cycle might take weeks to months; in XP, the cycle happens on the order of minutes to hours.



Let's look at the activities one at a time.

Standup Meeting at 9 AM

- Meet for ten minutes at the beginning of the day.

- Identify problems and who will solve them, but don't try to solve them in the meeting.
- The fixed starting time helps remind the team to work a 40-hour week.

Pair Up and do a Quick Design

- All production code is produced by a pair.
- The typist thinks tactically, the partner thinks strategically.
- Switch roles periodically.

Test

- Write only small bits of unit-test code at a time.
- Verify that the test fails before coding. (It's sure interesting if it doesn't fail.)
- "Test everything that could possibly break."
- Ask the customer if you're not sure what the answer should be.

Code

- Implement just enough to make the tests pass. ("Do the Simplest Thing That Could Possibly Work.")
- Follow the team's coding standard.

Refactor

- Seek out "code smells" (places that don't feel right); apply a refactoring; verify that the unit tests still pass.
- The code should:
  - Run all unit tests
  - Communicate what it needs to
  - Have no duplicate logic
  - Have as few classes and methods as possible
- Take small steps, and unit-test after each.
- See *Refactoring*, by Martin Fowler.

Q&A (Question & Answer)

- The Customer is available on-site to provide immediate answers.
- Many questions require decisions (not facts), and the Customer should be prepared to make them.

- The Customer should write an acceptance test or (more rarely) a story to capture their answer.

Integrate or Toss

- Move the code to the integration machine, build the system, and run all tests.
- Fix things to bring the unit tests back to 100%.
- If you can't easily integrate, throw it away and try again tomorrow.

Return to "Pair Up"

- If you have time left in the day, you can pair up (or at least switch roles), and start over on another task (perhaps the partner's).

Go Home @ 5PM

- Going home on time reinforces the practice of having a 40-hour week.
- Notice that nothing is hanging over your head: everything you've done for the day is integrated or tossed.

Summary: Programmer

These activities remind you of important aspects of XP:

- Always have a partner.
- Get answers before you code.
- Test-first programming.
- Throw it away when you need to.
- Go home "clean."

Manager

There are three prominent roles in managing XP:

- (Project) Manager
- Tracker
- Coach

The *Manager* owns the team and its problems. They present a face to the world, they form the team, they obtain resources, and they manage people and problems.

The *Tracker* helps the team know if they're on track for what they've promised to deliver; this is typically not a full-time role.

The *Coach* helps the team use and understand the XP approach, mentors the team, and helps the team get back on track if they go "into the weeds."

Depending on the size of the team, all three roles might be embodied in one person or in several. A team of moderate size will want separate people, as it can be emotionally hard to be, e.g., manager and coach.

What Doesn't *an XP Manager Do?*

If you're a manager of an XP team, there are several things you won't do, that a typical project manager might under some other discipline:

- ✧ You don't set priorities; the customer does that.
- ✧ You don't assign tasks; programmers do that.
- ✧ You don't estimate how long stories or tasks will take; programmers do that.
- ✧ You don't dictate schedules; the customer and programmers negotiate them.

What's left? That's what we'll explore.

The Manager

The Manager has several key jobs: face outside parties, form the team, obtain resources, manage the team, and manage the team's problems.

Face Outside Parties

As a Manager, you will deal with several parties. The first is "funders": a manager needs resources (including money) to manage. The Manager provides the funders with insight into the results of their spending. If a manager can't convince someone to support them, there will be no team.

You will deal with Customers. The Customer is the person (or group) willing to claim responsibility for setting priorities and

choosing functionality. This may be a user or user surrogate; it could be (but often isn't) the funder. XP requires an on-site Customer who will make critical decisions. An XP Manager needs to set the Customer's expectations appropriately: the team needs a knowledgeable and committed Customer.

You will interact with other people as well. This will include people internal to the company (e.g., database administrators), or external groups (e.g., salespeople or specialists).

Form the Team

The Manager assembles the team of Programmers (through hiring, transferring, or contracting). If you're forming an XP team, the Programmers need to know that going in. The Manager will help the team bond, and help to establish its process. Ideally, especially for a larger team, the team will include a Coach.

Obtain Resources

The Manager must obtain resources for the team.

