Extreme Programming in Perl

Robert Nagler

February 22, 2017

 nagler@extremeperl.org

Contents

nagler@extremeperl.org

nagler@extremeperl.org

nagler@extremeperl.org

Preface

> Have fun, and build something cool.
>
> – Pete Bonham

This book is about a marriage of two compatible yet unlikely partners. Extreme Programming (XP) is a software development methodology that enables users, business people, programmers, and computers to communicate effectively. Perl is a dynamic programming language that lets an XP team embrace the inevitable change caused by effective communication. Perl is the fixer and doer of the pair, and XP is the organizer and facilitator. Together they help you build robust software applications efficiently.

Like any good marriage, the partners of Extreme Perl support each other. For example, XP asks business people to write acceptance tests, and Perl lets the business people use their own language and tools for the tests. Much of Perl only happens when the program runs, and XP asks programmers to define what is supposed to happen in unit tests before they write the program. In this book, you'll see other examples where Perl reinforces XP and vice versa. This mutual support system is what makes Extreme Perl applications robust.

This book invites Perl programmers and their customers to take a fresh look at software development. Customers, and business people in general, will learn how XP enables customer-programmer communication for efficient and flexible requirements gathering. Programmers will see how XP's focus on teamwork, incremental testing, and continuous design allows them to take pride in their craft. The numerous examples demonstrate Extreme Perl in action, including the development of a complete, end-to-end application in the last chapter.

To Business People and Users

XP combines your project responsibilities into a single official role: the customer. That's the extent of the formalism. You don't need to learn use-case modeling, object modeling, or even fashion modeling. You write your requirements on a piece of paper with pen. You even get to draw pictures, although the programmers would prefer you didn't use crayon.

As the customer, you have the responsibility to speak in one voice. You can discuss the requirements as much as you like, but in the end, you write down a simple, clear requirement in your own language, called a story. Any disagreements need to be settled during the planning game, where you and the programmers hash out what needs to get done and how long it is going to take.

XP lets you change your mind. That means you have to hang around the programmers–something that may take getting used to. Programmers are terrible mind readers, and your immediate feedback is necessary when they get the requirements wrong or you realize a requirement isn't quite right.

Best of all, you get to see progress right away. The programmers do the simplest thing that could possibly work, and believe it or not, this actually produces a working program in a matter of weeks. There's nothing better than seeing your requirements embodied in software to ensure you are getting what you want, and that you get what you are paying for. Everybody is motivated by a working product in use.

To Programmers and Their Managers

The programming role is quite broad in XP. Programmers are responsible for listening to the customer, and reacting to the dynamic requirements of the customer's world.

With XP, you get to be real. No more fudged estimates or wild guesses. If the customer adds a complex requirement, like internationalization, right in the middle of the project, it's clearly not for free, and you get to say how long it will take.

XP managers are coaches and trackers. The programmers do all the work, and the coach gives sage advice while sipping martinis. If all goes well, the tracker has a passive role, too. XP's 12 simple practices add up to a lot of checks and balances. Sometimes the coach and tracker must remind the programmers how to use the practices effectively, however.

Code is the core artifact of an XP project. You have to like to code to

nagler@extremeperl.org

do XP. Fortunately, Perl is fun and easy to code. XP adds a modicum of discipline to Perl coding practices that enables you to code faster and better.

XP is reflective. The code gets better, because you refactor it frequently. This means you get to fix the bad code in the project that usually everybody is afraid to touch. With XP, you might not be so afraid. You and your pair programming partner have lots of tests to ensure you don't break anything while refactoring.

The goal of refactoring is to represent concepts once and only once. Perl lets us do this more easily than other languages. Extreme Perl code can be quite compact without obfuscation–rather the opposite. The code often evolves into what I call a subject matter oriented program (SMOP).

Subject matter oriented programming distills the essence of the problem into a little language. The SMOPs in this book are plain old Perl. There's no need to invent new syntax. Sometimes you may need to think differently to understand a SMOP unless you already are familiar with declarative programming, which is quite different than traditional imperative programming–what most programmers learn in school.

You need to know Perl fairly well to read many of the examples. I explain the examples without going into much detail about the Perl syntax and semantics. You may need to keep your favorite Perl reference book within reach, for example, to undertand how `map` works.

One last thing: the some of test examples use the bivio OLTP Platform (bOP), an open source application framework developed by my company (http://www.bivio.biz). If you write a lot of tests, you need tools to help you write them quickly and clearly. bOP employs SMOP to simplify unit and acceptance tests. I'm not trying to sell bOP to you–it's free anyway–but to demonstrate SMOP in testing. This book explains just enough of bOP to read the examples.

How to Read This Book

This book explains Extreme Perl to both programmers and business people. I also attempt to convey the Extreme Perl experience through examples and personal anecdotes. The book covers Extreme Programming (XP) in detail, so no prior experience is necessary.

The first part of this book is about the non-programming aspects of Extreme Perl: the why (The Problem), the what (Extreme Programming and Perl) and the how (Release Planning, Iteration Planning, Acceptance Testing, Tracking, and Pair Programming). There is some code in Acceptance

nagler@extremeperl.org

Testing, but everybody should be able to read it. The last chapter (It's a SMOP) is an end-to-end Extreme Perl example that combines XP, Perl, and SMOP. Non-programmers may want to scan this chapter and read the conclusions at the end.

The second part of this book contains numerous programming examples from the real world. The following chapters show you what it is like to do Extreme Perl: Coding Style, Logistics, Test-Driven Design, Continuous Design, Unit Testing, Refactoring, and It's a SMOP.

If you are a top-down thinker, I recommend you read this book front to back. Bottom-up thinkers may want to start at the last chapter and work backwards.

As noted in the previous section, the Perl code in this book is advanced. The programming examples are not complex, that is, they are short and contain only a few concepts. However, the code may appear complicated to some programmers. If you are familiar with functional programming and object-oriented Perl, the examples should be clear. If not, you may want to peek at the last chapter which describes functional programming. The references throughout the book may be helpful, too. The object-oriented aspects are not all that important, so you should be able to understand the examples without object-oriented experience.

Typographical notes: I note acronyms that are used in the XP and Perl communities throughout this book, but I use their expansions so you don't have to memorize them.

Acknowledgments

This book was a collaborative project. Many people contributed to make this book better. Any errors and omissions are my own despite the barrage of corrections from the from the following people. If I left out your name, please accept my apologies.

To Joanne, thank you for your love, support, knowledge, active participation, and editing skills. This book would not have happened without you.

To Ben and Aidan, thanks for accepting the million just a minute's, for manufacturing thousands of story cards, and for teaching me about life and people in the process. Living with children and practicing XP have much in common.

To Paul Moeller, thank you for being a brilliant business partner, friend, and programmer. Thanks for teaching me why there is no busy work in

nagler@extremeperl.org

programming, and for saying and not saying what you thought about this book.

To Ion Yadigaroglu, thank you for the thousand conversations, your support, and for believing in me. And, for having the courage to leave the programming to the programmers.

To Martin Lichtin, thank you for explaining that every layer of indirection creates a new software problem, and for helping me wade through myriad software problems over the years.

Thanks to the bivions: Danny Ace, Annaliese Beery, Greg Compestine, Eric Dobbs, Eric Schell, David Farber, Justin Schell, and Tom Vilot. You took part in my crazy experiments, listened patiently to my lectures and diatribes, and taught me much about programming, teamwork, and having fun.

Thanks to Johannes Rukkers for teaching me about applications programming in large organizations, and for the many enlightening conversations at the James Joyce and elsewhere.

Thanks to Rob Ward for gallantly giving up your true name at O&A, for years of patient guidance and support, and for smashing through the illogic and unEnglish in the many drafts of this book.

Thanks to Stas Bekman, Justin Schell, and Alex Viggio for pair programming with me in the last chapter. You kept me on task, and helped me avoid complexity.

Thanks to the many other reviewers who contributed the volumes of feedback that made my writing and examples more readable and correct. The reviewers in alphabetical order were: Jim Baker, Kent Beck, Tom Brown, David Bruhweiler, Sean Burke, chromatic, Phil Cooper, Ward Cunningham, Bryan Dollery, Jeff Haemer, Ged Haywood, Joe Johnston, Walter Pienciak, Chris Prather, Lewis Rowett, Michael Schwern, Jeremy Siegel, and Mike Stok.

I'd also like to thank my many mentors over the years: Jon Bondy, Pete Bonham, David Cheriton, Tom Lyon, Jim Madden, Richard Olsen, Andrew Schofield, and Roger Sumner. Bits of your wisdom are contained herein; I hope I got them right.

Finally, thanks and my deepest sympathies to the family of Pete Bonham. You allowed me to enter your lives at such a difficult time. Pete's death was a driving factor in this book, and his life's work guides me regularly.

Life and work are a series of successes. It just so happens that XP makes them more visible, but you have to choose to celebrate them.

Rob Nagler
Boulder, CO, US

nagler@extremeperl.org

Chapter 1

The Problem

> In spite of what your client may tell you, there's always a problem.
>
> – Gerald Weinberg

Software is a scarce resource, in that the demand for software greatly outstrips the supply. We hear about huge shortages of IT staff required to meet to this demand. Costs are rising, too. Some people believe the way we can increase output is to outsource development to places where qualified labor is cheap and plentiful. However, the problem with software development lies elsewhere, and increasing the number of programmers and separating them from the customer only makes the problem worse.

A programmer's job is getting the details exactly right, exactly once. This isn't at all like physical manufacturing where the brunt of the cost is in the process of making the exact copies of a product. Outsourced manufacturing works well, because the details have already been decided in the design phase. The manufacturing process merely replicates this fixed design. With software, the cost of making copies is almost free, and it's the efficiency of design phase that governs its cost. Cheap and abundant labor improves manufacturing efficiency, but this economy of scale does not make software development more efficient.

The cost of programming is directly related to network effects. As you add programmers to project, the communication costs increase proportionally with the square of the total number of programmers. There are that many more links over which the details must be communicated. And, as the customer and programmers drift farther apart, the cost of the most important link increases. Reducing the cost of communication between the

programmers and the customer is crucial to getting the details right efficiently. A time lag along this link multiplies the cost. To improve efficiency, the customer needs instantaneous communication with the programmers, and programmers need immediate feedback from the customer.

This chapter differentiates software development from physical manufacturing. We explain why traditional, plan-driven development methodologies increase project risk, and how fostering failure reduces risk. The chapter ends with a parable that shows the way to reduce both requirements and implementation risk is to bring the customer closer to development.

1.1 Some Statistics

According to the Business Software Alliance, the software industry is growing rapidly to meet a seemingly unlimited demand. From 1990 to 1998, the U.S. software industry's revenue grew by 13% per year.[1]

Despite, or perhaps as a result of this rapid growth, the software industry remains highly inefficient. While sales grew by 18% in 1998, an astounding 26% of U.S. software projects failed and another 46% were labeled as challenged by the Standish Group International.[2] They also estimate 37% of our resources are wasted on failed and challenged software projects.

We need to understand why software development is so inefficient and why projects fail.

1.2 Risk Averse Methodologies

It's not that failure is all bad. People learn from mistakes. However, we don't want to be driving on the mistake bridge builders learn from. We don't want to be flying in an engineering error, to live near a nuclear plant failure, or even to stand near a pancake griddle meltdown.[3]

---

[1] Business Software Alliance, *Forecasting a Robust Future: An Economic Study of the U.S. Software Industry*, Business Software Alliance. June 16, 1999. http://www.bsa.org/usa/globallib/econ/us_econ_study99.pdf

[2] The Standish Group conducted a study of 23,000 software projects between 1994 and 1998. Failed means "The project was canceled before completion." Challenged means "The project is completed and operational, but over-budget, over the time estimate and with fewer features and functions than initially specified." See *CHAOS: A Recipe for Sucess*, The Standish Group International, Inc., 1999.

[3] Our breakfast suddenly turned into splattered, molten metal one Sunday. Fortunately, no one was hurt.

nagler@extremeperl.org

To reduce risk, engineering methodologies are plan-driven. The plans help us ensure we catch mistakes as early as possible. The planning process involves many redundant steps. Emerging plans must pass reviews and consistency checks during the numerous phases of the project. The public is protected by layers of methodology and in some cases government regulations and laws.

Although public safety is certainly a concern, business probably evolved these risk mitigation methodologies for another reason: to reduce the risk of production failures. When you are manufacturing physical widgets, you don't want to find an error after you have produced one million widgets, or even a thousand. The cost of the raw materials plus the time to fix the error, to retool, and to rerun the job is usually high in comparison to the cost of the extra procedures to catch errors during the planning and design phases.

Software development is quite different from manufacturing. The cost of producing the physical software package is nominal, especially considering most software is developed for only one customer.[4] Today, automated updates via the Web further reduce the cost of software delivery. The cost of software production is borne almost entirely by research, development, and maintenance.

While software lacks the characteristics of physical products, we still develop most software with the same implementation risk averse methodologies. We are told "If [a requirements] error is not corrected until the maintenance phase, the correction involves a much larger inventory of specifications, code, user and maintenance manuals, and training material."[5] Mistakes are expensive, because we have "inventory" to update. Plan-driven software development is firmly grounded in avoiding production failures, which slows development in the name of implementation risk mitigation.

1.3 Fostering Failure

Implementation risk mitigation is expensive. The most obvious cost is the bookkeeping material (documents defining requirements, specifications, architecture, and detailed design) in addition to the code we need to maintain.

[4] The Business Software Alliance report estimates 64% of software sales is in the customized software and integrated system design services. This does not include internal IT budgets.

[5] *Software Engineering Economics*, Barry Boehm. Prentice-Hall, Inc. 1981, pp. 39-40. This classical reference is old but unfortunately not outdated, viz., *Get Ready for Agile Methods with Care*, Barry Boehm. IEEE Software. Jan, 2002, pp. 64-69.

Less risk averse methodologies lower the cost of software production. Reducing redundancy in the planning process means there is less to change when a requirements error is inevitably discovered. By not creating inventory in the first place we further reduce our overhead and inefficiencies.

When we improve efficiency in one part of the process, we gain flexibility in other areas. We have more resources and time to correct errors in all phases of the project. The fewer errors, the better the chance the project will succeed.

Implementation risk aversion is costly in other ways. We avoid change later in the project even if that change is justified. The cost of change is proportional to the amount of inventory. In plan-driven methodologies, change is increasingly costly as the project progresses. Not only do we have to update all the bookkeeping material, but it must pass the same manual reviews and consistency checks that were used to validate the existing plan and design.

And possibly the most important cost is risk aversion itself. Failure is a natural part of creation. We don't like to fail, but when we do, we usually learn from the experience. According to management gurus Jim Collins and Jerry Porras, "What looks *in retrospect* like brilliant foresight and preplanning was often the result of "Let's try a lot of stuff and keep what works.""[6]

An interesting side-effect of reducing the cost of correcting errors is that we reduce the risk associated with trying new and innovative solutions.

1.4 Get Me a Rock

Reducing the cost of correcting errors is one part of the problem. One reason projects fail is that they do not satisfy the end-users' needs. To help ensure a project's success, we need to mitigate requirements risk. The following story about a manager and his subordinate demonstrates the difficulty of specifying and satisfying requirements:
Boss: Get me a rock.
Peon: Yes, sir.

...a little while later...
Peon: Here's your rock, sir.
Boss: This rock is all wrong. We need a big rock.

[6] *Built to Last*, Jim Collins and Jerry Porras, HarperBusiness. 1997, p. 9.

nagler@extremeperl.org

...another delay...
Peon: Here ya go, boss.
Boss: We can't use this rock. It's not smooth.

...yet another delay...
Peon: [panting] Smooth, big rock, sir.
Boss: The other rocks you brought were black,
but this one's brown. Get a black one.

And the story goes on and on. We've all been there. Both roles are difficult. It is hard to specify exactly what you want when you're not sure yourself, or even when you are sure, you may have difficulty explaining to another person what you want. On the flip side, the subordinate probably doesn't speak the language of rocks, so he can't elicit what the manager wants in terms the manager understands.

The plan-driven lesson to be learned is: Customers must give precise instructions (specifications). Programmers should not be expected to be mind readers.

1.5 Requirements Risk

Most software projects are as ill-defined as the requirements in this story.[7] The plan-driven approach is to spend a lot of time up front defining the requirements in order to reduce the cost of the implementation. The theory is that planning is cheap, and programming is expensive. Once we get through the specification phase, we can ship the spec off to a source of cheap labor whose job it is to translate the spec into working code. That would work fine if the specification were exactly right, but it most likely is missing a lot of important detail, and the details it identifies probably aren't exactly right either. The Rock example doesn't do justice to the amount of detail involved in software. Large programs contain hundreds of thousands and sometimes millions of details that must be exactly right, or the software contains faults.

The cumulative effect of software faults is what causes projects to fail. It's easy to fix a few faults but not thousands. When users throw up their

[7] On page 310 of *Software Engineering Economics*, Barry Boehm states, "When we first begin to evaluate alternative concepts for a new software application, the relative range of our software cost estimates is roughly a factor of four on either the high or low side. This range stems from the wide range of uncertainty we have at this time about the actual nature of the product."

nagler@extremeperl.org

hands and scream in exasperation, they're saying the program misses the mark by a mile. It's insufficient to tell them the specification was right or that the programmers simply misunderstood it. It's the code users are frustrated with, and it's the code that is just plain wrong.

Planning and specification does not guarantee end-user satisfaction. Plan-driven methodologies ignore requirements risk, that is, the risk that details may be incorrect, missing, or somehow not quite what the customer wants. When we gather requirements, write the specification, ship it off, and only check the program against user expectations at the end, we are setting ourselves up for failure. Requirements change in this scenario is very expensive. This is what we see in the Rock example. The requirements risk is proportional to this time lag. Given the predominance of plan-driven software development, it's likely that a large number of project failures are directly attributable to too little requirements risk mitigation.

1.6 Let's Rock And Roll

Fortunately, there is an alternative version of the Get Me a Rock story, which solves the ill-defined requirements problem with greater efficiency:
Boss: Get me a rock.
Peon: Sure, boss. Let's go for a ride to the quarry.

...a little while later...
Boss: Thanks for pointing out this rock.
I would have missed it if I went by myself.
*Peon:*You're welcome, boss.

The moral of this story is: to increase efficiency and quality, bring the customer as close as possible to a project's implementation.

nagler@extremeperl.org

Chapter 2

Extreme Programming

> XP is the most important movement in our field today.
>
> – Tom DeMarco

Extreme Programming (XP) is an agile software-development methodology. XP helps you remain light on your feet by avoiding unnecessary baggage and by incorporating feedback continuously. Changing requirements are an expected and acceptable risk, because the customer sees the system being developed in real-time. Mistakes are immediately visible and are corrected while the feature's implementation is fresh and pliable, much like a potter reworks clay.

Programmers work and rework the code in XP projects. The customer sees a system grow from layer upon layer of detail. The software is only as effective as the details it embodies. A tax accounting system must round computations correctly, or it can't be used; it's insufficient to get the formula right without considering that taxes are collected in whole currency units. Details matter, and XP programmers reflect back to the customer in the only way that matters: working code.