You need to create a team workspace. XP values face-to-face communication as the highest-bandwidth way to share knowledge. To reinforce this, it uses an open workspace. This space needs to be set up to allow people to pair-program. (This doesn't fit the usual corporate model of cubes.) Most teams will want at least some "personal space" as well.

The team needs hardware and software with which to program. Some teams dedicate a machine to integration and release. Other teams simulate this with a physical token (although it's a lot easier to forget to grab the stuffed animal than to forget to move to the standard machine).

Your team won't be experts at everything: at some point, they may ask you to obtain specialist consulting. XP teams regard specialists as a resource for their learning: rather than expecting an expert to do a job, they want the expert to teach the team to do the job itself.

You'll need miscellaneous office supplies: paper, pens, markers, and of course file cards. (Like anything, you can delegate this job, but the ultimate responsibility is yours.)

Manage the Team

The Manager has several responsibilities centered around the team:

Report Progress. The funder, and possibly many other groups, care about the progress and status of your team. Track risks and issues so those people know what they need to.

Host Meetings. You'll handle the logistics for meetings: making sure there's a place to meet, that everyone knows about it, that supplies are available, and so on. The key meetings are the release planning meeting, the iteration planning meetings, and the daily stand-up meetings; you may find you need other meetings as well, but don't let "meetings" get in the way of "work."

Host Celebrations. You have many opportunities to celebrate what the team does: release plan complete, iteration complete, system released, or other important days. Handle the logistics: make sure the right people are invited, and that the team gets the celebration it deserves.

Manage Problems

As a Manager, you'll face problems. (XP doesn't change *everything*.)

A project may hit a crisis of some sort. The Manager needs to be in the loop, ready to deal with the Customer or even the funder. The attitude is not "Woe is me," but rather "This is something you need to know," hopefully followed by "Here are some offers we can make to help it get better."

A team will hit roadblocks that don't quite reach the crisis level. There may be places where your authority can get something done in minutes that would take the Programmers days or weeks. Perhaps it's in dealing with an troublesome outside team, or with

purchasing software, or solving an environment problem. If even you can't make headway, you may need to get the Customer involved.

Sometimes the problem will be a troublesome member of the team: you may have to get rid of a team member who refuses to follow the team's rules.

If there are outside distractions, the team should minimize these where it can, and find a way to deal with them where it must. Production support is an example; you might sacrifice someone to an iteration of pager duty.

If there is outside pressure, remember that the Customer is in the hot seat for setting priorities; deflect such pressure away from the Programmers.

Tracker

The Tracker handles the mechanics of measuring the team's progress. (Some teams combine the Tracker with the Manager).

The Tracker can act as a disinterested party, with little "skin" in the game. This is a strength. I've been in positions where I've been 95% Tracker and 5% Programmer. It amazes me how much having one task on the tracking list interferes with my perception of the priority and status of the others.

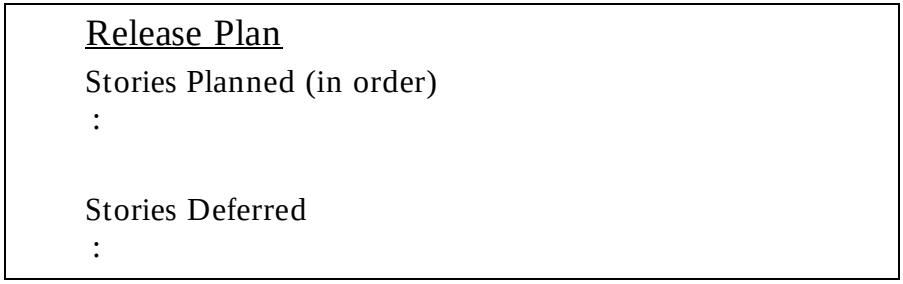
There are three basic things the Tracker will track: the release plan, the iteration plan, and acceptance tests. You can easily make simple spreadsheets for the tracking you need to do.

You may find other things to track as well. If you can measure a problematic but controllable behavior, the very act of putting up a chart will tend to improve it. Don't overburden yourself though: remove such a chart once it has served its purpose.

Track the Release Plan (Stories)

In release planning, the Customer decided the order of stories, and which ones are planned for the release. In its simplest form, this information can be maintained in two stacks of cards, those

"in" (in order) and those "out." Or, you may want a chart as a more permanent record:

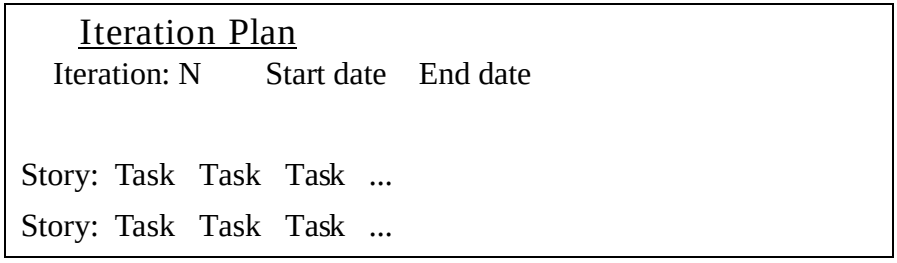


The release plan probably won't change radically more often than every few iterations, so it's not a big burden to track this information.

Track the Iteration Plan (Tasks)

The iteration plan covers a shorter time period, but requires more active tracking. Recall that iteration planning will identify the stories and tasks to be completed in an iteration.

Creating the iteration plan requires some critical information: how many story points did the team complete, and how many task points did each programmer complete, in the previous iteration. (It's your job to track and supply that information.) Again, you might maintain this information on cards, on a whiteboard, or in a chart:

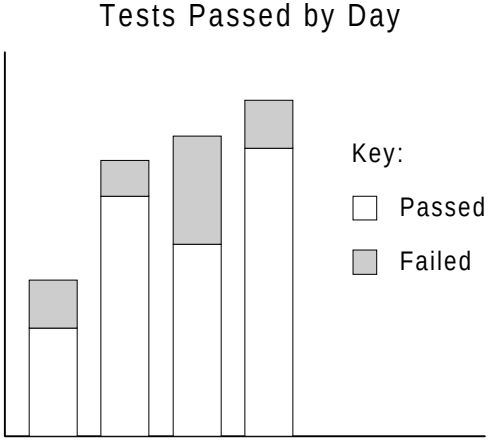


(Note that one task may support many stories.)

During the iteration, at least a couple times a week, you'll talk to each Programmer and find out how many task points they've spent on a task, and how many remain until completion. You might get this information during the daily standup meeting. You can use this information to decide if a team is on track for the iteration. If you're on the middle Monday morning of a 2-week iteration, but fewer than half the tasks are done, raise a flag. With luck, the team can adjust internally. If not, the team may have to ask the Customer to adjust the plan by deferring or splitting stories.

Acceptance Tests

The Tracker will maintain a chart for the acceptance tests (the tests defined by the Customer). Normally, the Customer or a Tester will actually run the tests. You might make a table, but the information is far more compelling as a poster in graph form:



As the days go by, you would like to see more tests in total, and fewer failed tests. If that's not the case, people will notice.

The Coach

The Coach role evolved as someone "on the ground" to help the team maintain its process. Alistair Cockburn describes XP as a "high-discipline" process, and the Coach is one of the tools XP uses to maintain that discipline. We'll look at the attributes of a Coach, the activities they do, and some problems they might face.

Attributes

The Coach is a mature person. "Calm" is too passive a word; it would be more accurate to say they are "centered" or unflappable, not easily fazed. The ones I've known (XP or not) have all been good, even expert, Programmers. (And the team seems to sense that if they really need it, the Coach can go deep into the well for something to help.)

A Coach is respected, but also respectful. They're willing to talk, but also willing to listen. They let the team explore, but provide a guard-rail in case of danger.

Finally, the Coach is on-site. Just like the Customer must be there to answer questions, run tests, and set priorities, the Coach is there to mentor, monitor, and help.

Activities

The Coach's activities focus on the process and the team. The Coach will tend to own few or no development tasks.

Monitor the Process. The Coach's key job is to monitor the process. If stand-up meetings become 2-hour affairs, or people are skipping unit tests, or acceptance tests are not being written, or people are consistently leaving late, the Coach needs to blow the whistle. Usually, just calling attention to a problem is enough to wake people up to solving it.

Enforce the Process. On occasion, you get someone who purposefully does not follow the team's rules. This can't be allowed to continue; it can destroy the team's productivity and morale.