All this working and reworking requires a stable base and good tools. To throw pots effectively, you need to be seated comfortably at your potter's wheel, and your tools need to be within easy reach. In the world of idea creation, people need comfort, too. They need to know what's expected of them and why. An XP team is expected to follow 12 simple practices. You aren't supposed to execute the practices blindly, however. XP gives us a framework of four core values that lets us adjust the practices to suit our particular project. The four core values are like a comfortable mental chair;

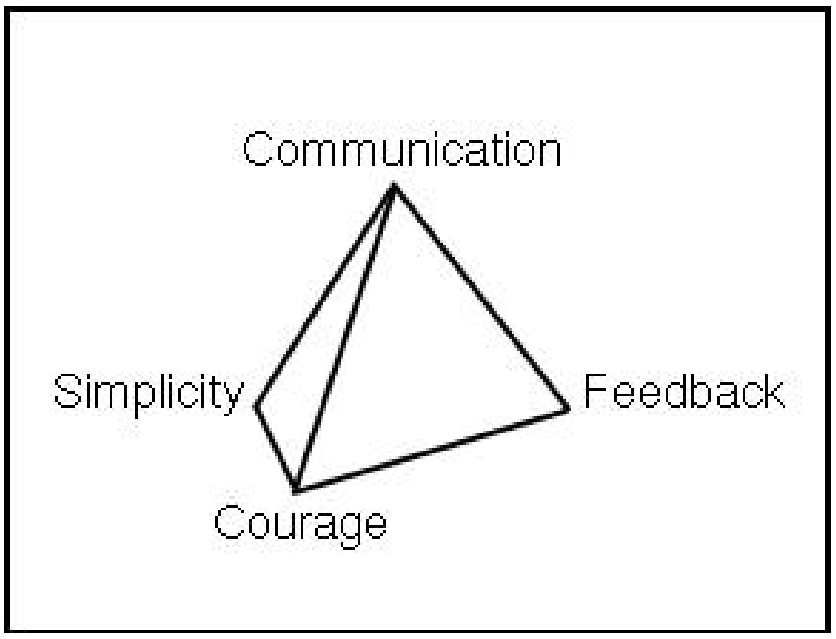
we work our code using the practices with the core values supporting our every movement.

This chapter explains XP's core values: communication, simplicity, feedback, and courage. We then enumerate XP's 12 practices, and discuss how to adopt them.

2.1 Core Values

XP is built on four core values: communication, simplicity, feedback, and courage. The values reinforce each other to form a stable structure as shown in the figure:

Core Values



The four values give the people in XP projects a firm foundation to stand on, the why for the how. Unlike plan-driven software methodologies mentioned in The Problem, XP is people-driven. We value people over process.

The idea of giving people reasons (core values) for what they do (practices) is not new. For example, before XP was a twinkle in Kent Beck's eye, John Young, then CEO of Hewlett-Packard, stated, "We distinguish between core values and practices; the core values don't change, but the

practices might."[1] It's important to trust people to judge the validity of a practice for their particular job. They need a value system to frame their judgments so that a person can change a practice without undermining the goals of an organization.

The values relate to each other to form a framework. Without these relationships, the values would not hold together, and people would be less likely to accept and to work with them. The tetrahedron symbolizes the importance of the bonds between the values. As you read through the descriptions in the following sections, you'll see how the values support each other and the practices.

2.2 Communication

> A little Consideration, a little Thought for Others, makes all the difference.
>
> – Eeyore (A. A. Milne)

Software is developed as quickly as the communication links in the project allow. The customer communicates her requirements to programmers. The programmers communicate their interpretation of the requirements to the computer. The computer communicates with its users. The users communicate their satisfaction with the software to the customer.

Communication in XP is bidirectional and is based on a system of small feedback loops. The customer asks the users what they want. The programmers explain technical difficulties and ask questions about the requirements. The computer notifies the programmers of program errors and test results.

In an XP project, the communication rules are simple: all channels are open at all times. The customer is free to talk to the programmers. Programmers talk to the customer and users. Unfettered communication mitigates project risk by reducing false expectations. All stakeholders know what they can expect from the rest of the team.

2.3 Simplicity

> Pooh hasn't much Brain, but he never comes to any harm. He does silly things and they turn out right.

[1] As cited in *Built to Last*, Jim Collins and Jerry Porras, HarperBusiness. 1997, p. 46.

– Piglet (A. A. Milne)

We all want simple designs and simple implementations, but simple is an abstract concept, difficult to attain in the face of complexities. XP takes simplicity to the extreme with practical guidelines:

- Do the simplest thing that could possibly work (DTSTTCPW),
- Represent concepts once and only once (OAOO),
- You aren't going to need it (YAGNI), and
- Remove unused function.

Do the simplest thing that could possibly work (DTSTTCPW) means you implement the first idea that comes to mind. This can be scary. Rely on your courage to try out the idea. Remember that failure is an important part of creation. It is unlikely the simplest idea is what you will end up with. However, it's also unlikely you can anticipate what's wrong with your simple solution until you try it out. Let the feedback system guide your implementation. DTSTTCPW is simplicity as in fast and easy.

Once and only once (OAOO) helps you maintain your agility by reducing the size of your code base. If you let conceptual redundancy permeate your system, you have to spend more and more time rooting out faults. Every time you copy-and-paste, you take one more step closer to bloatware. Each copy creates an implicit coupling, which must be communicated to the rest of the team. Be courageous, just say no to your mouse. Say yes to refactoring the code for re-use. OAOO is simplicity as in few interchangeable parts.

You aren't going to need it (YAGNI) is a popular and fun expletive. If you can solve the immediate problem without introducing some feature, that's YAGNI! And, you simplified your problem by omission. YAGNI is a corollary of OAOO. If you don't have to implement the feature in the first place, your system just took a step away from bloatware. YAGNI is simplicity as in basic.

Sometimes you add a function for good reason but later find out the reason is no longer valid. At this point you should delete the function. It is unnecessary complexity. It shouldn't require much courage, because the code is still there in your source repository. You can always pull it out if you need it again. Removing dead code is simplicity as in pure and uncluttered.

nagler@extremeperl.org

2.4 Feedback

> Well, either a tail *is* or isn't there. You can't make a mistake about it. And yours *isn't* there!
>
> – Pooh (A. A. Milne)

The more immediate feedback, the more efficiently a system functions. A simple example can be found in the shower. Some showers respond instantly to changes in the faucet handle. Other showers don't. I'm sure you've experienced showers installed by Central Services engineers from the movie *Brazil*.[2] You turn on the shower, adjust the temperature, and hop into a hailstorm or *The Towering Inferno*.[3] After you peel yourself off the shower wall, you adjust the temperature, and wait a bit longer before timidly stepping in again. The long delay in the system makes showering unpleasant and inefficient.

For many customers, this is what software development is like. You request a change, and it is delivered many months later in some big release. Often the change fails to meet your expectations, which means another change request with yet another long delay.

XP is like a well-designed shower. You request a change and out comes software. Adjustments are visible immediately. The customer sees her requirements or corrections implemented within weeks. Programmers integrate their changes every few hours, and receive code reviews and test results every few minutes. Users see new versions every month or two.[4]

The value of immediate, real world feedback should not be underestimated. One of the reasons for the success of the Web is the abundance of structured and immediate feedback from users. Developers see errors in real time, and contract all input and output that causes Web application failures. Users benefit from running the latest version of the software, and seemingly on demand fault corrections. When people talk about the enduring value of the Web in the distant future, I predict they will value the extreme acceleration of user feedback and software releases. The impact of this feedback on quality and development efficiency is what differentiates Web applications.

XP reduces project risk by taking iterative development to the extreme. The customer's involvement does not end at the planning phase, so re-

[2] http://www.filmsite.org/braz.html

[3] http://www.norcalmovies.com/TheToweringInferno

[4] XP's founders recommend multi-week iterations and releases to the customer every three or four iterations. My experience with Extreme Perl is that an iteration per week and one iteration per release works well, too. See Logistics for how to make this happen.

quirements errors are reconciled almost immediately. The system's internal quality is maintained by programmers working in pairs who are striving for simplicity. Automated testing gives everybody feedback on how well the system is meeting expectations.

XP uses feedback to integrate towards a solution, rather than trying to get it through a discontinuity.[5]

2.5 Courage

> It is hard to be brave, when you're only a Very Small Animal.
>
> – Piglet (A. A. Milne)

Fear is a prime motivator, or as Napoleon Bonaparte put it, "There are two levers for moving men: interest and fear." With courage, our relationships take on a new quality: trust. XP helps build the bonds of trust by repeatedly exposing people to small successes.

Courage is required at all levels. Is this solution too simple? Is it too complex? Does this test cover all the cases which could possibly break? Will the programmers understand what I mean by the story? Will we make it to Comdex without a detailed schedule?

We overcome fear, uncertainty, and doubt in XP with courage backed by the other three values. A simple system is harder to break than a complex one. Multilevel, rapid feedback lets us know quickly when our courageous changes fail. Open communication means we don't have to face our fears alone. Our team members will support us. All we have to do is speak of our fears as openly as Piglet did in the epigraph to this section. And, Rabbit finds the right words to support him[6]:

> "It is because you are a very small animal that you will be Useful in the adventure before us."
>
> Piglet was so excited at the idea of being Useful that he forgot to be frightened any more [...] he could hardly sit still, he was so eager to begin being useful at once.

Sometimes we feel as small and ineffectual as Piglet. During these downtimes, it's likely one or more of our team members feel as courageous as

[5] Thanks to Johannes Rukkers for this excellent observation.

[6] All A. A. Milne quotes in this chapter are from *Winnie-the-Pooh*, A. A. Milne, Dutton's Childrens Books, 1994.

nagler@extremeperl.org

Rabbit or Pooh. XP accepts that people's emotions vary, so XP uses team interactions to keep the project stable and to provide emotional support in those inevitable, difficult times.

Courage is a double-edged sword. You needed to overcome your fears, but too much courage can be dangerous. XP uses small steps to promote courage and keep it in check. Team members see a continuous flow of failures and successes. XP uses small, regulated doses to discourage excess and encourage success.

2.6 The Practices

XP's practices embody the values described in the previous sections. In his book *Extreme Programming Explained* Kent Beck defines the 12 practices as follows (quoted verbatim):

The Planning Game Quickly determine the scope of the next release by combining business priorities and technical estimates. As reality overtakes the plan, update the plan.

Small releases Put a simple system into production quickly, then release new versions on a very short cycle.

Metaphor Guide all development with a simple shared story of how the whole system works.

Simple design The system should be designed as simply as possible at any given moment. Extra complexity is removed as soon as it is discovered.

Testing Programmers continually write unit tests, which must run flawlessly for development to continue. Customers write tests demonstrating the features are finished.

Refactoring Programmers restructure the system without changing its behavior to remove duplication, improve communication, simplify, or add flexibility.

Pair programming All production code is written with two programmers at one machine.

Collective ownership Anyone can change any code anywhere in the system at any time.

nagler@extremeperl.org

Continuous integration Integrate and build the system many times a day, every time a task is completed.

40-hour week Work no more than 40 hours a week as a rule. Never work overtime a second week in a row.

On-site customer Include a real, live user on the team, available full-time to answer questions.

Coding standards Programmers write all code in accordance with rules emphasizing communication through the code.

These 12 simple practices realize the four core values. The remainder of this book explains how to implement the practices in detail. Before we get into implementation, let's briefly discusses how you might adopt XP in your organization.

2.7 Adopting XP

Organizations have their own way of doing things. There are practices, processes and probably even some core values. In order to adopt XP, you'll need to work within this framework, and accept the status quo. One way to start is to use XP in a project composed of volunteers. The project may even be important, which is good, because your success with XP should be visible. Once that project succeeds, pick another and let another team coalesce around it, possibly including a few members, but not all, from the original XP team.

You may not be in a position to pick and choose projects, but you probably have some latitude in how you work on a daily basis. If this is the case, try selecting a few practices that align with your existing methodology. For example, if your organization values testing, try test first programming. Or, organize your day around stories. I find this technique to be helpful for non-software problems, too, as you'll see in Release Planning. Keep the stories on index cards, and work through them serially. Check off each card as you complete it.

Once you see your productivity go up, discuss the practices you found successful with a co-worker or your manager. Use the core values as your guide. You'll need courage to start the communication. Keep your explanations simple, focusing on the practice, not the whole of XP. Be open to feedback, and incorporate it immediately. And, as always in XP, take small steps and iterate.

nagler@extremeperl.org

As you read through this book, look at the practices from your organization's perspective. You'll see plenty of ways to integrate them. XP is an evolutionary methodology that can be adopted incrementally and organically. Even if you are the head honcho, don't install XP with a Big Bang. Organizations and people have their own natural pace of change. XP cannot accelerate this rate of change overnight. Change cannot be mandated. Rather XP values feedback and communication to allow you to measure your progress and to integrate change continuously.

nagler@extremeperl.org

nagler@extremeperl.org

Chapter 3

Perl

> Perl is a language for getting your job done.
>
> – Larry Wall

Perl is a dynamic, object-oriented, interpreted, applications programming language with a full complement of security features, syntax-directed editors, debuggers, profilers and libraries. You can also write scripts in it. Perl lets you program any way you want, and XP helps you choose which way is the most effective for your project.

This chapter discusses how Perl shares similar values with XP, and how the Perl culture uses XP programming practices. Finally, we note why XP is needed to organize Perl's exuberance.

3.1 Core Values

Perl and XP share a similar set of values. It's rare for programming languages and methodologies to define values at all, so this basic fact puts Perl and XP in a unique category. We discussed XP's values in Extreme Programming, so let's compare Perl's core values to XP's:

- Laziness means you work hard to look for the simplest solution and that you communicate efficiently. You don't want to misunderstand what someone said which might cause you to do more work than you have to.

- Impatience encourages you to do the simplest thing that could possibly work. You use the code to communicate your understanding of the

problem to the customer, because you don't like sitting through long, boring meetings.

- Hubris is courage born from the fear your code will be too complex for others to understand. Hubris makes you strive for positive feedback and to react quickly to negative feedback from your peers, the computer, and the customer.

Larry Wall, Perl's inventor, calls these values the "three great virtues of a programmer".[1] They tell us why Perl is the way it is: a language that grows organically to meet the demands of its customers.

3.2 Customer-Orientation

Perl and XP were invented in the trenches. Larry Wall had to produce reports for configuration management based on netnews.[2] Kent Beck was tasked with saving the Chrysler Comprehensive Compensation project. Neither Perl nor XP were designed in the ivory towers of academia. Both XP and Perl were developed to solve a specific problem, and quickly so that Kent and Larry would keep their jobs.

It's too early to tell with XP, but Perl has remained true to its roots. Perl continues to evolve based on feedback from its customers: the Perl community. Features are added or changed if enough people clamor for them.

This smorgasbord approach to programming languages is non-traditional, much like XP's focus on people over process is non-traditional in the development methodology community. Perl gives you a wide variety of tools without constraining how you use them. For example, Perl is object-oriented but objects aren't required to create modules or scripts. Perl programmers aren't forced to encapsulate all their code in objects to solve every problem, especially when simpler alternatives exist.[3] XP asks you to ignore what you aren't going to need, and Perl lets you put this principle into action.

---

[1] *Programming Perl*, 3rd Edition by Larry Wall et al, p xix.

[2] This wonderful story of doing the simplest thing that could possibly work is elaborated in *Programming Perl*, p. 646.

[3] See Object-Oriented Programming in It's a SMOP for an example of when objects obfuscate.

nagler@extremeperl.org

3.3 Testing

Another parallel between XP and Perl is testing. The Perl culture is steeped in testing, and in XP, we test to get feedback from the computer and the customer. Perl comes with a complete test suite for the language, and virtually every CPAN (Comprehensive Perl Archive Network) module comes with unit tests. Testing in Perl is about impatience. It's faster to get accurate feedback from a unit test than by firing up the whole system to see that it works.

It's easy to write tests in Perl. The software quality assurance community has long known this, and Perl is a favorite tool among testers.[4] Perl programmers are lazy, so sometimes on CPAN the only useful documentation for a package is its tests. Some programmers find it's easier to write tests than documents, and conveniently that's the XP way of doing things anyway. Unit tests communicate exactly how to use an API, and acceptance tests demonstrate correctness and progress to the customer.

3.4 CPAN

In XP, we share code in a collective repository. CPAN is probably the largest collection of Perl code on the Internet. There are over 3500 open source packages available for download. It's also the official repository for the Perl core implementation. Any perl programmer can get the latest version of Perl, and perl developers check in their changes directly to CPAN. The Perl community shares openly, because they're lazy, and want to solve problems once and only once.

Hubris plays a role on CPAN and in the Perl community in general. For example, type in `xml parser` into http://search.cpan.org, and you'll get at least six pages of results. It took some time for us to find the right XML parsing package to use in It's a SMOP. CPAN, like Perl, offers you many more ways to do it than any one project needs.

3.5 Organizing Your Workshop

Perl is often called a Swiss Army Chainsaw. When you add in CPAN, it's really more like the world's largest selection of hardware.[5] You can get lost

[4] Visit http://www.stickyminds.com, and search for Perl.

[5] McGuckin Hardware in Boulder, Colorado (http://www.mcguckin.com) is the physical world equivalent of Perl.

in the dizzying array of choices. Some programmers find Perl daunting.[6]

This is where XP comes to the rescue. XP is the organizer in the Extreme Perl marriage that complements Perl, the doer and fixer. XP's role is to keep Perl from fixing the car when the kids need to be put to bed. XP gently guides Perl to do the right thing for the customer.

The next few chapters show you how Extreme Perl is organized, and the second half of this book shows you know Extreme Perl gets things done.

[6] In a presentation at the 2002 PHP Conference, Michael J. Radwin lists Perl's Motto, "There Is More Than One Way To Do It", as one of the three reasons Yahoo! was moving away from Perl. Visit http://public.yahoo.com/~radwin/talks/yahoo-phpcon2002.htm for the complete presentation.

Chapter 4

Release Planning

> Man has an intense desire for assured knowledge.
>
> – Albert Einstein[1]

We plan to communicate our intentions and to prepare for the unplanned. In XP, the plan evolves with feedback from building and using the system. We adjust our intentions as we see our ideas being realized. Sometimes reality matches what we want. Other times we gain new insight and change our plans accordingly. XP ensures we get our recommended daily allowance of reality checks.

Planning is a three part process in XP:

Release Planning We decide on the course of action based on customer desires and team capacity in the planning game. The result is the release plan: a list of stories (requirements) prioritized by business value. A release encompasses the next one to two months of work.

Iteration Planning We divvy up the stories based on individual programmer desires and capacity in an iteration planning meeting. An iteration plan is a prioritized list of development tasks estimated by the people who will be implementing them. Iterations are delivered every one to three weeks.

Pair Programming We continuously balance between improving internal quality and adding business function based on peer-to-peer discussions and individual task commitments. A pair programming session lasts

[1] *Ideas and Opinions*, Albert Einstein, Crown Publishers Inc., 1985, p. 22.

a few hours. The output is one or more unit tests and the software that passes those tests.

This chapter covers release planning. The subsequent chapters discuss the other two activities.

4.1 Planning Game

To start the ball rolling, the customer and programmers sit in a room together to define the requirements, priorities, and deadlines for the release. We call this the planning game.

Planning is an interpersonal process, a game, if you will. The customer wants a long list of features, and the team wants to implement them all. Unfortunately, the team is limited in what it can accomplish in one release. In XP, the planning game defines the rules, pieces, and roles. The objective is clear communication of business objectives and technical constraints.

4.2 Roles

In XP, we speak about the customer as a single person, although several people may fill this role simultaneously. It's important that the people acting as the customer speak in one voice. This ensures that the project adds business value optimally–to the the best of the customer's knowledge. Obviously, no one knows the optimal path, that's why XP encourages tight feedback loops. However, the team cannot be divided in its goals, or the project will fail. XP has no mechanisms to resolve disputes between the people acting as the customer.