Ron Jeffries, the coach on Chrysler's C3 project, describes his approach:

> What I do (as opposed to how I talk) is that when someone is transgressing rules in some way, I'll have a talk with them. Usually something bad will have happened, which lets us begin by agreeing that something bad shouldn't happen again. I relate the rules to the bad effect in an impersonal way:
>
> "The UnitTests weren't run before the code was released. The UnitTests would have shown the problem. That's why we insist on running them every time. We're fallible, the computer isn't."
>
> Usually the problem won't occur again. Also I watch the person and nag them a few times in a friendly way.
>
> Perhaps most importantly, I'd coach the other team members to watch out for the bad behavior when partnering. In other words, gang up on him.
>
> If it does happen again, I'll have a still friendly but serious chat that says "Everyone in the group really does have to follow the rules." Since they're programmers, they can do the math on people who don't follow the rules.
>
> If it happens a third time, I would politely but gently remove them from the group. If removing them isn't possible, I'd make it turn out that they didn't get to work on anything important. [...]
>
> ExtremeProgramming (and leadership in general) is a work of love. If you don't respect others, you're not doing it right. I try always to let my great respect show through for people who try hard to do the right thing. And sure enough, they do try, in almost every case. The others, who are perhaps trying in some way I don't understand... I respect them too... and wish them success elsewhere.
>
> Ron Jeffries, http://c2.com/cgi/wiki?EnforcingMethods

Change the Process. One of XP's rules is "They're just rules": sometimes, the best thing to do is change a process. The Coach can help the team spot these areas, and work through to a new approach.

Mentor. The Coach is available for mentoring. This may be to work as a design partner or programming pair, or to help someone find their way on a subject new to them.

Supply Toys. The Coach may see that the team needs a reminder about something, and a toy may be just what's needed. Or a toy may be needed to give the team permission to relax a little.

Problems

In addition to keeping the process in line, the Coach will be sensitive to problems that might arise. Here's a sampling:

Velocity Slowing Down. The Tracker may report that the team is completing fewer task points or story points than planned. If there are no obvious reasons (a hurricane shut down the building, lots of family emergencies, etc.), the team needs to do some soul-searching: are they refactoring? producing all the tests? pairing? etc.

Messy or Duplicate Code. This is a sign that refactoring is not being done well or often enough. The team needs to be reminded that "simplest" code requires work to keep it that way. In a bad case, the team may have to take some time just for refactoring.

Quality Going Down. The team needs to focus more on testing "everything that could possibly break." See if you can identify a particular type of error that seems to be slipping through, and devise a test approach for it.

Summary: Manager

We've looked at three roles in managing XP:

Manager: face outside parties, form the team, obtain resources, manage the team, and manage problems.

Tracker: track the release plan, track the iteration plans, track acceptance tests.

Coach: monitor, enforce, and change the process; mentor; supply toys; handle problems.

A team may organize these roles how it needs to, but you should ensure that each of these areas is addressed.

References

Kent Beck, *Extreme Programming Explained*, Addison-Wesley, 1999.

Kent Beck and Martin Fowler, *Planning Extreme Programming*, Addison-Wesley, 2000.

Boris Beizer, *Black-Box Testing: Techniques for Functional Testing of Software and Systems*, John Wiley & Sons, 1995.

Cem Kaner, Hung Quoc Nguyen, and Jack Falk, *Testing Computer Software 2/e*, John Wiley & Sons, 1999.

Alistair Cockburn, "Just in Time Methodology Construction," http: / / members.aol.com / humansandt / papers / jitmethy / jitmethy.htm.

Chapter 11 Conclusion

How rigid is XP? XP is not as inflexible as I've made it sound throughout the book. Kent Beck's "Twelve Practices" are a starting point, but XP will change in each environment.

There's a caution though: if you drop the open workspace (your facilities won't allow it), and the on-site customer (they're too busy), and the pair programming (we don't like it and we're in two locations anyway), and so on, you can find yourself thinking you're doing XP without really getting all its possible benefits.

XP asks a team to be "aware," thinking about what works and what doesn't, and looking for ways to improve.

What does XP tell us that we already knew? Several things:

- High-bandwidth communication works. If you get people in the room where they can make decisions and see results, you can speed things up immensely.
- Validation is important. XP reduces the time between an idea, its validation criteria, and its implementation. By requiring automated, repeatable tests, the team is sure it is moving forward.