The role of the customer is played by people who represent the collective business interests of the project. For example, product managers, users, and clients act as the customer. The business requirements, priorities, and deadlines are defined by the customer.

XP defines the roles of customer and programmer unambiguously. The customer's job is to identify *what* needs to get done, and the programmers' job is to explain *how* it will get done. The customer is not allowed to say an estimate is wrong or to contradict a technical statement made by the programmers. And, conversely, programmers cannot say a feature is unnecessary or the customer has her priorities wrong. The planning game is based on mutual respect for these equal and distinct responsibilities.

If you are new to XP, you may want to have an impartial coach participate in your first few planning games. The planning game is non-adversarial,

nagler@extremeperl.org

and a coach may help to remind people that planning is about getting as accurate a picture as possible given limited data.

4.3 Stories

A story is how the customer communicates a requirement in XP. The content of a story is simple, usually one or two sentences. Here are some actual stories:

GenderTags The questions and answers will be customized with the name of the bride and groom where applicable.

InstallDisks Figure out which computer to use, and install disks. Make available via Samba and NFS.

SubscriptionStartEnd Subscription start/end dates should not overlap.

DemoClubUpdate Update data in demo club. Include AccountSync entries.

DBExportSplit file_t is too large for a single export file. Need to figure out how to split.

ECtoQB Convert EC payments to QuickBooks IIF format. Group checks into deposits.

CloseChase Move account to Wells Fargo.

For the most part, the customer writes the stories. Note the incompleteness of the stories in some cases. For example, DemoClubUpdate doesn't specify what data is required. If the programmer needs more info, he will ask the customer when he starts implementing the story.

Some of these example stories were not written by the customer, for example, DBExportSplit and InstallDisks. Technical stories come up during implementation, but it is the customer who decides how important they are. The programmers' job is to note the technical need on a story card, and to present the business case to the customer during the planning game.

4.4 On-site Customer

The customer does not write the stories and say, "get to it!" There is an explicit trade-off with simple requirements; the customer must stick around

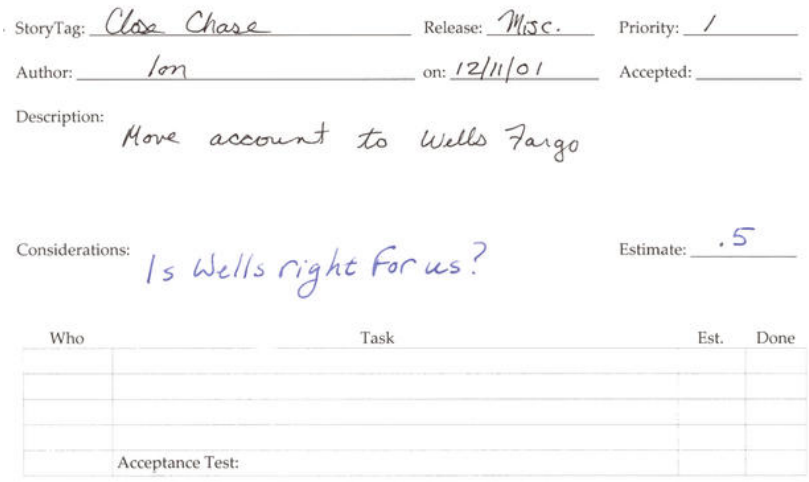
for the implementation. She must be available to the programmers to fill in the details during execution of the plan.

A story is not sufficient criteria for acceptance. Part of the customer's job is to write the acceptance tests with the help of the programmers. Perl makes this easy. Acceptance Testing explains how in detail.

4.5 Story Cards

The stories are written on story cards. Here's the CloseChase story card:

Close Chase Story Card



I use story cards for almost everything. In this case, it's an administration problem. The mechanism is always the same: when there is a problem to solve, write it on a story card. It represents a work item, and all work items need to be put in the queue.

It often is a good idea to prepare story cards before the planning game. Everybody should keep a stack on their desks. The customer may think of new stories while using the system. The programmers may encounter internal quality problems while coding.

There is no officially-sanctioned XP story card. Each project may need special customizations. This story card is one we have developed for our company's needs.[2] Here's the way we interpret the fields:

[2] The source can be found at http://www.extremeperl.org.

StoryTag This is a mixed-case identifier for the story. Giving a handle to the story simplifies communication about it. In my experience, customers tend to have trouble filling in this field in the beginning. Programmers are used to naming things (subroutines, modules, etc.), and they may be able to help the customer choose appropriate story tags.

Release The date or name of the release. We use this field loosely to categorize stories by release or project. We couldn't predict when CloseChase would be done, so we just put it in the Misc category. Some teams work on several projects simultaneously, and this field helps keep the story cards organized.

Priority After the customer physically orders the story cards, she writes their numeric priority in this field. The numbering scheme is arbitrary. Sometimes you need to insert a story card in the middle of a sequence. We add a letter, for example, 5a, or use a dotted decimal notation, 5.1. If you drop the story cards, it's important to be able to get them back in priority order.

Author The person who wrote the story. You then know whom to ask for the details.

on The date the story was written.

Accepted The date the story's implementation is accepted by the customer. We do this at the start of every planning game before we get into the new stories. This helps remind the customer what has just been accomplished, and, sometimes, lets the customer catch incomplete stories.

Description A few sentences and/or pictures explaining the requirement. Be brief. If the description isn't clear, rip up the card and start over. This is the most important field, so you may want to fill it in first. See examples in Stories.

Estimate The programmers write their best guess for the implementation time. See discussion about the time scale in Estimation.

Considerations The programmers list the implementation risks and prerequisites in this section. This gives the customer more information about the confidence of the estimate and helps her order dependent

nagler@extremeperl.org

stories. Don't implement the story here. It's tempting to write pseudocode or to outline the design. Save the design for when you are in front of a computer with your programming partner.

Task The list of activities required to implement the story. If the list of tasks grows to more than fits in a few lines, you probably need to split the story. The original story estimate is probably wrong, too. We try to avoid large, complex stories to ensure the software pipeline stays full, and XP's many feedback loops remain active. This field is filled in during iteration planning, which is covered in Iteration Planning.

Who The person who is responsible for this task. Sometimes we add the partner who helped implement the task. It can be useful when trying to remember why the code was written in a particular way.

Est. The estimate for this task.

Done The date the task is finished. This field can also be used to record the actual time to implement the task. We use it to help audit our billable hours.

Acceptance Test This field reminds everybody that stories should have acceptance tests. Ideally, the customer should be responsible for this task, but a programmer probably ends up doing most of the work. See Acceptance Testing for more detail.

4.6 Dead Wood

One important characteristic of story cards is their physical implementation. They are real dead plant matter. You write on them with ink, not electrons. PDAs and laptops are not recommended for planning meetings.[3]

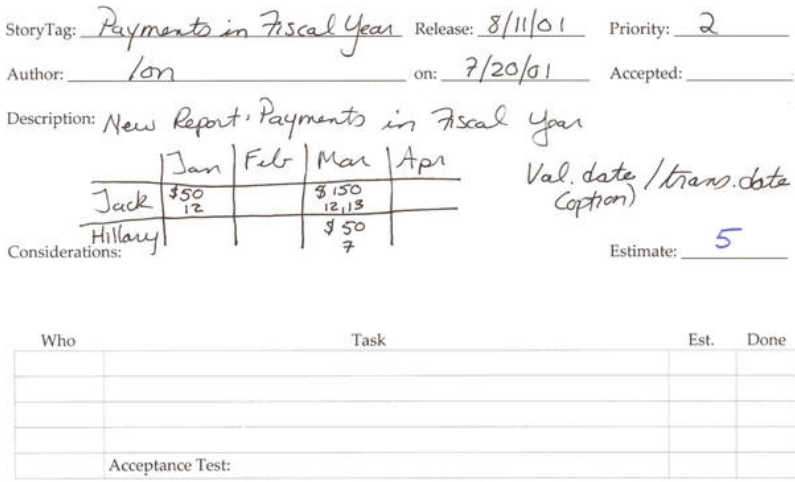
The planning game is an interpersonal communication process. Story cards are a simple communication medium, which enables people to have as much face time as possible. You can read the cards from any angle, in almost any lighting condition.

[3] XP assumes the entire team, including the customer, share the same office. If the customer or part of the team are not colocated, you may want to scan your story cards after the planning game and store them in the collective repository. I'm not a fan of purely electronic story cards, but some people like them. As with all things XP, try it by the book and then season to your taste.

nagler@extremeperl.org

The stories listed so far have been text. Sometimes text won't do. Paper is also an excellent medium for creating pictures on-the-fly. Here's another example that demonstrates the versatility of story cards to capture the story:

PaymentsInFiscalYear Story Card


StoryTag: Payments in Fiscal Year Release: 8/11/01 Priority: 2

Author: Jon on: 7/20/01 Accepted:

Description: New Report: Payments in Fiscal Year

	Jan	Feb	Mar	Apr
Jack	$50 12		$150 12,13	
Hillary			$50 7	

Val. date / trans. date (option)

Considerations:

Estimate: 5

Who	Task	Est.	Done
	Acceptance Test:		

And, here's the final user interface:

PaymentsInFiscalYear User Interface

Member Contributions and Withdrawals from 01/01/2001 to 12/31/2001

Payments grouped by transaction month, with valuation dates. (show units)

Name	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec	
Bartlett, Janine	600.00 1/10						600.00 7/8			600.00 10/20	(3,000.00) 10/20		(1,200.00)
Caesar, Rose	600.00 1/10			600.00 4/15			600.00 7/8			600.00 10/20			2,400.00
Chase, Sam	600.00 1/10			600.00 4/15			600.00 7/8			600.00 10/20			2,400.00
Rodriguez, Tam	600.00 1/10			600.00 4/15			600.00 7/8			600.00 10/20			2,400.00
Sherman, Rollie	325.00 1/10			400.00 4/15						1,215.00 multiple			1,940.00
	2,725.00	0.00	0.00	2,200.00	0.00	0.00	2,400.00	0.00	0.00	3,615.00	(3,000.00)	0.00	7,940.00

The simple picture was enough to get the point across.[4]

[4] Cards are context sensitive, so you may not understand the details of the PaymentsInFiscalYear story or implementation. The point is to demonstrate how little information you need on a story card, and how closely a quick sketch matches the final implementation.

nagler@extremeperl.org

4.7 Estimation

After a customer completes her portion of the story card, the programmers can estimate its implementation, and note any special considerations, such as, new hardware requirements or prerequisite stories. Estimation is an art, where consistency is much more important than precision or accuracy. The customer uses them to assess the relative cost of stories to aid prioritization. Programmers use estimates to compute and compare velocities (velocity is the sum of estimates for an iteration–see Velocity for details).

Stories are estimated in ideal programming days, not real-time. It's hard enough to forecast a story's implementation without considering other factors such as, hardware failures, sick time, or activation of military reservists. Not to mention unforeseeable refactorings required to accommodate the implementation or changes requested by the customer as the implementation evolves. Ideal day are the way we avoid thinking about such factors that we can neither influence nor estimate.

Don't be afraid of making incorrect estimates. By definition, estimates are forecasts made with incomplete information. You should be concerned when estimates end up matching reality exactly, because it's likely due to the effects of Parkinson's Law: work expands so as to fill the time available for its completion.

Plan-driven methodologies often succumb to Parkinson's Law. Programmers are rewarded for making or beating their estimates. Over-estimation is a great way to make sure this happens. In XP, if you over-esimate, you simply get more work to do. You don't get to go home early, because the length of an iteration is measured in real-time, not ideal time. There is always more business value to add, and the code can always be improved through refactoring. Other XP feedback mechanisms, such as, small releases, pair programming, and continuous integration, also help to combat Parkinson's Law.

So don't worry about Parkinson, the weather, or politics. Just write your best guess as to how many ideal days the story will take to implement, and move on to the next card.

4.8 Easing Estimation

Sometimes you can't come up with a number that you have any confidence in. The typical problem is that the story is too large, and it needs to be split up. As a rule of thumb if the story comprises more than one week's

nagler@extremeperl.org

worth of work, decompose it into smaller stories.

For example, this is a story that was too large for me to estimate:

> Create a web site with the ability to upload Microsoft Word documents by me and viewed by anybody. The documents are categorized by visitor (consumer, broker, etc.).

With the help of a customer, we split the story into four smaller, more easily estimable stories:

- Create the web site home page and navigation scheme.
- Implement pages with Word document links and hand-installed documents.
- Allow site administrators to login.
- Implement upload page for Word documents accessible only by administrators.

We were able to deliver business value on the first story. After the second story, the customer had a functional site from the end-user perspective. We were able to populate the site using non-web-based file transfer tools. The last two stories gave the customer full administrative control of the site, which is business value, but not as important as having a site at all. Most stories can be broken down this way. It's quite similar to decomposing a story into tasks (see Iteration Planning).

4.9 Spike Solutions

Some stories defy estimation or decomposition. The programmers knows so little about the problem domain that they may not even know if it is feasible to implement the story. Some fundamental research is required. In XP, we call this a spike solution. Two programmers prototype, mockup, or just explore the problem space in detail for a short time.

System integration is often important to spike. There are usually so many variables that are out of the programmers' control. For example, one project we were involved in required us to download transactions from financial websites. Our first attempt was to use the Open Financial Exchange (OFX) protocol. It turned out that OFX is not very open, and we hit a non-technical roadblock. In parallel with talking to the OFX consortium, we also spiked a web crawling solution for downloading transactions from

financial websites. We quickly learned that the algorithmic complexity was not in the transaction download component but in the backend–integrating the transactions into an existing accounting database. This was an unexpected and important discovery. The spike solution gave us the information we needed to estimate our stories.

The spike solution should be estimated, too. This caps the time the customer must pay for experimentation. The goal is to determine feasibility and cost, not to implement a fully working solution. Spike solutions are thrown away. The result of a spike solution is a new story or stories, which can be estimated with confidence.

4.10 Prioritization

Once the stories that can be estimated have been estimated, the customer groups and prioritizes them. The physical ordering process requires a large table so the customer can see as many of the cards as possible. The customer groups the stories in two piles: this release and subsequent releases.

The size of the release depends on the customer, although you should avoid releases exceeding a few months. The team will divide this release into iterations that last a few weeks at most (see Iteration Planning). From the customer's perspective each iteration is a partial but usable software distribution. The size of a release is therefore a bit of a fuzzy concept. The customer may choose to stop work on a release before all the stories are implemented, and shall still have a working system albeit with fewer features than planned.

This is why prioritization is so important and can't be left to the programmers. The customer should see a continuous flow of software distributions in order of decreasing business value. XP eschews big bang releases. A working end-to-end system is available very early on in the implementation of a release plan.

For this reason, programmers should avoid creating too many dependencies between stories. It's all too easy to control the prioritization through unnecessary linking. If you find yourself saying, "We need to build the infrastructure for this story." Remember XP's guideline: you aren't going to need it (YAGNI). Infrastructure evolves from refactoring as you discover what parts of the system need it.

When the planning game is over put the subsequent releases pile into storage until the next planning game. The team may get done early in which case pull them out mid-release and add enough stories to meet the

nagler@extremeperl.org

deadline for the current release. The nice thing with XP is that you can add or rearrange stories at any time. The stories that are already implemented have already been tested within the context of a working system. There's no end-of-release panic to integrate the parts into the whole. Indeed, the system probably is already in use well before the end of the release.

4.11 All the Facts

The customer's job is hard. She is managing other factors besides implemention complexity, such as, time to market and resource acquisition. To simplify the planning process, she has to have all the data from the programmers. Any work which consumes time and resources needs to be on the planning game table.

The considerations of stories may be significant. If the story requires a relational database to be implemented, you need to make this clear. Or, it may be obvious that a story requires major code restructuring. If so, get it on a card.

While you're coding, you'll notice problems. You may have to copy-and-paste, because the refactoring required to make the pasted code reusable may be lengthy.[5] Write down the problem on a story card and bring it to the next planning game or iteration planning meeting. The customer and management are made aware of internal quality issues before they get out of control.

4.12 Small Releases

Release planning is the way we communicate business objectives to the programmers and enable programmers to communicate technical constraints. The planning game is a simple forum for effective communication that relies on interactive feedback.

By limiting the length of releases to one or two months, we limit the length of planning meetings. This means we have fewer objectives to discuss. It is easy to maintain the cohesion of small releases and to identify dependencies and other potential problems. XP simplifies plan digestion by encouraging us to take a small bite and chew it thoroughly before taking the next bite.

[5] Lengthy to me means anything that takes an hour or more. You may have completed several refactorings already, and any more will blow your budget. For a complete discussion on refactoring, see Refactoring.

nagler@extremeperl.org

Chapter 5

Iteration Planning

> Plans are useless, but planning is indispensable.
>
> – Dwight D. Eisenhower

An iteration adds function to a system without breaking it. The new features are delivered to the customer in a complete software distribution. This way the customer can try out the story implementations, and the programmers can get feedback on the system every few weeks, before the entire release is implemented.

While release planning sets the pace for the customer, iterations are the heartbeat of the system's implementation. By distributing the software on a fixed schedule, typically from one to three weeks, the programmers stay focused on delivering complete stories. This avoids the situation where the system remains "95% done" until the entire release has been implemented. XP programmers complete the most important story first before moving on to the next one.

This chapter covers dividing stories into tasks during the iteration planning meeting, estimating tasks, grouping tasks into iterations, and tracking team and individual velocity to schedule the next iteration's work.

5.1 Tasks

A task is the basic unit of work of an iteration. The stories are broken down into a list of tasks during the iteration planning meeting. For example, here are the tasks for two of the stories from Release Planning:

PaymentsInFiscalYear

- Mockup report.

- Implement year at a glance widget.
- Implement data model and report.
- Acceptance test: validate payments in months for members.

CloseChase

- Compare rates and services at local banks.
- Open new account.
- Move automatic payments (FedEx, EarthLink)
- After last check clears, transfer remaining funds.
- Acceptance test: verify account is closed and automatic payments working.

For reasons of simplicity, I write the tasks directly on the bottom of the story cards. This keeps the motivation for the work (the story) and the list of work items on one piece of paper. This helps me stay focused on the trees without getting lost in the forest.

Alternatively, you might create separate task cards, one for each work item. This allows you to divide up the work for an individual story among several programmers. If you tend towards larger, multi-week stories, task cards are probably the way to go.

5.2 The Meeting

Iteration planning involves the entire team. Somebody reads the stories aloud, and everybody contributes to identifying the tasks, which are written on whiteboards or flip charts for all to see. Another person writes the tasks down on the cards once everyone agrees they are correct.

The meeting is a time for group design discussions. It isn't always obvious what tasks are required to implement a story. While ad hoc design discussions pop up during programming, iteration planning is a time for the whole team to discuss the system's implementation. You might realize that a global refactorings need scheduling. This may add dependencies and possibly a new story for the customer to prioritize.

After all the tasks are listed on the board, people commit to implementing them. Individuals take the cards and estimate the tasks they are planning to implement. As with stories, estimate in ideal programming days. Everybody's ideal timescale is different, don't expect the task and story estimates to add up.

If a task is difficult to estimate, ask for help. Others may have a better feeling for the problem and/or implementation. Sometimes a task is too

nagler@extremeperl.org

complex, and you'll realize as a group that the risk factors were underestimated. Restructure the task list for this story. If you think this story's estimate is way off, discuss it with the person responsible for the story estimate. It may need to be changed, and the customer should be informed of the unforeseen considerations and the new estimate.

5.3 Get Me a Bucket

In plan-driven methodologies, the amount of work that goes into software distribution is dictated by the plan. The schedule is extended as you experience the actual implementation time for the work items. A software distribution only comes out when the entire release is complete. XP takes another view. Software distributions flow out from the team in fixed time iterations, much like water flowing through a water wheel.

The buckets on a water wheel[1] are a fixed size, too, and only one bucket is necessary to get the water wheel turning. If the bucket is too large, the wheel won't turn for awhile, and the water will flow in fits and starts. Tiny buckets don't supply enough energy to move the wheel at all. There's a reasonable range of sizes for efficient transfer of potential to kinetic energy. Experience suggests that the most efficient range for XP iterations is one to three weeks. Each team must decide, however, what works best.

5.4 Velocity

The rate at which the water flows through a water wheel is measured in cubic feet or meters per second. The implementation flow rate in XP is called velocity, and is the ratio of estimated ideal days to real implementation days. Since the number of real days is fixed per iteration, we simply keep track of the sum of the estimated ideal days for each iteration. The ideal time budget for the next iteration is the velocity of the iteration just completed.

Our goal in XP iteration planning is to predictably fill an iteration with exactly the right number of stories. This reduces planning time to a minimum, and keeps the software process flowing smoothly and reliably, just like an efficiently designed water wheel.

Velocity is a relative measure, however. You can only compare like estimates. The team's velocity is the sum of estimates for completed stories per iteration. Likewise, your individual velocity is measured in your own

[1] Overshot water wheels with horizontal axles are the subject of this analogy.

estimation timescale, the sum of your task estimates completed per iteration. You can't compare the velocities of different individuals, and it doesn't matter if the sum of task estimates doesn't add up to their story estimate. It's like comparing water wheel flow rates with the rate at which people flow on an escalator. The denominators are the same, but the numerators are different.

Your velocity should converge after the first few iterations. If it doesn't, something's wrong, and in Tracking, I discuss what to do about it. The starting budget for the first iteration is a guess. It's likely to be way off. At first, it's reasonable to assume the ratio of ideal days to real days is 1:1. Scheduling the first iteration is just the beginning. You'll find you and your team's natural velocity converges rapidly no matter what the starting point is.

5.5 Watch Your Speed

Velocity is a self-governing speed limit. The team limits the sum of its story estimates to the velocity of the prior iteration. The sum of your task estimates should not exceed your individual velocity. That is, you should sign up for the same amount of estimated work that you completed last iteration.

If you signed up for too many tasks, look for somebody with excess capacity. Pick a simpler story if no one volunteers. We avoid splitting stories across iterations, because it complicates tracking. All stories will be implemented by release end, so you'll be able to pick the story you dropped next iteration, if it still interests you.

5.6 Customer Priorities

The priority on the story card expresses the customer's expectation of business value. Programmers work on the highest priority stories in each iteration. In XP, we deliver as much business value as early as possible in the release.

There's no guarantee all stories can be implemented. The best way for the customer to ensure a particular feature is added is to limit the scope of the release. If the programmers get done early, you can add stories in a mid-release planning game followed by an iteration planning meeting. And, if you run out of work before the end of the iteration, grab the highest

nagler@extremeperl.org

priority story card, estimate the tasks, and implement them. This is how you increase your individual velocity.

5.7 Taking Care of Business

XP was originally invented to solve an internal software development problem. The idea of light requirements is more easily accepted in internal projects than it is for consulting projects.
You can still use XP if your relationship with the customer is contractual. My consulting company takes three approaches:

- We write small, fixed-price contracts after the planning game. We scan the story cards and include them in the contract.
- We write a general contract which promises fixed length iterations for a fixed price. The planning game happens before every iteration.
- We write a time and materials contract, and the customer provides stories on an as needed basis.

The three cases are listed in order of increasing trust. We use the first option with customers who are skeptical about our development process. To help with this, we absorb the risk with a fixed price iteration with a satisfaction guarantee. We know that the sooner we get coding, the lower the risk of dissatisfaction. And, we bypass many hours of unpaid work by circumventing protracted negotiations on a detailed, legalistic specification.
After the first iteration, we usually slip into the second option. The customer has seen a working end-to-end system, and she has experienced our process. Functioning software has a way of convincing even the most skeptical customers about XP.
Finally, we move into time and materials mode. By the second or third iteration, XP has established a strong trust relationship due to its intense focus on the customer and her priorities.

5.8 The Beat Goes on

From the first iteration to the last, an XP team cranks out a steady stream of software distributions. Programming, like many other creative professions,

nagler@extremeperl.org

needs a regular beat.[2] It's all too easy to get distracted by an interesting problem. Regular, high frequency deadlines help people to stay on track.

Every team has to find its own rhythm. Weekly iterations work well in my experience, and they dovetail nicely with the 40-hour week practice of XP. At the end of the week, we go home with a clean plate, and come back with a fresh mind for the next iteration.

Whether you deliver every week or every few weeks, the goal is to keep the software water wheel flowing to the customer. She sees steady progress, and programmers get high frequency feedback about the quality of the system.

[2] I had a music teacher who believed this, too. He used to thwack my leg with his baton to make sure I didn't forget the beat. I recommend a kinder, gentler approach for your team. Although some in XP advocate a RolledUpNewspaper, visit the XP Wiki at http://c2.com/cgi/wiki?RolledUpNewspaper to learn more.

nagler@extremeperl.org

Chapter 6

Pair Programming

> When you're stuck together like this, I figure small differences in temperament are bound to show up.
>
> – Robert Pirsig[1]

The last planning activity in XP is pair programming. Two programmers discuss what they are about to program. They write a unit test which is a formal specification. Only after their intention has been communicated to each other, do they begin the implementation.

Pair programming is probably the most distinctive practice in XP. It qualitatively differentiates XP from other software methodologies. Pair programming is also the hardest practice to integrate. The payoff is that pair programming will elevate your system's quality to a new level. And, surprisingly, your team's overall efficiency will improve, too.

This chapter explains value of pair programming, how it works, and ways to adopt the practice. A number of the human factors surrounding solitary and pair programming are also covered.

6.1 Quality

We want to write quality software. Yet there are no rules that guarantee quality. For example, some people think applying the once and only once (OAOO) principle everywhere is a way to quality. Others think OAOO is not always applicable.

[1] *Zen and The Art of Motorcycle Maintenance*, Robert M. Pirsig, Bantam Books, 1989, p. 40.

Yet we know quality when we see it. Two programmers often agree about the quality of a solution, but they probably won't agree on all the characteristics that contribute to or detract from its quality. It's particularly hard to assess the quality of the code as you are writing it. We can recognize quality, but it is not feasible to encode how to achieve it in all circumstances.

Your programming partner can tell you about the quality of what you are writing. He acts as an alter ego. With gentle reminders about the team's coding standards or even fuzzy statements like "this doesn't feel right", your partner improves the code before you invest too much energy into it. When there is a disagreement about quality, you discuss it. There are no rules. Your different experiences guide you.

Simply put, pair programming yields better software than solitary programming. Or, as Eric Raymond says, "Given enough eyeballs, all bugs are shallow."[2]

6.2 How It Works

Pair programming is two programmers sitting at one computer working on the same task. One person drives, and the other kibitzes.[3] They switch roles naturally when the driver gets stuck or when the kibitzer wants to express his thoughts as code or just whenever it's time to switch.

One programmer codes at a time. Just as the customer is supposed to speak in one voice, a programming pair does the same. XP's coding standard practice eliminates discussions about minor details, such as, how to align the statement braces.

The observer thinks strategically about the code with respect to the whole. He's likely to see more of the forest than the driver. For example, the observer may be thinking of additional test cases as the code is being written or seeing a pattern that was used elsewhere and should be carved out so it can be reused.

The communication between the driver and the onlooker is continuous. The feedback from the onlooker helps correct small errors in real-time before they become big problems. The driver and onlooker code in unison with the ultimate goal of not remembering who wrote which line of code. This won't

[2] The Cathedral & the Bazaar, Eric. S. Raymond, O'Reilly and Associates, 2001, p. 30. Available online at http://www.catb.org/~esr/writings/cathedral-bazaar/. Thanks to Joe Johnston for adding another connection between open source development and XP. For others, see Deviance Testing

[3] According to Webster's New World Dictionary: "An onlooker at a card game, etc., esp. one who volunteers advice."

nagler@extremeperl.org

happen at first, but when it does, you both know the code resulted from the synergy of your experiences and skills.

6.3 Ease on down the Road

Your initial sessions should be short (two hours or less) with frequent breaks. Adopt pairing gradually until all production code is written in pairs. In some teams this is impossible, and that's OK. XP values people over process. Some people can't pair program full-time (see sidebar).

Unlike marriage, you don't try to pair with someone until death do you part. And, even in marriage, it's a rare couple that can sustain eight hours working closely together, day in and day out. In XP, you switch partners once or twice a day. It's not like musical chairs, where everybody moves to the next chair at the same time. You might seek out specific partner, because he is knowledgeable about a particular problem or technique. Or, somebody might ask you or your partner for help.

Don't change partners too often, however. It takes time for a new partner to develop the context necessary to be productive. As with all the XP practices, you'll find what works well for your team through trial and error.

6.4 Rest & Relaxation

Pair programming is a lot of work. Sharing is hard, as anybody who has observed a preschool classroom will note. Working side by side taxes even the most peaceful programmers. You will need to remember to take breaks before tempers flare.

Pair programming is more intense than solitary programming, because somebody is always driving. It's like running in a relay race where you hand off the baton and then hop on a bike to keep up with your partner. One reason XP recommends you work only 40 hours a week is that pair programming is hard. To be an effective partner, you need to be fresh and rested. The 40 hour work week reminds us that most of XP's practices were devised to address the human side of software development.

6.5 People Problems

Software is written by people. We're the source of all the ideas and errors that go into the code and that makes us an important, if not the most important, implementation risk factor of software projects. XP doesn't try

nagler@extremeperl.org

to hide the people issues behind a veneer of methodology and technology to protect us from each other. We don't ignore risk factors; we face them with courage.

Most of the XP practices are designed to force interpersonal issues to the surface so the team can address them. Software methodologists have long known that human factors are the weakest links along the chain from project conception to deployment. Gerry Weinberg sums this up in his second law consulting: "No matter how it looks at first, it's always a people problem."[4] One of the goals of XP is to help teams deal with interpersonal issues by themselves before management hires a consultant to figure out it for them.

I'm not trying to paint a picture that programming is a constant struggle with interpersonal issues, rather the complexity of our everyday lives and past experiences necessarily affects how we work. For example, have you ever had a bad hair day? Your hair, your breakfast, and the code comes out all wrong, and you just want to go home. Or, have you ever had a fight (excuse me, technical debate) with a co-worker that couldn't be resolved? You then each go off to solve it your own way. Instead of burying the hatchet, you end up burying the interpersonal conflict in the project's codebase.

XP's safeguards help resolve interpersonal issues like these. If you aren't your usual self one day, your partner will notice. He'll drive until you snap out of it. Everybody needs to coast sometimes. When you and your partner have a conflict, you have to work it out. If you don't, someone else on your team will notice you wrestling the keyboard from each other. They'll help you resolve the conflict, because they don't want to deal with battles embedded in the code.[5]

Pair programming teaches us about sharing, good manners, and acknowledging others' feelings and opinions. It's your average preschool curriculum.[6] And, by the way, pair programming can be as much fun as preschool, too. I find myself laughing and celebrating successes much more often than when I'm programming by myself. Sharing is about getting more out of who we are as individuals.

---

[4] *The Secrets of Consulting*, Gerald Weinberg, Dorset House, 1985, p. 5.

[5] Or, to have to dislodge the keyboard embedded from your partner's skull. It's bad for your partner and the keyboard.

[6] My son's preschool teacher was a corporate trainer. She said both jobs require the same skill set.

nagler@extremeperl.org

6.6 Different Strokes

The first step to acknowledging emotions is accepting that people are different. It's not as easy as it sounds. Over the years, we have developed bad habits in response to emotional situations. Some people withdraw in conflict situations. Others become more aggressive. Like most people, we reuse these patterns without regard to the other people in the situation.

Psychologists can help us to handle and to resolve conflict easily. Robert and Dorothy Bolton have developed a pragmatic approach to help people work with each other. Their system, called style flex, teaches us how to treat people differently based on our individual characteristics:[7]

> Style flex involves tailoring your behavior so that your work fits better with the other person's style. Flexing your behaviors is like a professional baseball player electing to swing differently at a fastball, a slider, and a curve. [...] Style flex is a way of adapting to another person's process; it is not about conforming to his or her point of view. It is about relating constructively while appropriately disclosing your perspective on things as well as listening empathetically to others. The better the interpersonal process, the more likely that people accurately hear each other and creatively resolve conflicting opinions.

And, one of their most important tips is:[8]

> When a relationship isn't going well, don't do more of the same; try something different.

If you keep this rule in mind, you will not only program with a partner better, you will also become more adept at dealing with your peers and your customers. Pair programming is an all-around learning experience that helps the team get the most out of each member, and everybody therefore gets the most out of the project. And, you'll also get help on bad hair days.

[7] *People Styles at Work*, Robert and Dorothy Bolton, American Management Association, 1996, p. 68.

[8] Ibid, p. 72.

6.7 Yea, Whatever

Not everybody in my company can sustain pair programming all the time. We try to flex to each other's styles when we can. Sometimes it just doesn't work.

One of my partners gets what I call the "yea whatevers" when he's had enough pair programming. He ends up saying, "yea, whatever" whenever there's some type of disagreement. The yea whatevers only happen after too many conflicts in one session. Our personalities would start hindering our efforts if we continued pair programming so we know it's time to stop.

XP promotes people over process first, and a set of best practices, second. If the people dynamics aren't right, I don't recommend forcing pair programming. It's taken this programmer and me years to figure out what works best for us. We didn't start out pair programming when we first started working together, and we don't pair continuously all the time now. Most of the time, we pair when we run into a roadblock, and we know we need help.

This also turns out to be a good way to introduce pairing, that is, waiting until you get a hard problem before you bring in a partner. It may be something you do already, possibly at a whiteboard. Next time, try sharing a keyboard instead. You will be surprised how much more effective it is to communicate through the code than in a conference room.

A whiteboard solution is theoretical, and code is being there. It's like the difference between planning to climb a mountain and hooting in unison when you both reach the top. Coding together means helping each other with the inevitable wrong turns and near falls. You gain mutual respect and learn to trust each other. You can't pair program effectively without these qualities, and the best way to get there is by working side by side on a hard problem.

6.8 Gumption Traps

Bad hair days are when nothing seems to go right. Robert Pirsig says this happens when you run into a gumption trap, something that saps your energy, discourages you, and causes you to make mistakes. Some examples

of gumption traps are:[9]

Intermittent Failure In this the thing that is wrong becomes right all of a sudden just as you start to fix it.

Parts Setback It's always a major gumption trap to get all the way home and discover that a new part won't work.

Value Rigidity The facts are there but you don't see them.

Ego Traps When the facts show that you've just goofed, you're not as likely to admit it. When false information makes you look good, you're likely to believe it.

Pirsig defines these gumption traps in terms of motorcycle maintenance, but they apply equally well to software development. Whether you are a mechanic or programmer, gumption traps keep you from getting your job done.

You can get out of gumption traps by yourself. Pirsig gives solutions for each of these traps. The biggest problem is recognizing that you have fallen into one. Even when you recognize you are in a trap, it can take you days to get out of one, if you are working alone. With a partner you'll probably be out in a few hours or possibly even minutes.

You and your partner think differently. You have different backgrounds and skills. When you are pair programming together, these differences balance each other out to help both of you avoid and recover from gumption traps.

Each of you probably won't fall into the same gumption traps. If you both fall into the same trap, one of you will come out first and help the other out. I'm sure you have had the experience of searching for a code defect with another person. It would be a rare event for both of you to find it at the exact same moment. It doesn't matter who finds it first, because both of you can continue coding once it is found.

6.9 Reducing Risk Through Knowledge Transfer

Programming pairs are not only better at finding defects, they produce better code and learn more than solitary programmers. Programming partners often come up with two different implementations for the same task. One

[9] *Zen and the Art of Motorcycle Maintenance*, Robert M. Pirsig, Bantam Books, 1989, p. 277-283.

or both of the implementations might contain ideas new to the other person. You'll discuss the differences, and possibly merge the solutions. In any event, one or both of you have learned something.

One partner is likely to know more about the code or problem you both are working on than the other. While you add business value to the project, the more experienced partner transfers knowledge to the less experienced partner. When all production code is written in pairs, no one person has unique knowledge of the codebase. This reduces risk, and spreads experience. Everybody benefits from the continous learning that goes on in an XP project. In addition, the customer benefits from the reduced risk and from the increased quality that pair programming delivers.

nagler@extremeperl.org

Chapter 7

Tracking

> He achieves successes
> he accomplishes his tasks
> and the hundred clans all say: We are just being natural.
>
> – Lao-Tze[1]

XP's focus is on the here and now. The team knows how it is doing by eliciting feedback through testing and frequent releases. Sometimes this feedback is not enough to keep a project on track. Everybody strays from the path now and then, and runs the risk of getting caught in a web made by a big, hairy spider. That's why every XP project needs a tracker.

Tracking in XP let's us know if we've strayed off the path. Tracking introduces feedback about how we are doing with respect to iteration and release plans. The tracker is the team member who is responsible for asking the team how far along they are with their tasks for the iteration. If the iteration is behind schedule, the tracker asks the customer to drop tasks. Intra-release feedback is generated by tracking the velocity trend. When the velocity curve is unstable, it's a sign of trouble.

The role of tracker is typicially filled by a manager, but can be anybody who communicates well with the rest of the team. The tracker's job is to help find root causes as well as gather data about the project's status. To be effective, the tracker must be a perceptive listener to hear the subtle clues that lead to root causes.

[1] *The Tao of The Tao Te Ching*, Lao-Tze, translated by Michael LaFargue, SUNY Press, 1992, p. 118.

This chapter explains how tracking works, what's tracked, ways to keep everybody informed, and most importantly, how to get back on track if you're lost battling big, hairy spiders.

7.1 Iteration Tracking

Iteration tracking is a polling activity. The tracker asks team members how their work is going once or twice a week. These meetings are between the tracker and one team member at a time. This keeps everyone at ease, and allows the tracker to ask personal questions if need be.

To simplify tracking, try to limit task length to one ideal day. With multi-day tasks, the tracker needs to ask how much time is left to get a precise reading of the project's status. With single day tasks, the tracker need only ask whether the task is completed or not. This latter precision is enough to provide an accurate picture of the iteration's progress.

Iteration tracking is not micro-management. The tracker doesn't try to control how people do things. The purpose of tracking is to find out what is happening using a simple, objective measure, such as, what tasks are done and what are their estimates. If the tracker has to ask how close a task is to being done, the measure is less objective, and the tracker may have to be more intrusive into how the team member came up with the estimate. The simple binary test ("is it done?") avoids this extra level of detail, and keeps the meetings shorter and more straightforward.

After the tracking meetings, the tracker sums the estimates of completed tasks, divides the sum by the current velocity, and multiplies by 100. The result is the percentage of planned work which has been completed for this iteration. If the number is 60% and you are at the end of day three of a one week iteration, the iteration is on track. If the number is lower, the tracker needs to ask the customer to cut scope.

7.2 Don't Slip the Date

Traditional project management tends towards adjusting the schedule to meet reality, that is, scope is fixed, and it's the date that needs to slip. XP's view is that scope is variable, and it's more important to keep dates fixed.