- ✧ Iteration works. Iteration isn't new, but many teams that claim to be iterative are not. XP forces iteration by prescribing multiple integrations per day, one- to three-week long iterations that deliver a working system, and releases every few months.

What is the next big skill? The market will go to those who have big visions, but can get there through small releases.

- ✧ Identify and implement 90% solutions
- ✧ Break stories into easy and hard parts
- ✧ Know what's important and what's not

Where is XP going? "It's hard to make predictions, especially about the future." (Yogi Berra) I'll make some guesses anyways:

- ✧ Simplification and clarification of the process. The planning process still feels a little more complicated than it has to be. How to integrate continuously is probably not explained well enough. The metaphor idea is powerful, but many teams aren't using it.
- ✧ Search for underlying values. The XP community has been exploring whether there are underlying principles that can inform other useful methods.
- ✧ Limits of XP. Can a large team do XP? Can it apply in this domain or that? When and where is XP the right thing to do?
- ✧ Acceptance tests. XP uses automated acceptance tests generated by the customer. I think we'll see a burst of energy here as people face what this means in different environments.
- ✧ Customer role. We've seen the programmer's view of this role. ("Give me a bunch of easy stories to work on, answer all my questions, and give me a bunch of tests so I know when I'm done.") Being a customer of software is a hard job, and I think there will be many more ideas on this subject.

—William C. Wake

William.Wake@acm.org

www.xp123.com

Chapter 12

Annotated Bibliography

Extreme Programming

Anonymous, "Extreme Programming Roadmap," *Portland Pattern Repository,* Dec. 12, 2000;
http://c2.com/cgi/wiki?ExtremeProgrammingRoadmap

Kent Beck, *Extreme Programming Explained*, Addison-Wesley, 1999. The original book describing the XP process, and you won't want to miss it. It describes the philosophy of XP, and gives overviews of all the core practices.

Kent Beck and Martin Fowler, *Planning Extreme Programming,* Addison-Wesley, 2000. Planning, in detail.

Ron Jeffries, Ann Anderson, and Chet Hendrickson, *Extreme Programming Installed*, Addison-Wesley, 2000. tells how the C3 team at Chrysler used XP. It begins with "The Circle of Life,"

clear customer and programmer roles, and continues running strong on how the practices are used.

Martin Fowler et al., *Refactoring: Improving the Design of Existing Code,* Addison-Wesley, 1999. The mechanics of how to improve code. The core of the book is a catalog of step-by-step refactorings, shown in Java.

Classics

Jon L. Bentley, *Writing Efficient Programs,* Prentice Hall, 1982.

Jon L. Bentley, *Programming Pearls 2/e* (1999), and *More Programming Pearls* (1988), Addison-Wesley.

Fred Brooks, *The Mythical Man-Month,* 1982.

Tom DeMarco and Timothy Lister, *Peopleware,* Dorset House, 1987.

Ehrich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software,* 1995.

Donald E. Knuth, *The Art of Computer Programming. Volume 1: Fundamental Algorithms,* Addison-Wesley, 1968.

Slow Going

Richard Coyne, *Designing Information Technology in the Postmodern Age,* MIT Press, 1995. Metaphors and multiple viewpoints.

Donald Schön, *The Reflective Practitioner,* Basic Books, 1990.

Testing

Boris Beizer, *Black-Box Testing: Techniques for Functional Testing of Software and Systems,* John Wiley & Sons, 1995.

Cem Kaner, Hung Quoc Nguyen, and Jack Falk, *Testing Computer Software 2/e*, John Wiley & Sons, 1999.

For Fun

Rob Reiner (director), *The Princess Bride*, MGM, 1987.

J.R.R. Tolkien, *The Hobbit, or There and Back Again*, Houghton Mifflin, 1999 (originally 1937).

Web Sites

news://comp.software.extreme-programming

www.egroups.com/group/extremeprogramming

www.extremeprogramming.org, Don Wells' site

www.junit.org, JUnit's home.

www.refactoring.com, Martin Fowler's refactoring catalog.

www.xpdeveloper.com

www.xprogramming.com, Ron Jeffries' site

www.xp123.com, my site.