If you change the date, the entire team has to be involved. The team members who are on track will need new tasks (more communication). Their flow will be disrupted by an extra meeting to figure out what tasks need to be added.

nagler@extremeperl.org

The cheaper alternative to slipping the date is to cut tasks. This is less costly in terms of communication. The tracker already knows which tasks are in trouble, and he is already are talking with the people who are responsible for them. The tracker's job is to make the customer aware of the problems. The programmers who are on track needn't be bothered.

The customer picks which stories to drop. The tracker gives the customer the raw data (list of problematic stories) as well as your own interpretation of the factors involved. Call in the programmers to help you explain the details if need be. The better informed the customer is, the more easily she can decide what to drop.

7.3 Adding Tasks

Believe it or not, there are times when stories need to be added to an iteration. For example, the customer may see an important problem while using the prior iteration or a programmer may have finished all his tasks before the end of the iteration.

The customer decides what to add. The tracker only involves those programmers who are available for the additional stories. Not that the cusomter must stay within budget, so tasks may need to be dropped to balance out the additions. The tracker gives the customer a list of tasks in progress to help prevent unnecessary task switching.

7.4 The Tracker

The tracker must be someone who can speak directly to all team members. He needs to be a perceptive listener, because tracking is more than just numbers. It's about finding root causes and resolving them. Most problems you run into are related to the people, not the technology. And, people are needed to solve them.

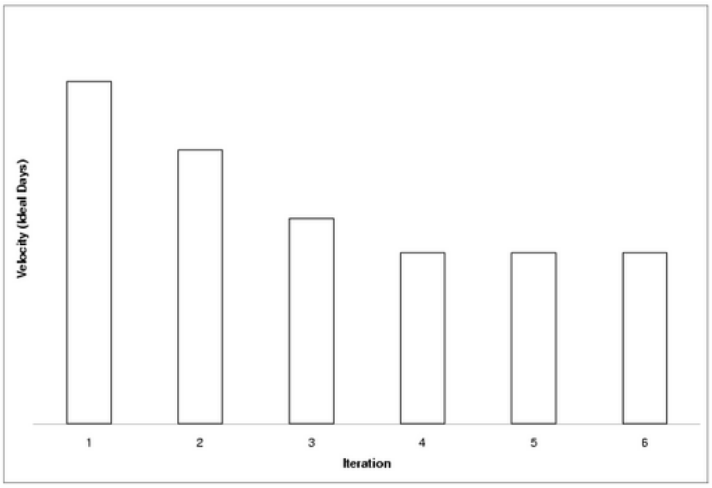
The role of tracker is best played by the project manager, in my opinion. Alternatively, you may want to rotate the position of Tracker throughout your project, or rotate the trackers between projects. Tracking, like management in general, needs to be understood by the whole team.[2] Rotating everybody through the position of tracker, gives the entire team the opporunity to see the forest for the trees.

[2] If you want to know why management is important, pick up a copy of *What Management Is: How it works and why it's everyone's business*, Joan Magretta, Simon and Schuster, Inc., 2002. It's an excellent, slim book on management.

7.5 Release Tracking

The simplest way to track release progress is with a graph of velocity. The graph should quickly converge on a constant value, for example:

Typical Velocity Graph



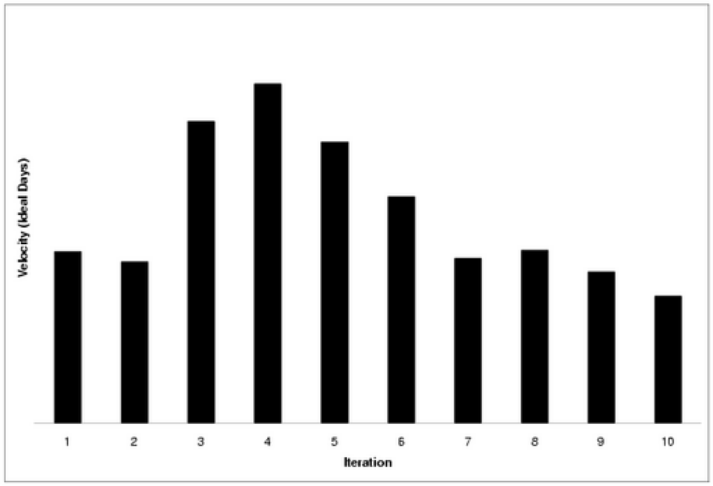
Velocity converges when the task estimates align with actual capacity. Once this happens, velocity should remain relatively constant.

Now let's take a look at a graph from a troubled project:[3]

Troubled Project's Velocity Graph

[3] This data was taken from 10 week-long iterations from a project at my company.

nagler@extremeperl.org

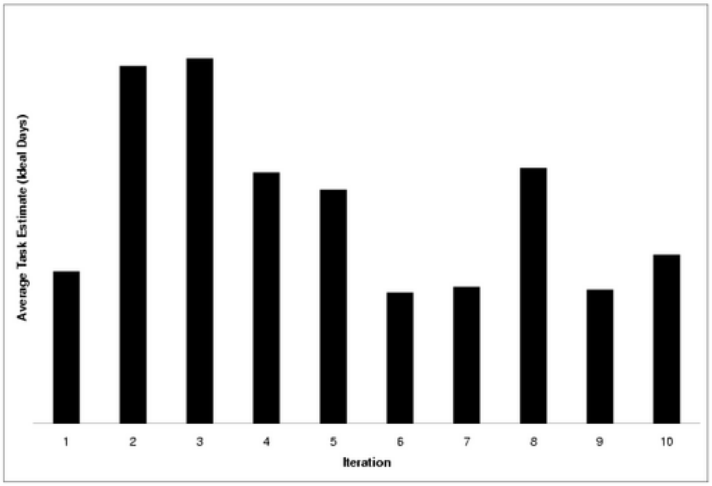


The project started out normally with velocity starting to converge around the third and fourth iteration. By the fifth or sixth iteration velocity started to drop off. The project was in serious trouble by the ninth iteration. The next step is figuring out why.

7.6 What Goes Wrong?

Finding the root cause for a velocity trend change can be difficult. In this project, we had a problem with quality which was indicated by an increase in the number of stories to fix defects. This was obvious when reviewing the story cards. When I went back to review the cards for this chapter, it struck me that the task estimates seem shorter as the project progressed. While I don't recommend tracking average task length in general, the graph illuminates the problem in this case:

Troubled Project's Average Task Length Graph



The graph shows that the average task length started dropping around the fourth iteration. In conjunction with the drop in velocity, this indicates the tasks were shorter, and we were underestimating them. These symptoms are indicative that change was harder, and were probably caused by quality issues. The team was battling the big, hairy, low quality spider.

Low quality is not a root cause, rather it's a symptom of not using best practices. After reviewing the project, we attributed the poor quality to the following root causes:

Customer Unavailable The customer didn't start using the system until the fifth iteration.

Too Few Acceptance Tests The team didn't test end to end function. It was thought there wasn't enough time, and was partially related to the team's relative inexperience.

Inexperience The domain, development environment, Perl, and XP were all new to the developers. The customer was new to XP. Doing too many new things at once is rarely a good idea, and in this case, was a significant weak point.

In the next section, you'll see how we fixed these problems. For now, let's take a look at a few other problems you might encounter when developing with XP:

nagler@extremeperl.org

Failing to Finish The customer does not stop making changes, and won't launch until all changes are done. This is a planning failure. With XP, you need to pick dates, and stick to them. The tracker should noticed that dates are slipping. In XP, we don't slip dates, we cut scope instead.

Too Little Refactoring Refactoring is put off until after launch. This reduces quality, which can be difficult to recover from, because the team falls into the bug-fixing trap soon after launch. Too little refactoring multiplies defects through copy-and-paste-itis: an inflammation of defects caused by over use of the copy-and-paste function in editors. The tracker should notice that the code base is growing too rapidly. If the number of lines of code is growing linearly with the number of ideal days, it means there's not enough refactoring. It's unlikely that every programmer thinks up the right code abstraction for every task. The programmers have to refactor to create the right abstractions, and that will shrink the code base.

Too Much Refactoring The programmers are perfectionists, and they won't stop trying to improve the code. This greatly reduces the amount of business value being added, and pushes off the launch date. This one may be a bit tricky to notice with a simple velocity graph, because the velocity may be constant. This is where the tracker needs to use his sixth sense that velocity is simply too low.

Not Enough Unit Tests Finding the right balance on unit tests is not easy, but it's fairly clear when there aren't enough: the team is afraid to make changes. This happened on one of my first XP-like (or should I say, XP-lite) projects. The project was successful enough, but now we are living under the constant fear of change. We're adding unit tests slowly, but it takes time, and it's hard to justify writing tests when "everything is working fine".

Too Little Pair Programming The programmers go off into their corners to get work done. The problem is that they don't know what everybody else is doing, and there's probably a lot of duplicated effort. Solitary programming is visually obvious to the tracker. It may even be mandated by management to save money. Velocity may be constant. However, quality will soon suffer.

nagler@extremeperl.org

7.7 Fixing Troubled Projects

All of these problems can be resolved. Sometimes the solutions take courage, such as, writing unit tests for a large testless codebase or telling the customer to launch now instead of waiting for one more change. The hardest part, I find, is figuring out what is wrong in the first place. The solution in most cases is simple: stop the counter-productive behavior.

Let's return to our troubled project. The customer realized there was a problem. We (the management) realized what was wrong, and the solutions were obvious. We cut scope so we could focus on improving quality. We substituted a senior developer into the project. We added some acceptance tests. And, probably most importantly, we made sure the customer was using the system. After four week-long iterations, we launched the system. We encountered a few structural issues after launch, but the project was on track, and the customer was much happier.

Probably one of the hardest problems to fix is when there has been too little refactoring. Extreme Perl has a size advantage in this case. Part of the reason we were able to fix our troubled project so quickly is that there was very little code to fix. Just a few thousand lines is all we needed to solve a fairly complicated problem in Perl.

If you are facing a mountain of code written in a relatively short time frame, it's likely to be bad code. Unfortunately, a mountain of code is often associated with a dearth of tests. You may end up having to throw all the code away, but the first step is to write tests to be sure you have documented the expected behavior. Once you have tests, start whittling the mountain away. The tests will make sure you don't break anything, and whittling will turn any code mountain into a codehill–pardon my metaphoritis.

nagler@extremeperl.org

7.8 Meetings

XP puts people over processes. We don't like wasting time in group meetings. Formal meetings are limited to planning and what we call, stand-up meetings.

A stand-up meeting occurs daily, usually in the morning. The meetings are held standing up, hence the name. Standing helps limit the meeting's duration. The content of a stand-up is announcements and a brief status report from all team members.

As an alternative to stand-ups, you might consider status meetings held over lunch once or twice a week. Lunch is a more relaxed way than a stand-up to make announcements and discuss what you are working on. The status lunch also gives the team a chance to socialize.

The company should pay for the meal. At one company we hired an Italian cook to prepare family-style meals three times a week. Nobody was late to lunch when Dina was cooking!

7.9 Show Your Stuff

Up till now, being the tracker may not seem like a lot of fun. However, part of the tracker's job is encouraging people to celebrate their successes as well as helping out with failures. Everybody wants to be praised for their work. The tracker should encourage people to show off their completed work to the entire team. Success is contagious. When you see the excitement from a sweet demo, you'll be motivated to work that much harder so you can demo your cool stuff.

Demonstrations are great way to maintain team spirit and momentum. Even a short e-mail explaining your result is enough. For example, you introduce a cache and improve the performance by 10 times. Send out a graph containing the performance test comparison.

The tracker should organize demonstrations if they don't come about naturally. Sometimes team members are reticent to show their work. Weekly demonstrations are a good way to ensure everybody is praised for their accomplishments. Before or after lunch are good times for demos (see sidebar).

7.10 Sign Off

The main indicator for how much business value has been implemented is the percentage of acceptance tests that are passing. Acceptance testing is the way we validate stories have been implemented. While this is a great way to measure where you are, it's also nice to audit the cards themselves. I like my customers to formally sign off acceptance on the story cards in an acceptance meeting. The formality helps ensure the customer, the tracker, and the cards are in sync. In large projects, there's bound to be a problem with or a feature missing from one of the stories. The customer and the tracker should read the story cards again to make sure they really are complete.

7.11 Here and Now

Rather than trying to predict the future, XP focuses on the present with an eye on the past. We know where we are and from whence we have come. The unit tests tell the programmer if his changes are valid. The acceptance tests validate the system as a whole. The tracker knows which tasks are done and which are not. These feedback mechanisms keep everybody informed about the project's state.

It's normal to have unfinished tasks at the end of an iteration. Don't worry. You'll complete them next iteration. No amount of charting, analysis, and/or forecasting will magically complete the tasks for you. If there is a pervasive planning problem, it's almost always related to the people on the project, not some technological issue. XP's practices are designed to prevent people issues from remaining hidden. The tracker's main job is to remind people what those best practices are.

Chapter 8

Acceptance Testing

> For defect removal, the most prominent attribute of a requirements specification is its testability.[1]
>
> – Robert Dunn

Acceptance tests are the functional specifications of XP. Each story card is elaborated in one or more scripts by the customer. The test suite is run regularly on the entire application, and the programmers can use the scripts to diagnose implementation defects.

The purpose of this chapter is to demonstrate how the customer can create automated acceptance tests. Acceptance tests are explained in general and with example test scripts. An alternative approach called, data-driven testing, is presented that allows the customer to generate tests using ordinary office software, such as, spreadsheets and word processors.

8.1 Acceptance Tests

Acceptance tests, also known as, customer tests and functional tests, validate the customer's view of the application. They are distinct from unit tests, which validate the programmer's view of the software internals. The programmers help write acceptance tests, but as with writing story cards, the customer needs to be involved directly. Otherwise, customer expectations probably won't be met by the application.

When writing tests, the customer fills in the details the story cards leave out. Each test encodes one or more specific user scenarios. When the test

[1] *Software Defect Removal*, Robert Dunn, McGraw-Hill, Inc., 1984, p. 28.

runs, it performs the same actions a user would, and, ideally, validates what the user would see. An acceptance test simulates the user automatically.

8.2 Automation

Acceptance tests need to be automated to be effective XP tools. You can write manual tests, that is, tests where a person acts as the simulated user. However, manual tests don't fit well with XP, because they don't provide the consistent, instantaneous feedback that XPers crave. Manual tests have their place, and the existing literature covers them well.[2] This book is about empowering teams to use XP, and manual tests are not tools in a typical XP project.[3]

The trade-off between a manual and automated test is cost. Automation has a higher up front cost, but in XP the cost is amortized quickly. Acceptance tests are run nightly or more frequently in an XP project. When dealing with physical devices, such as printers, automation is expensive. For typical Perl applications, however, inertia is a greater barrier to automation than cost.

I take a top-down approach to overcome automation inertia. Let the customer write or specify the test, and then figure out how to implement it. Tests can be implemented partially yet still be valuable. As always in XP, take it one step at a time, and do the simplest thing that could possibly work.

8.3 I'm Gonna Buy Me a Dog[4]

The examples that follow test PetShop, a demonstration, open source, online pet store.[5] The specifics of the scripts could apply to most online stores so you don't need to try PetShop to read the examples. Potential buyers add items to a shopping cart while they browse or search a hierarchical catalog. To complete the check out, they must login or register.

[2] *Testing Computer Software* by Cem Kaner et al is an excellent book about classical testing and quality assurance.

[3] In *Testing Extreme Programming*, Lisa Crispin and Tip House state that "All acceptance tests on an Extreme Programming project must be automated" and devote an entire chapter to explain why.

[4] I want to go on record that I do not promote buying animals in stores. Buy animals from a reputable breeder, or better yet, adopt a pet from your local animal shelter.

[5] Visit http://petshop.bivio.biz to see it live and to browse the source. The idea was inspired by Sun Microsystems, Inc.'s J2EE Blueprint Pet Store.

This first script tests several ways to find a Corgi, a small herding dog. The stories tested by the script are:

- Organize catalog hierarchically by category, breed, and animal for sale.
- Allow buyers to search for animals by keyword(s).

The test script in Perl syntax is:

```
test_setup(&#39;PetShop&#39;);

home_page();
follow_link(&#39;Dogs&#39;);
follow_link(&#39;Corgi&#39;);
add_to_cart(&#39;Female Puppy Corgi&#39;);

search_for(&#39;corgi&#39;);
add_to_cart(&#39;Female Puppy Corgi&#39;);
search_for(&#39;CORGI&#39;);
add_to_cart(&#39;Female Puppy Corgi&#39;);
search_for(&#39;dogs wales&#39;);
add_to_cart(&#39;Female Puppy Corgi&#39;);
```

The first line in the script tells the test framework[6] what we are testing. `test_setup` establishes the functions, or actions, used by the test. Since this is an online store, the actions include: `home_page`, `follow_link`, `search_for`, and `add_to_cart`.

Acceptance test scripts are similar to the script to a movie or play. The roles are the user and the computer. The script shows the user role, or what the user does and expects to see. For example, the second section of the script follows the site's hierarchy to put a dog in the cart. It goes to the home page, selects the `Dogs` animal category, drills down to the `Corgi` breed, and, finally, puts a `Female Puppy Corgi` into the cart. These are the actions of an ordinary user.

The test script is run through an interpreter, a program that translates the functions into interactions with the application. The programmers

[6] Programmers: This chapter is devoted to explaining to the customer's view of acceptance testing, not the programmer's. The test script interpreter consists of a few lines of Perl, and uses `eval` and `AUTOLOAD`. You could write this yourself, or download my company's open source implementation from http://www.bivio.biz.

nagler@extremeperl.org

implement the functions on demand for the customer. The functions are responsible for checking the computer's role in the script. For example, this test script states that the user clicks on the `Dogs` link on the home page. If the `Dogs` link is missing from the home page or spelled incorrectly, the `follow_link` action stops and indicates that the test script has failed. This approach is called `fast fail`, and makes it easy to write and maintain test scripts. In keeping with the movie script analogy, it's like when the director sees computer screwing up its lines, and yells, "cut!". Everybody stops. The director corrects what's wrong, and the actors start over again.

The next section tests the search facility. We should be able to find our dog by searching for `corgi`, `CORGI`, and `dogs wales`. We aren't particularly interested in Corgis[7], rather our goal is to test that the search mechanism is case-insensitive and supports multiple words. And, most importantly, the list of search results allows the buyer to place found animals in their cart easily. Shoppers should be given an opportunity to buy what they find.

8.4 Group Multiple Paths

8.4. GROUP MULTIPLE PATHS

The previous example demonstrated testing multiple paths, that is, different ways of doing the same thing. In one case, we searched for a `Female Puppy Corgi` hierarchically, and then we used the search box to find the same dog using different keywords. Here is another example that demonstrates multiple paths:

```
test_setup(&#39;PetShop&#39;);
home_page();
follow_link(&#39;Reptiles&#39;);
follow_link(&#39;Rattlesnake&#39;);
add_to_cart(&#39;Rattleless Rattlesnake&#39;);
remove_from_cart(&#39;Rattleless Rattlesnake&#39;);

search_for(&#39;Angelfish&#39;);
add_to_cart(&#39;Large Angelfish&#39;);
update_cart(&#39;Large Angelfish&#39;, 0);
```

[7] Or other high-energy small dogs.

This example tests the two ways you can remove animals from the cart. `remove_from_cart` uses a button labeled `Remove` to delete the item from the cart. `update_cart` allows buyers to change the quantity desired. Setting it to zero should have the same effect as `remove_from_cart`.

Most applications allow you to do something in more than one way, like in this example. Grouping similar functions in the same test is another organizational technique for your acceptance test suite. It also provides an opportunity to talk about an application cross-functionally. The creation of test scripts is a collaborative effort, much like pair programming. This sort of detailed matters, and probably won't come up during the planning game. The details emerge when the stories and their acceptance tests are being implemented. The test suite opens a communication channel between the programmers and the customer to discuss application consistency and other technical details, such as what to do when the user enters an unexpected value.

8.5 Without Deviation, Testing Is Incomplete

The acceptance test suite also checks that the application handles unexpected input gracefully. For example, if the user enters an incorrect login name, the application should tell the user `not found` or something similar. The technical term for this is deviance testing. It's like kicking the tires or slamming the car into reverse while driving on the highway. The previous examples are conformance tests, because they only validate using the application for its intended purpose. When you write a deviance test, you break the rules in order to ensure the application doesn't do the wrong thing, such as displaying a stack trace instead of an error message or allowing unauthorized access.

For example, here's how we test login conformance and deviance of the PetShop:

```
test_setup(&#39;PetShop&#39;);
home_page();
login_as(&#39;demo&#39;, &#39;password&#39;);
login_as(&#39;DEMO&#39;, &#39;password&#39;);
login_as(&#39;demo@bivio.biz&#39;, &#39;password&#39;);
login_as(&#39;Demo@Bivio.Biz&#39;, &#39;password&#39;);
```

```
test_deviance(&#39;does not match&#39;);
login_as(&#39;demo&#39;, &#39;PASSWORD&#39;);

test_deviance(&#39;must supply a value&#39;);
login_as(&#39;demo&#39;, &#39;&#39;);
login_as(&#39;&#39;, &#39;password&#39;);

test_deviance(&#39;not found&#39;);
login_as(&#39;notuser&#39;, &#39;password&#39;);
login_as("demo&#39;||&#39;", &#39;password&#39;);
login_as(&#39;%demo%&#39;, &#39;password&#39;);
```

The first section tests conformance. We login as `demo` and `DEMO` to test that user names can be case insensitive. The PetShop allows you to login with an email address, case insensitively.

Passwords are case sensitive, however. The next section expects the application to return an error message that contains `does not match` when given a password in the wrong case. This is a deviance test, and the `test_deviance` that begins the next section tells the test framework that the subsequent statements should fail and what the expected output should contain. This is an example where the test script specifies the computer's role as well as the user's.

The application should ask the user to supply a value, if either the login name or password fields on the form are blank. The next section tests this. This case might be something a programmer would suggest to the customer. The customer might decide that `must supply a value` is too computer-like, and ask the programmer to change the application to say something like, `Please enter your login ID or email address.`

In the last section, we test a variety of `not found` cases. The first case assumes that `notuser` is not a user in the system. The test suite database is constructed so that this is the case. The last two cases are highly technical, and are based on the programmer's knowledge of the application internals, that is, SQL, a database programming language, is used to find the user. Some applications do not correctly validate application input, which can allows the user unauthorized access to system internals. This is how computer virii and worms work. This test case validates that the user name is checked by the application before it is used in a low-level SQL statement. If the user name syntax is not checked by the application, one of the last two cases might allow the user to login, and the deviance test would fail.

nagler@extremeperl.org

Note that we didn't test `notuser` without a password. It's not likely that an invalid user could login without a password when a valid user couldn't. In testing parlance, the two tests are in the same equivalence class. This means we only need to test one case or the other but not both.

We use equivalence classes to reduce the size of the test suite. A large application test suite will have thousands of cases and take hours to run. It's important to keep the runtime as short as possible to allow for frequent testing. And, as always, the less code to do what needs to get done, the better.

8.6 Subject Matter Oriented Programming

8.6. SUBJECT MATTER ORIENTED PROGRAMMING

Another way to minimize test length is letting the problem, also known as subject matter, guide the development of the functions used by the scripts. The customer is probably not a programmer. Moreover, the customer's terminology has probably been refined to match her subject matter. The programmers should let the customer choose the function names, and the order and type of the function parameters. The language she uses is probably near optimal for the subject and workflow.

The process of bringing the program to the problem is what I call, subject matter oriented programming (SMOP). It is what XP strives for: creating an application that speaks the customer's language. The acceptance test suite is probably the customer's most important design artifact, because it encodes the detailed knowledge of what the application is supposed to do. If she or her co-workers can't read the tests, the suite's value is greatly diminished.

The design and implementation of the acceptance test suite evolves as the customer encodes her knowledge. The programmer may need to help the customer to identify the vocabulary of the subject matter. Subject matter experts sometimes have difficulty expressing what they do succinctly. The programmer needs to be part linguist, just like Larry Wall, Perl's inventor. Unlike other language designers, Larry lets the problems programmers face dictate the solution (the programming language) they use. Perl is not prescriptive, in linguistics terms, but descriptive, evolving to meet the language used by programmers, not the other way around.

Enough theory. I'm in danger of getting lost in the solution myself. If you are a programmer, you'll learn how to implement a subject matter oriented program in the It's a SMOP chapter. I'll get back to the customer,

nagler@extremeperl.org

and another method by which she can create the acceptance test suite.

8.7 Data-Driven Testing

8.7. DATA-DRIVEN TESTING

The test examples up to this point have been written in Perl syntax. While I fully believe just about anybody can follow these simple syntactic conventions, customers may balk at the idea. Ward Cunningham, a well-known XPer, has taken subject matter oriented programming to a new level. His framework for intergrated testing (FIT) lets customers write acceptance tests in their own language using their own tools, office applications, such as, word processors and spreadsheets. Here's the login test translated as a FIT document:

FIT Login

This document is a FIT input file that tests login conformance and deviance.

PetShop			
login	demo	password	
login	DEMO	password	
login	demo@bivio.biz	password	
login	Demo@Bivio.Biz	password	
login	demo	PASSWORD	does not match
login	demo		must supply a value
login		password	must supply a value
login	notuser	password	not found
login	demo'll'	password	not found
login	%demo%	password	not found

FIT ignores all text in the document except for tabular text. The tables contain the text inputs and expected outputs. This allows the customer to document the test, and to have one document which contains many tests. The order of the columns and what they are for is worked out between the

customer and the programmer. Once that's done, the framework does the rest.[8]

Just like the Perl examples earlier, the customer must specify the test language interpreter, `PetShop`. In this type of FIT test, the customer enters actions (`login`) on a row-by-row basis. The programmer can create new actions. The cells to the right of the action name are parameters. The `login` action accepts a user name, a password, and an error message. If the there's no error message, `login` tests that the login was successful.

The subject matter may suggest a different organization for the tables. For example, here's a denser test format for a simple math module:[9]

FIT Math

This FIT document tests a simple math application using columnar tests.

SimpleMath			
x	y	sum	diff
1	2	3	-1
-8	12	4	-20
0	33	33	-33
101	0	101	101

As with the login test, the first line contains the test language interpreter, `SimpleMath`. The next row lists the actions in a columnar format. The first action sets an `x` value, the next sets `y`, and the last two columns test adding

[8] Thanks to Brian Ingerson for implementing `Test-FIT`, and making it available on CPAN.

[9] `SimpleMath` and the test data were adapted from `Test-FIT`, version 0.11, on CPAN.

(`sum`)) and subtracting (`diff`). The subsequent rows contain a test in each cell of the table. The first row sets `x` and `y` to `1` and `2` and tests that `sum` and `diff` return `3` and `-1`. As you can see, this kind of FIT test gives the customer a clear overview of the acceptance test data using an ordinary word processor. With this style of testing, customers can create spreadsheets using formulas.

The general term for using documents as test inputs is called data-driven testing. And, sometimes there's no practical alternative to using tabular data. On one project we developed, we needed to test the correctness of a pre-marital evaluation tool. Each partner in a couple had to answer 350 questions. The scoring algorithm related the couple's answers for compatibility. The customer had supplied us with the questions, answers, scores, and weights in tabular format. When we asked for acceptance test data, he simply added the answers for test couples in another column, and we generated the test suite by parsing out the data. As it turned out, the test data uncovered several areas that were misunderstood by the programmers. Without customer generated test data, the software would have contained critical defects.

8.8 Empower The Customer to Test

8.8. EMPOWER THE CUSTOMER TO TEST

Whether the customer uses a spreadsheet, a word processor, or Perl, she can write tests. And, she needs to. No one else on the team knows the subject matter better than she does.

Getting started is the hardest part. Take the simplest and most straightforward part of the application. Write a test outline for it together on the whiteboard. Implement that test, and run it together.

After the first steps, you'll fill in more and more detail. As the suite grows with the implementation, the application will benefit from the regular exercise. The programmers will gain deeper insight into the subject matter. The customer will see the quality improve firsthand. And, everybody will benefit from the well-structured knowledge base encoded by your acceptance test suite.

Chapter 9

Coding Style

> Language requires consensus.
>
> – Larry Wall[1]

Code is the primary means of communication in an XP team. A uniform coding style greatly facilitates code comprehension, refactoring, pair programming, collective ownership and testing. An XP team agrees on a coding style before development starts.

Coding style is the first problem and XP team has to work out as a group. The solution to the problem needs to be clear and unambiguous. It's amazing how far this can be, and with some teams it's impossible.

Coding style discussions are like a lightning rod. If there's a storm brewing within the team, coding style will usually attract the first lightning strike. If you can't reach agreement on a style, your team is going to have difficulty building an application together.

Tension around style choices is natural in some ways. We are all individuals. Programmers take pride in their own work, further motivating their own success, just as individual athletes value their own accomplishments. However, not even the best pitcher, quarterback, or forward in the world can win a game alone. It takes a team and teamwork to win a game or write a large application. Programming is a team sport.[2]

If you are a programmer, you may find yourself gritting your teeth at my coding style. It wouldn't surprise me. Athletes and programmers on

[1] *Open Sources: Voices from the Open Source Revolution*, DiBona et al, 1999, O'Reilly, p. 127. Available online at http://www.oreilly.com/catalog/opensources/book/larry.html

[2] And more generally, "Business is a team sport." *Rich Kid, Smart Kid*, Kiyosaki et al, Warner Books, 2001, p. 224-225.

different teams sometimes bristle at each other's style. However, if we were to join the same team, we'd work out a compromise on coding style to ensure the success of the project.

This chapter explains the need for a coding style in XP and discusses how to go about creating one. I also explain and demonstrate the coding style used in this book through a comparative example.

9.1 There's More Than One Way To Do It

Perl is a rich and complex language. If you ask a question about how to do something in Perl on the Internet, you'll probably get several different answers. And, the answers will often include the caveat: TMTOWTDI. This is the acronym for Perl's motto: There's more than one way to do it. The solution you choose will depend on the way you program Perl.

So how do you program Perl? Larry Wall et al present a coding style in the `perlstyle` man page.[3] Yet there are myriad divergent styles on CPAN and in the Perl literature. In the Perl community, diversity is seen as a strength, and no one is going to tell you how to program Perl. Well, even if they did, you wouldn't listen to them.

9.2 Give Me Consistency or Give Me Death

Your team still needs to pick a style. This isn't just XP dogma; it's human nature. In the anthropology classic, *The Silent Language*, Edward Hall wrote, "The drive toward congruity would seem to be as strong a human need as the will to physical survival." I conclude from this that if you don't pick a Perl coding style, you'll die. If that isn't a good enough reason, stop reading now.

Seriously, consistency is not an end in itself, it is the means to facilitate testing, collective ownership, pair programming, and refactoring. If you are developing a small application (a few thousand lines of Perl), it's easy to keep the code consistent, or to clean it up in an afternoon or two. For large applications (tens or hundreds of thousands of lines spread over hundreds or thousands of files), quick fixes are impossible. You would never have the time to reformat the entire codebase.

The code changes too quickly. You don't get to ask everybody working on a large application to stop while you fix some style issue. However, for

[3] http://www.perl.com/doc/manual/html/pod/perlstyle.html

some necessary refactorings, such as, a change in a widely used API, you may have no choice but to dive into tens or possibly hundreds of files. You'll want this to happen as quickly as possible so you'll automate the refactoring. With a consistent style, you can probably do this fairly easily. If you have to account for the many ways you can do things in Perl, you'll probably resort to hand editing each file. Not only is this labor intensive, but it's error prone, too.

Even when you aren't making global changes, you and your partner still have to read unfamiliar code. A programming pair's ability to read and to communicate through the code is affected directly by its consistency. Communication is hard enough without you and your partner having to wade through several code dialects. And, allowing for style variations when writing code opens up too many unnecessary thoughts. Do I adapt to my partner's style? Should we adopt the style of the code we're editing now? Or perhaps, I should insist on my style. After all, it has worked well for me over the years. Starting out with an agreed upon style, frees our minds of such distractions and allows us to focus on the important bit: solving the customer's problem.

9.3 Team Colors

In *Extreme Programming Explained*, Kent Beck wrote, "The standard must be adopted voluntarily by the whole team." This may not be so simple. Establishing consensus requires work on everybody's part. If your team has coded together before, you'll probably have an easy time agreeing on a style.

For newly formed teams, use the style guide as a team building exercise. Everyone should be encouraged to contribute. If a particular point is too contentious, drop it until after the first iteration or so. The goal is to get full consensus on the entire guide. If someone is particularly inflexible during the discussions, it's a warning sign that a team adjustment may be necessary. Better sooner than later.

A style guide can be highly motivating, however. It's like your team's colors. It's something relatively insignificant which provides significant cohesion. If even one team member is coerced into agreement, the team isn't sticking together, and the rift may grow into a chasm. When everybody voluntarily accepts the style choices, you are functioning as a team, and you are ready to code.

9.4 An Example

Rather than discuss style in the abstract, I'd like to demonstrate my opinion of good style through a comparative example. Here's an excerpt from the popular `Test` package on CPAN: [4]

```
package Test;
use strict;

sub plan {
  croak "Test::plan(%args): odd number of arguments" if @_ & 1;
  croak "Test::plan(): should not be called more than once" if $planned;

  local($\, $,);   # guard against -l and other things that screw with
          # print

  _reset_globals();

  _read_program( (caller)[1] );

  my $max=0;
  for (my $x=0; $x < @_; $x+=2) {
    my ($k,$v) = @_[$x,$x+1];
    if ($k =~ /^test(s)?$/) { $max = $v; }
    elsif ($k eq &#39;todo&#39; or
       $k eq &#39;failok&#39;) { for (@$v) { $todo{$_}=1; }; }
    elsif ($k eq &#39;onfail&#39;) {
      ref $v eq &#39;CODE&#39; or croak "Test::plan(onfail => $v): must be CODE"
      $ONFAIL = $v;
    }
    else { carp "Test::plan(): skipping unrecognized directive &#39;$k&#39;" }
  }
  my @todo = sort { $a <=> $b } keys %todo;
  if (@todo) {
    print $TESTOUT "1..$max todo ".join(&#39; &#39;, @todo).";\n";
  } else {
    print $TESTOUT "1..$max\n";
  }
```

[4] http://search.cpan.org/src/SBURKE/Test-1.22/lib/Test.pm

nagler@extremeperl.org

```
  ++$planned;
  print $TESTOUT "# Running under perl version $] for $^O",
   (chr(65) eq &#39;A&#39;) ? "\n" : " in a non-ASCII world\n";

  print $TESTOUT "# Win32::BuildNumber ", &Win32::BuildNumber(), "\n"
   if defined(&Win32::BuildNumber) and defined &Win32::BuildNumber();

  print $TESTOUT "# MacPerl verison $MacPerl::Version\n"
   if defined $MacPerl::Version;

  printf $TESTOUT
   "# Current time local: %s\n# Current time GMT:   %s\n",
   scalar(   gmtime($^T)), scalar(localtime($^T));

  print $TESTOUT "# Using Test.pm version $VERSION\n";

  # Retval never used:
  return undef;
}
```

The routine `plan` is used to set up a unit test, for example:

```
use Test;
use strict;
BEGIN {
  plan(tests => 2);
}
ok(1 + 1 == 2);
ok(2 * 2 == 4);
```

This unit test calls `plan` to declare that there are two test cases. The `ok` function checks the result after each case executes, and prints success or failure.

Here's what I think is good about this implementation of `plan`. The routine:

- is well-used and mature. We can be relatively sure it addresses the needs of its users and is relatively stable.

nagler@extremeperl.org

- has had several authors. The more eyes on a problem, the better the solution.

- addresses type safety. `Test` has a broad user base, so it makes sense to put extra effort into argument validation.

- fails fast, that is, if `plan` encounters an unexpected argument or state, it terminates execution (calls `croak`) in all but one case. The sooner a programming error is detected, the less damage the errant program can do.

- is backwards compatible. The parameters `test` and `failok` are deprecated, that is, they shouldn't be used in new tests, but existing tests that use them still work.

- comes with a thorough unit test suite that describes expected behavior and enables refactoring.

- uses fine-granularity, feature-driven portability. It uses narrow feature checks (for example, `defined $MacPerl::Version` and `chr(65) eq 'A'`) instead of broad general checks, such as, checking the operating system (for example, `$^O eq 'MacOS'`). Powerful and easy to use introspection is one of the reasons Perl and CPAN packages are usable on so many platforms.

9.5 You Say, "`if else`", And I Say, "`? :`"

While indentation, lining up braces, or other formatting is important to ease automated refactoring, you won't find much discussion about them in this book. However, the more strictly you follow your coding style, the more easily you can automate refactorings. For example, if function arguments are always surrounded by parentheses, you can rename functions or reorder parameters using a simple editor macro.[5]

And speaking of editors, most programmers' editors have style formatters. If yours doesn't, you can always use `perltidy`, a very flexible Perl code reformatter.[6] Automatic formatters improve your team's efficiency and adherence to your style guidelines. That's all I'm going to say about formatters

[5] You can download some refactoring functions for Emacs from http://www.bivio.biz/f/bOP/b-perl.el

[6] Available for free from http://perltidy.sourceforge.net

nagler@extremeperl.org

and editors. The only thing worse than coding style discussions are editor wars.

Like editors, style choice is defined by your experience and personal taste, and the details matter. I follow the `perlstyle` man page for the most part, but I disagree with some of their parentheses and alignment choices. You may not like my style, so go ahead and indent by three spaces if you like.[7]

9.6 Once And Only Once

Parentheses and indentation aside, the important bit of my style is that I refactor ruthlessly. I don't like redundancy, especially in the form of single use temporary variables and repetitive calls. In XP, we call this the once and only once (OAOO) rule, and it's what you do when you refactor.

For new code, I try to do the simplest thing that could possibly work (DTSTTCPW). This is XP's most important coding guideline. First I get it working simply. I might have to copy and paste some code or create a temporary variable. Once it passes the tests, I look at the design and simplify it so that each concept is expressed once and only once. There are some time and planning trade offs here, and the Refactoring chapter discusses them. The relevant point is that once and only once is an overarching style guidline, and one that I value highly. When concepts are expressed once and only once, the code is more robust, more easily extensible, and performs better than code with needless duplication.

9.7 Refactored Example

The code that follows has been changed to demonstrate once and only once and other style choices I value. The formatting matches the style used in this book. More importantly, the rewritten code is more cohesive. Not only should each concept be expressed only once, but each routine should implement only one concept. Strong cohesion allows people to comprehend and to abstract code entities (routines and packages) easily. By isolating and naming each behavior, we make it easy to understand what each piece of the software puzzle does.

The four new routines are also loosely coupled. This means their inputs and outputs are few and well-defined. Loose coupling is important when isolating behavior, because it is difficult to understand and to test a routine

[7] Even though it's a sin.

with many inputs and outputs. In effect, the routine's identity is a combination of its name and its inputs and outputs, which is commonly know as its signature. We remember shorter signatures and the behavior they identify more easily than longer ones.

That's enough theory for now, here's the my version of `plan`:

```
package Test;
use strict;

sub plan {
  my($args) = {@_};
  Carp::croak(&#39;should not be called more than once&#39;)
    if $_TODO;
  _reset_globals();
  _read_program((caller)[1]);
  _plan_print(_plan_args($args));
  return;
}

sub _plan_args {
  my($args) = @_;
  $_ONFAIL = _plan_arg_assert($args, [&#39;onfail&#39;], &#39;CODE&#39;);
  my($max) = _plan_arg_assert($args, [&#39;tests&#39;, &#39;test&#39;], &#39;inte
  # $_TODO is the initialization sentinel, so it&#39;s the last value set
  $_TODO = {map {$_ => 1}
    @{_plan_arg_assert($args, [&#39;todo&#39;, &#39;failok&#39;], &#39;ARRAY&#39
  Carp::carp("@{[sort(keys(%$args))]}: skipping unrecognized or",
    &#39; deprecated directive(s)&#39;)
    if %$args;
  return $max;
}

sub _plan_arg_assert {
  my($args, $names, $type) = @_;
  foreach my $n (@$names) {
    next unless exists($args->{$n});
    Carp::croak("$n: parameter must not be undef")
      unless defined($args->{$n});
    Carp::croak("$args->{$n}: $n must be $type")
      unless $type eq &#39;integer&#39; ? $args->{$n} =~ /^\d+$/
```


nagler@extremeperl.org

```
        : ref($args->{$n}) eq $type;
    return delete($args->{$n})
  }
  return undef;
}

sub _plan_print {
  my($max) = @_;
  _print(join("\n# ",
    "1..$max"
      . (%$_TODO ne &#39;&#39; && " todo @{[sort {$a <=> $b} keys(%$_TODO)]};"),
    "Running under perl version $] for $^O"
      . (chr(65) ne &#39;A&#39; && &#39; in a non-ASCII world&#39;),
    defined(&Win32::BuildNumber) && defined(Win32::BuildNumber())
      ? &#39;Win32::BuildNumber &#39; . Win32::BuildNumber() : (),
    defined($MacPerl::Version)
      ? "MacPerl version $MacPerl::Version" : (),
    &#39;Current time local: &#39; . localtime($^T),
    &#39;Current time GMT: &#39; . gmtime($^T),
    "Using Test.pm version $VERSION\n"));
  return;
}

sub _print {
  local($\, $,);
  return print($TESTOUT @_);
}
```

9.8 Change Log

The following is a detailed list of changes, and why I made them. Most of the changes are refactorings, that is, they do not modify the way `plan` works from the caller's perspective. A few changes improve the behavior ever so slightly, and are noted below. This list is ordered from most important to trivial:

- The four main behaviors: control flow, validating arguments, type

nagler@extremeperl.org

checking, and printing are contained in separate routines. Each routine is responsible for one and only one behavior.

- The `localtime` and `gmtime` calls are now in the correct order. This defect in the original version only became apparent to me when I separated the two output lines.

- Argument type validation is consistent, because it has been isolated into a single routine (`_plan_arg_assert`) that is used for all three parameters. Several new cases are caught. For example, passing `undef` to `tests` or passing both `tests` and `test` (deprecated form) is not allowed.

- Carp::croak unrecognized directive warning is printed once instead of a warning per unrecognized directive. The check for unrecognized directives still does not fail fast (`croak` or `die`). I would have liked to correct this, because passing an invalid directives to `plan` probably indicates a broken test. However, the broad user base of `Test` makes this change infeasible. Somebody may be depending on the behavior that this is only a warning.

- Two temporary variables (`@todo` and `$x`) were eliminated by using a functional programming style. By avoiding temporary variables, we simplify algorithms and eliminate ordering dependencies. See the It's a SMOP chapter for a longer example of functional programming.

- `$planned` was eliminated after `$_TODO` was converted to a reference. `$planned` is known as a denormalization, because it can be computed from another value (`$_TODO` in this case). Normal form is when data structures and databases store the sources of all information once and only once.

- `_plan_print` writes a single string. The seven calls `print` were unnecessary duplication. I often use logical operators instead of imperative statements to avoid the use of temporary variables, which are another form of duplication (denormalization).

- The return value from `plan` is better represented as an empty `return`, because it handles list and scalar return contexts correctly. This is a subtle point about `return`, and it actually involves an interface change. The following use assigns an empty list:

```
my(@result) = Test::plan(tests => 1);
```

In the old version, `@result` would contain the list `(undef)`, that is, a list with a single element containing the value `undef`.

- The check for an odd number of arguments is unnecessary, because the assignment to a hash will yield a warning and the argument parsing is more rigorous (no argument may be `undef`, for example).
- `_print` encapsulates the output function that is used throughout `Test`. The concept that the output is directed to `$TESTOUT` is only expressed once.
- The global variables are named consistently (`$_ONFAIL` and `$_TODO`). I name global variables in uppercase. I use a leading underscore to identify variables and routines which are to be used internally to the package only.

  `$TESTOUT` was not renamed, because it is exported from the package `Test`. In general, variables should never be exported, but this would be an interface change, not a refactoring.
- I fully qualify all names defined outside a package (`Carp::carp` and `Carp::croak`). This helps the reader to know what is defined locally as well as enabling him to find the implementation of or documentation for external functions quickly. I apply this guideline to perl modules. In specialized Perl scripts, such as, templates and tests, I prefer the brevity of the unqualified form. For example, in the unit test example above, I used `ok`, not `Test::ok`.
- `carp` and `croak` print the file and line number for you, so including `Test::plan` in the error string is unnecessarily redundant.
- The spelling error (`verison`) in the `$MacPerl::Version` output string was corrected.
- The two calls to `sprintf` and `scalar` are unnecessary. The concatenation operator (dot) is sufficient, more succinct, and used consistently.
- The old style call syntax (`&Win32::BuildNumber()`) was eliminated, because it was not used in all places (`_reset_globals()`).
- The comment `# Retval never used:` was removed, because it is superfluous, and it indicates an unprovable assertion. You can't know that the return value won't be used.

nagler@extremeperl.org

- The comment `# guard against -l and...` was removed, because the context of `_print` is enough to explain why the `local` call is needed.[8] Even if you don't know what `$,` and `$\` are, you know they are relevent only to the call to `print`, since that's the only thing that it could possibly affect.

9.9 Refactoring

Now kids, don't try this at work. Refactorings and small corrections are not an end to themselves. The do not add business value–unless you are writing your coding style guideline as is the case here. Refactorings need to be related to the task at hand. For example, if there I was given a story to fix the minor defects in `plan` or to add automatic test case counting, then I would have refactored the code to allow for those changes, and possibly a bit more. However, random and extensive refactoring as I've done here is worthless. The original version works just fine, and all the corrections are minor. If you spend your days refactoring and making small corrections without explicit customer approval, you'll probably lose your job.

The new `plan` is also not just a refactoring. When an interface changes, it's only a refactoring if all its uses are changed simultaneously. For public APIs like this one, that's impossible to do. In this particular case, I took a chance that the return value of `plan` was not used in this rather obscure way, that is, expecting a single list containing `undef`.

9.10 Input Validation

Perl is a dynamically typed language. The routine `plan` contains a set of type assertions, and the refactored version expanded on them. Is this the best way to write dynamically typed code?

but It depends. In this case, explicit type checking is possibly overkill. For example, the `$_TODO` and `$_ONFAIL` are dereferenced elsewhere in the package. Dereferencing a non-reference terminates execution in Perl, so the error will be caught anyway. Since `Test` is only used in test programs, it's probably sufficient to catch an error at any point.

On the other hand, `Test` is a very public API, which means it has a broad and unknown user base. Explicit type checking almost always yields more

[8] In XP, "we comment methods only after doing everything possible to make the method not need a comment." See http://xp.c2.com/ExtremeDocuments.html for a document about documentation by XP's founders.

easily understood error messages than implicit error checks. This helps users debug incorrect parameters. `plan` is only called once during a test execution so the performance impact of the additional checking is insignificant.

Here are some guidelines we use to determine when to add type assertions:

- Always validate data from untrusted sources, for example, users or third party services. It's important to give informative error messages to end users. This type of validation occurs at the outermost level of the system, where meaningful error messages can be returned with the appropriate context.

- Add type assertions to low level modules that define the data types, and leave them out at the middle levels where they would be redundant. There may be a performance trade off here. In general, the more public the API, the more important validation is. For example, `plan` defines and asserts that the test count is positive integer.

- Assert what is likely to be wrong.

- Write deviance tests, that is, tests which result in exceptions or type validation errors. Add assertions if the tests don't pass. The appropriateness of a particular type assertion is often hard to assess. Don't sweat it. You'll learn what's appropriate as your system evolves.

- Don't expect to get it right, and think about the consequences if you get it wrong. The more that's at stake, the more important assertions are.[9]

Writing robust code is hard. If you add too many assertions, their sheer volume will introduce more defects than they were intended to prevent. Add too few assertions, and one day you'll find a cracker who has compromised your system, or worse. Expect the code to evolve as it gets used.

9.11 You'd Rather Die

Nothing is more boring than reading someone's opinion about coding style. Rather than kill off my readership, I'll stop here. When you get up to stretch your legs, I'd like you to walk away with five points:

- An XP team needs a consistent coding style.

[9] Thanks to Ged Haywood for reminding me of this one.

nagler@extremeperl.org

- It doesn't matter what the style is, as long as everyone agrees to adhere to it.
- Take refactoring into consideration when determining your coding style.
- Do the simplest thing that could possibly work when writing new code.
- Simplify your design so that concepts are expressed once and only once.

nagler@extremeperl.org

Chapter 10

Logistics

> Failure is not an option. It comes bundled with the software.
>
> – Anonymous

This chapter is under construction.

Chapter 11

Test-Driven Design

> The belief that a change will be easy to do correctly makes it less likely that the change will be done correctly.
>
> – Gerald Weinberg[1]

An XP programmer writes a unit test to clarify his intentions before he makes a change. We call this test-driven design (TDD) or test-first programming, because an API's design and implementation are guided by its test cases. The programmer writes the test the way he wants the API to work, and he implements the API to fulfill the expectations set out by the test.

Test-driven design helps us invent testable and usable interfaces. In many ways, testability and usability are one in the same. If you can't write a test for an API, it'll probably be difficult to use, and vice-versa. Test-driven design gives feedback on usability before time is wasted on the implementation of an awkward API. As a bonus, the test documents how the API works, by example.

All of the above are good things, and few would argue with them. One obvious concern is that test-driven design might slow down development. It does take time to write tests, but by writing the tests first, you gain insight into the implementation, which speeds development. Debugging the implementation is faster, too, thanks to immediate and reproducible feedback that only an automated test can provide.

Perhaps the greatest time savings from unit testing comes a few months or years after you write the test, when you need to extend the API. The

[1] *Quality Software Management: Vol. 1 Systems Thinking*, Gerald Weinberg, Dorset House, 1991, p. 236.

unit test not only provides you with reliable documentation for how the API works, but it also validates the assumptions that went into the design of the API. You can be fairly sure a change didn't break anything if the change passes all the unit tests written before it. Changes that fiddle with fundamental API assumptions cause the costliest defects to debug. A comprehensive unit test suite is probably the most effective defense against such unwanted changes.

This chapter introduces test-driven design through the implementation of an exponential moving average (EMA), a simple but useful mathematical function. This chapter also explains how to use the CPAN modules `Test::More` and `Test::Exception`.

11.1 Unit Tests

A unit test validates the programmer's view of the application. This is quite different from an acceptance test, which is written from the customer's perspective and tests end-user functionality, usually through the same interface that an ordinary user uses. In constrast, a unit test exercises an API, formally known as a unit. Usually, we test an entire Perl package with a single unit test.

Perl has a strong tradition of unit testing, and virtually every CPAN module comes with one or more unit tests. There are also many test frameworks available from CPAN. This and subsequent chapters use `Test::More`, a popular and well documented test module.[2] I also use `Test::Exception` to test deviance cases that result in calls to `die`.[3]

11.2 Test First, By Intention

Test-driven design takes unit testing to the extreme. Before you write the code, you write a unit test. For example, here's the first test case for the EMA (exponential moving average) module:

```
use strict;
use Test::More tests => 1;
BEGIN {
```

[2] Part of the Test-Simple distribution, available at http://search.cpan.org/search?query=Test-Simple I used version 0.47 for this book.

[3] Version 0.15 used here. Available at http://search.cpan.org/search?query=Test-Exception

```
  use_ok(&#39;EMA&#39;);
}
```

This is the minimal `Test::More` test. You tell `Test::More` how many tests to expect, and you import the module with `use_ok` as the first test case. The `BEGIN` ensures the module's prototypes and functions are available during compilation of the rest of the unit test.

The next step is to run this test to make sure that it fails:

```
% perl -w EMA.t
1..1
not ok 1 - use EMA;
#     Failed test (EMA.t at line 4)
#     Tried to use 'EMA'.
#     Error:  Can't locate EMA.pm in @INC [trimmed]
# Looks like you failed 1 tests of 1.
```

At this stage, you might be thinking, "Duh! Of course, it fails." Test-driven design does involve lots of duhs in the beginning. The baby steps are important, because they help to put you in the mindset of writing a small test followed by just enough code to satisfy the test.

If you have maintenance programming experience, you may already be familiar with this procedure. Maintenance programmers know they need a test to be sure that their change fixes what they think is broken. They write the test and run it before fixing anything to make sure they understand a failure and that their fix works. Test-driven design takes this practice to the extreme by clarifying your understanding of all changes before you make them.

Now that we have clarified the need for a module called `EMA` (duh!), we implement it:

```
package EMA;
use strict;
1;
```

And, duh, the test passes:

```
% perl -w EMA.t
```


nagler@extremeperl.org

```
1..1
ok 1 - use EMA;
```

Yeeha! Time to celebrate with a double cappuccino so we don't fall asleep.

That's all there is to the test-driven design loop: write a test, see it fail, satisfy the test, and watch it pass. For brevity, the rest of the examples leave out the test execution steps and the concomitant duhs and yeehas. However, it's important to remember to include these simple steps when test-first programming. If you don't remember, your programming partner probably will.[4]

11.3 Exponential Moving Average

Our hypothetical customer for this example would like to maintain a running average of closing stock prices for her website. An EMA is commonly used for this purpose, because it is an efficient way to compute a running average. You can see why if you look at the basic computation for an EMA:

today's price x weight + yesterday's average x (1 - weight)

This algorithm produces a weighted average that favors recent history. The effect of a price on the average decays exponentially over time. It's a simple function that only needs to maintain two values: yesterday's average and the weight. Most other types of moving averages, require more data storage and more complex computations.

The weight, commonly called *alpha*, is computed in terms of uniform time periods (days, in this example):

2 / (number of days + 1)

For efficiency, alpha is usually computed once, and stored along with the current value of the average. I chose to use an object to hold these data and a single method to compute the average.

11.4 Test Things That Might Break

Since the first cut design calls for a stateful object, we need to instantiate it to use it. The next case tests object creation:

[4] Just a friendly reminder to program in pairs, especially when trying something new.

nagler@extremeperl.org

```
ok(EMA->new(3));
```

I sometimes forget to return the instance (`$self`) so the test calls `ok` to check that `new` returns some non-zero value. This case tests what I think might break. An alternative, more extensive test is:

```
# Not recommended: Don&#39;t test what is unlikely to break
ok(UNIVERSAL::isa(EMA->new(3), &#39;EMA&#39;));
```

This case checks that `new` returns a blessed reference of class `EMA`. To me, this test is unnecessarily complex. If `new` returns something, it's probably an instance. It's reasonable to rely on the simpler case on that basis alone. Additionally, there will be other test cases that will use the instance, and those tests will fail if `new` doesn't return an instance of class `EMA`.

This point is subtle but important, because the size of a unit test suite matters. The larger and slower the suite, the less useful it will be. A slow unit test suite means programmers will hesitate before running all the tests, and there will be more checkins which break unit and/or acceptance tests. Remember, programmers are lazy and impatient, and they don't like being held back by their programming environment. When you test only what might break, your unit test suite will remain a lightweight and effective development tool.

Please note that if you and your partner are new to test-driven design, it's probably better to err on the side of caution and to test too much. With experience, you'll learn which tests are redundant and which are especially helpful. There are no magic formulas here. Testing is an art that takes time to master.

11.5 Satisfy The Test, Don't Trick It

Returning to our example, the implementation of `new` that satisfies this case is:

```
sub new {
  my($proto, $length) = @_;
  return bless({}, ref($proto) || $proto);
}
```


nagler@extremeperl.org

This is the minimal code which satisfies the above test. `$length` doesn't need to be stored, and we don't need to compute alpha. We'll get to them when we need to.

But wait, you say, wouldn't the following code satisfy the test, too?

```
# Not recommended: Don&#39;t fake the code to satisfy the test
sub new {
  return 1;
}
```

Yes, you can trick any test. However, it's nice to treat programmers like grown-ups (even though we don't always act that way). No one is going to watch over your shoulder to make sure you aren't cheating your own test. The first implementation of `new` is the right amount of code, and the test is sufficient to help guide that implementation. The design calls for an object to hold state, and an object creation is what needed to be coded.

11.6 Test Base Cases First

What we've tested thus far are the base cases, that is, tests that validate the basic assumptions of the API. When we test basic assumptions first, we work our way towards the full complexity of the complete implementation, and it also makes the test more readable. Test-first design works best when the implementation grows along with the test cases.

There are two base cases for the `compute` function. The first base case is that the initial value of the average is just the number itself. There's also the case of inputting a value equal to the average, which should leave the average unchanged. These cases are coded as follows:

```
ok(my $ema = EMA->new(3));
is($ema->compute(1), 1);
is($ema->compute(1), 1);
```

The `is` function from `Test::More` lets us compare scalar values. Note the change to the instantiation test case that allows us to use the instance (`$ema`)

for subsequent cases. Reusing results of previous tests shortens the test, and makes it easier to understand.

The implementation that satisfies these cases is:

```
package EMA;
use strict;

sub new {
  my($proto, $length) = @_;
  return bless({
    alpha => 2 / ($length + 1),
  }, ref($proto) || $proto);
}

sub compute {
  my($self, $value) = @_;
  return $self->{avg} = defined($self->{avg})
    ? $value * $self->{alpha} + $self->{avg} * (1 - $self->{alpha})
    : $value;
}

1;
```

The initialization of `alpha` was added to `new`, because `compute` needs the value. `new` initializes the state of the object, and `compute` implements the EMA algorithm. `$self->{avg}` is initially `undef` so that case can be detected.

Even though the implementation looks finished, we aren't done testing. The above code might be defective. Both `compute` test cases use the same value, and the test would pass even if, for example, `$self->{avg}` and `$value` were accidentally switched. We also need to test that the average changes when given different values. The test as it stands is too static, and it doesn't serve as a good example of how an EMA works.

11.7 Choose Self-Evident Data

In a test-driven environment, programmers use the tests to learn how the API works. You may hear that XPers don't like documentation. That's not

quite true. What we prefer is self-validating documentation in the form of tests. We take care to write tests that are readable and demonstrate how to use the API.

One way to create readable tests is to pick good test data. However, we have a little bootstrapping problem: To pick good test data, we need valid values from the results of an EMA computation, but we need an EMA implementation to give us those values. One solution is to calculate the EMA values by hand. Or, we could use another EMA implementation to come up with the values. While either of these choices would work, a programmer reading the test cases would have to trust them or to recompute them to verify they are correct. Not to mention that we'd have to get the precision exactly right for our target platform.

11.8 Use The Algorithm, Luke!

A better alternative is to work backwards through the algorithm to figure out some self-evident test data.[5] To accomplish this, we treat the EMA algorithm as two equations by fixing some values. Our goal is to have integer values for the results so we avoid floating point precision issues. In addition, integer values make it easier for the programmer to follow what is going on.

When we look at the equations, we see alpha is the most constrained value:

> today's average = today's price x alpha + yesterday's average x (1 - alpha)

where:

> alpha = 2 / (length + 1)

Therefore it makes sense to try and figure out a value of alpha that can produce integer results given integer prices.

Starting with length 1, the values of alpha decrease as follows: 1, 2/3, 1/2, 2/5, 1/3, 2/7, and 1/4. The values 1, 1/2, and 2/5 are good candidates, because they can be represented exactly in binary floating point. 1 is a degenerate case, the average of a single value is always itself. 1/2 is not ideal, because `alpha` and `1 - alpha` are identical, which creates a symmetry in the first equation:

> today's average = today's price x 0.5 + yesterday's average x 0.5

[5] Thanks to Ion Yadigaroglu for teaching me this technique.

nagler@extremeperl.org

We want asymmetric weights so that defects, such as swapping today's price and yesterday's average, will be detected. A length of 4 yields an alpha of 2/5 (0.4), and makes the equation asymmetric:

today's average = today's price x 0.4 + yesterday's average x 0.6

With alpha fixed at 0.4, we can pick prices that make today's average an integer. Specifically, multiples of 5 work nicely. I like prices to go up, so I chose 10 for today's price and 5 for yesterday's average. (the initial price). This makes today's average equal to 7, and our test becomes:

```
ok(my $ema = EMA->new(4));
is($ema->compute(5), 5);
is($ema->compute(5), 5);
is($ema->compute(10), 7);
```

Again, I revised the base cases to keep the test short. Any value in the base cases will work so we might as well save testing time through reuse.

Our test and implementation are essentially complete. All paths through the code are tested, and `EMA` could be used in production if it is used properly. That is, `EMA` is complete if all we care about is conformant behavior. The implementation currently ignores what happens when `new` is given an invalid value for `$length`.

11.9 Fail Fast

Although `EMA` is a small part of the application, it can have a great impact on quality. For example, if `new` is passed a `$length` of -1, Perl throws a divide-by-zero exception when alpha is computed. For other invalid values for `$length`, such as -2, `new` silently accepts the errant value, and `compute` faithfully produces non-sensical values (negative averages for positive prices). We can't simply ignore these cases. We need to make a decision about what to do when `$length` is invalid.

One approach would be to assume garbage-in garbage-out. If a caller supplies -2 for `$length`, it's the caller's problem. Yet this isn't what Perl's divide function does, and it isn't what happens, say, when you try to dereference a scalar which is not a reference. The Perl interpreter calls `die`, and I've already mentioned in the Coding Style chapter that I prefer failing fast rather than waiting until the program can do some real damage. In our

example, the customer's web site would display an invalid moving average, and one her customers might make an incorrect investment decision based on this information. That would be bad. It is better for the web site to return a server error page than to display misleading and incorrect information.

Nobody likes program crashes or server errors. Yet calling `die` is an efficient way to communicate semantic limits (couplings) within the application. The UI programmer, in our example, may not know that an EMA's length must be a positive integer. He'll find out when the application dies. He can then change the design of his code and the EMA class to make this limit visible to the end user. Fail fast is an important feedback mechanism. If we encounter an unexpected `die`, it tells us the application design needs to be improved.

11.10 Deviance Testing

In order to test for an API that fails fast, we need to be able to catch calls to `die` and then call `ok` to validate the call did indeed end in an exception. The function `dies_ok` in the module `Test::Exception` does this for us.

Since this is our last group of test cases in this chapter, here's the entire unit test with the changeds for the new deviance cases highlighted:

```
use strict;
use Test::More tests => 9; use Test::Exception;
BEGIN {
  use_ok(&#39;EMA&#39;);
}
ok(my $ema = EMA->new(4));
is($ema->compute(5), 5);
is($ema->compute(5), 5);
is($ema->compute(10), 7); dies_ok {EMA->new(-2)}; dies_ok {EMA->new(0)};
lives_ok {EMA->new(1)}; dies_ok {EMA->new(2.5)};
```

There are now 9 cases in the unit test. The first deviance case validates that `$length` can't be negative. We already know -1 will die with a divide-by-zero exception so -2 is a better choice. The zero case checks the boundary condition. The first valid length is 1. Lengths must be integers, and 2.5 or any other floating point number is not allowed. `$length` has no explicit upper limit. Perl automatically converts integers to floating point numbers

nagler@extremeperl.org

if they are too large. The test already checks that floating point numbers are not allowed so no explicit upper limit check is required.

The implementation that satisfies this test follows:

```
package EMA;
use strict;

sub new {
  my($proto, $length) = @_; die("$length: length must be a positive
32-bit integer") unless $length =~ /^\d+$/ && $length >= 1 && $length
<= 0x7fff_ffff;
  return bless({
    alpha => 2 / ($length + 1),
  }, ref($proto) || $proto);
}

sub compute {
  my($self, $value) = @_;
  return $self->{avg} = defined($self->{avg})
    ? $value * $self->{alpha} + $self->{avg} * (1 - $self->{alpha})
    : $value;
}
1;
```

The only change is the addition of a call to `die` with an `unless` clause. This simple fail fast clause doesn't complicate the code or slow down the API, and yet it prevents subtle errors by converting an assumption into an assertion.

11.11 Only Test The New API

One of the most difficult parts of testing is to know when to stop. Once you have been test-infected, you may want to keep on adding cases to be sure that the API is "perfect". For example, a interesting test case would be to pass a NaN (Not a Number) to `compute`, but that's not a test of `EMA`. The floating point implementation of Perl behaves in a particular way with

respect to NaNs[6], and `Bivio::Math::EMA` will conform to that behavior. Testing that NaNs are handled properly is a job for the Perl interpreter's test suite.

Every API relies on a tremendous amount of existing code. There isn't enough time to test all the existing APIs and your new API as well. Just as an API should separate concerns so must a test. When testing a new API, your concern should be that API and no others.

11.12 Solid Foundation

In XP, we do the simplest thing that could possibly work so we can deliver business value as quickly as possible. Even as we write the test and implementation, we're sure the code will change. When we encounter a new customer requirement, we refactor the code, if need be, to facilitate the additional function. This iterative process is called continuous design, which is the subject of the next chapter. It's like renovating your house whenever your needs change. [7]

A system or house needs a solid foundation in order to support continuous renovation. Unit tests are the foundation of an XP project. When designing continuously, we make sure the house doesn't fall down by running unit tests to validate all the assumptions about an implementation. We also grow the foundation before adding new functions. Our test suite gives us the confidence to embrace change.

[6] In some implementations, use of NaNs will cause a run-time error. In others, they will cause all subsequent results to be a NaN.

[7] Don't let the thought of continuous house renovation scare you off. Programmers are much quieter and less messy than construction workers.

nagler@extremeperl.org

Chapter 12

Continuous Design

> In the beginning was simplicity.
>
> – Richard Dawkins[1]

Software evolves. All systems are adapted to the needs of their users and the circumstances in which they operate, even after years of planning.[2] Some people call this maintenance programming, implementing change requests, or, simply, firefighting. In XP, it's called continuous design, and it's the only way we design and build systems. Whatever you call it, change happens, and it involves two activities: changing what the code does and improving its internal structure.

In XP, these two activities have names: implementing stories and refactoring. Refactoring is the process of making code better without changing its external behavior. The art of refactoring is a fundamental skill in programming. It's an important part of the programmer's craft to initiate refactorings to accommodate changes requested by the customer. In XP, we use tests to be sure the behavior hasn't changed.

As any implementation grows, it needs to be refactored as changes (new features or defect fixes) are introduced. Sometimes we refactor before implementing a story, for example, to expose an existing algorithm as its own API. Other times, we refactor after adding a new feature, because we only see how to eliminate unnecessary duplication, after the feature is implemented. This to and fro of code expansion (implementing stories) and contraction (refactoring) is how the design evolves continuously. And, by the way, this is

[1] *The Selfish Gene*, Richard Dawkins, Oxford University Press, 1989, p. 12.

[2] The most striking and recent example was the failure, debugging, and repair of the Mars Exploration Rover Spirit.

how Perl was designed: on demand and continuously. It's one of the reasons Perl continues to grow and thrive while other languages whither and die.

This chapter evolves the design we started in Test-Driven Design. We introduce refactoring by simplifying the EMA equation. We add a new class (simple moving average) to satisfy a new story, and then we refactor the two classes to share a common base class. Finally, we fix a defect by exposing an API in both classes, and then we refactor the APIs into a single API in the base class.

12.1 Refactoring

The first step in continous design is to be sure you have a test. You need a test to add a story, and you use existing tests to be sure you don't break anything with a refactoring. This chapter picks up where Test-Driven Design left off. We have a working exponentional moving average (EMA) module with a working unit test.

The first improvement is a simple refactoring. The equation in `compute` is more complex than it needs to be:

```
sub compute {
  my($self, $value) = @_;
  return $self->{avg} = defined($self->{avg})
    ? $value * $self->{alpha} + $self->{avg} * (1 - $self->{alpha})
    : $value;
}
```

The refactored equation yields the same results and is simpler:

```
sub compute {
  my($self, $value) = @_;
  return $self->{avg} += defined($self->{avg})
    ? $self->{alpha} * ($value - $self->{avg})
    : $value;
}
```

After the refactoring, we run our test, and it passes. That's all there is to refactoring. Change the code, run the test for the module(s) we are mod-

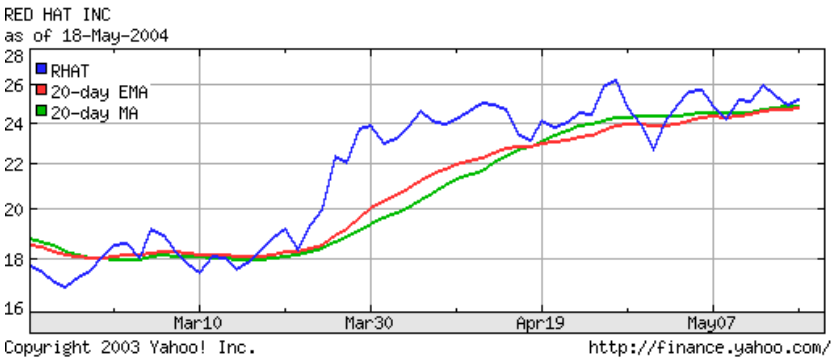
nagler@extremeperl.org

ifying, run the entire unit test suite, and then check in once all tests pass. Well, it's not always that easy, sometimes we make mistakes. That's what the tests are for, and tests are what simplifies refactoring.

12.2 Simple Moving Average

Our hypothetical customer would like to expand her website to compete with Yahoo! Finance. The following graph shows that Yahoo! offers two moving averages:

Yahoo! 20 day moving averages on 3 month graph from May 18, 2004[3]



In order to provide the equivalent functionality, we need to implement a simple moving average (SMA or MA in Yahoo!'s graph). An SMA is the arithmetic mean of the last N periods of the price series. For a daily graph, we add in the new day's price and we remove the oldest price from the sum before we take the average.

12.3 SMA Unit Test

The following test demonstrates the algorithm. The test was created by starting with a copy of the EMA test from the Test-Driven Design chapter. We replaced EMA with SMA, and changed the values to match the SMA

[3] http://finance.yahoo.com/q/ta?s=RHAT&t=3m&p=e20,m20

algorithm:

```
use strict;
use Test::More tests => 11;
use Test::Exception;
BEGIN {
  use_ok(&#39;SMA&#39;);
}
ok(my $sma = SMA->new(4));
is($sma->compute(5), 5);
is($sma->compute(5), 5);
is($sma->compute(11), 7);
is($sma->compute(11), 8);
is($sma->compute(13), 10);
dies_ok {SMA->new(-2)};
dies_ok {SMA->new(0)};
lives_ok {SMA->new(1)};
dies_ok {SMA->new(2.5)};
```

Like the EMA, the SMA stays constant (5) when the input values remain constant (5). The deviance cases are identical, which gives us another clue that the two algorithms have a lot in common. The difference is that average value changes differently, and we need to test the boundary condition when values "fall off the end" of the average.

12.4 SMA Implementation

The EMA and the SMA unit test are almost identical. It follows that the implementations should be nearly identical. Some people might want to create a base class so that *SMA* and `EMA` could share the common code. However, at this stage, we don't know what that code might be. That's why we do the simplest thing that could possibly work, and copy the `EMA` class to the `SMA` class. And, let's run the test to see what happens after we change the package name from `EMA` to `SMA`:

```
1..11
ok 1 - use SMA;
ok 2
```


nagler@extremeperl.org

```
ok 3
ok 4
not ok 5
#     Failed test (SMA.t at line 10)
#          got: &#39;7.4&#39;
#     expected: &#39;7&#39;
not ok 6
#     Failed test (SMA.t at line 11)
#          got: &#39;8.84&#39;
#     expected: &#39;8&#39;
not ok 7
#     Failed test (SMA.t at line 12)
#          got: &#39;10.504&#39;
#     expected: &#39;10&#39;
ok 8
ok 9
ok 10
ok 11
# Looks like you failed 3 tests of 11.
```

The test fails, because an EMA algorithm in an SMA's clothing is still an EMA. That's good. Otherwise, this section would be way too short. Without further ado, here's the correct algorithm:

```
package SMA;
use strict;

sub new {
  my($proto, $length) = @_;
  die("$length: length must be a positive 32-bit integer")
    unless $length =~ /^\d+$/ && $length >= 1 && $length <= 0x7fff_ffff;
  return bless({
    length => $length,
    values => [],
  }, ref($proto) || $proto);
}

sub compute {
  my($self, $value) = @_;
```


nagler@extremeperl.org

```
  $self->{sum} -= shift(@{$self->{values}})
    if $self->{length} eq @{$self->{values}};
  return ($self->{sum} += $value) / push(@{$self->{values}}, $value);
}

1;
```

The `sum` calculation is different, but the basic structure is the same. The `new` method checks to makes sure that `length` is reasonable. We need to maintain a queue of all values in the sum, because an SMA is a FIFO algorithm. When a value is more than `length` periods old, it has absolutely no affect on the average. As an aside, the SMA algorithm pays a price for that exactness, because it must retain `length` values where EMA requires only one. That's the main reason why EMAs are more popular than SMAs in financial engineering applications.

For our application, what we care about is that this implementation of SMA satisfies the unit test. We also note that EMA and SMA have a lot in common. However, after satisfying the SMA test, we run all the unit tests to be sure we didn't inadvertently modify another file, and then we checkin to be sure we have a working baseline. Frequent checkins is important when designing continuously. Programmers have to be free to make changes knowing that the source repository always holds a recent, correct implementation.

12.5 Move Common Features to a Base Class

The SMA implementation is functionally correct, but it isn't a good design. The quick copy-and-paste job was necessary to start the implementation, and now we need to go back and improve the design through a little refactoring. The classes `SMA` and `EMA` can and should share code. We want to represent each concept once and only once so we only have to fix defects in the implementation of the concepts once and only once.

The repetitive code is contained almost entirely in the `new` methods of `SMA` and `EMA`. The obvious solution is to create a base class from which `SMA` and `EMA` are derived. This is a very common refactoring, and it's one you'll use over and over again.

Since this is a refactoring, we don't write a new test. The refactoring must not change the observable behavior. The existing unit tests validate

nagler@extremeperl.org

that the behavior hasn't changed. That's what differentiates a refactoring from simply changing the code. Refactoring is the discipline of making changes to improve the design of existing code without changing the external behavior of that code.

The simple change we are making now is moving the common parts of `new` into a base class called `MABase`:

```
package MABase;
use strict;

sub new {
  my($proto, $length, $fields) = @_;
  die("$length: length must be a positive 32-bit integer")
    unless $length =~ /^\d+$/ && $length >= 1 && $length <= 0x7fff_ffff;
  return bless($fields, ref($proto) || $proto);
}

1;
```

The corresponding change to `SMA` is:

```
use base &#39;MABase&#39;;
sub new {
  my($proto, $length) = @_;
  return $proto->SUPER::new($length, {
    length => $length,
    values => [],
  });
}
```

For brevity, I left out the the `EMA` changes, which are similar to these. Note that `MABase` doesn't share fields between its two subclasses. The only common code is checking the length and blessing the instance is shared.

nagler@extremeperl.org

12.6 Refactor the Unit Tests

After we move the common code into the base clase, we run all the existing tests to be sure we didn't break anything. However, we aren't done. We have a new class, which deserves its own unit test. `EMA.t` and `SMA.t` have four cases in common. That's unnecessary duplication, and here's `MABase.t` with the cases factored out:

```
use strict;
use Test::More tests => 5;
use Test::Exception;
BEGIN {
  use_ok(&#39;MABase&#39;);
}
dies_ok {MABase->new(-2, {})};
dies_ok {MABase->new(0, {})};
lives_ok {MABase->new(1, {})};
dies_ok {MABase->new(2.5, {})};
```

After running the new test, we refactor `EMA.t` (and similarly `SMA.t`) as follows:

```
use strict;
use Test::More tests => 5;
BEGIN {
  use_ok(&#39;EMA&#39;);
}
ok(my $ema = EMA->new(4));
is($ema->compute(5), 5);
is($ema->compute(5), 5);
is($ema->compute(10), 7);
```

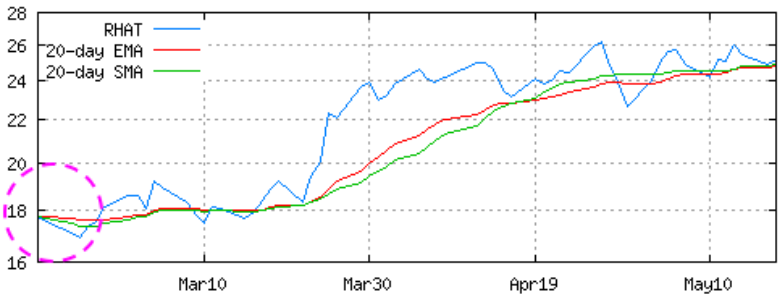
By removing the redundancy, we make the classes and their tests cohesive. `MABase` and its test is concerned with validating `$length`. `EMA` and `SMA` are responsible for computing moving averages. This conceptual clarity, also known as cohesion, is what we strive for.

nagler@extremeperl.org

12.7 Fixing a Defect

The design is better, but it's wrong. The customer noticed the difference between the Yahoo! graph and the one produced by the algorithms above:

Incorrect moving average graph



The lines on this graph start from the same point. On the Yahoo! graph in the SMA Unit Test, you see that the moving averages don't start at the same value as the price. The problem is that a 20 day moving average with one data point is not valid, because the single data point is weighted incorrectly. The results are skewed towards the initial prices.

The solution to the problem is to "build up" the moving average data before the initial display point. The build up period varies with the type of moving average. For an SMA, the build up length is the same as the length of the average minus one, that is, the average is correctly weighted on the "length" price. For an EMA, the build up length is usually twice the length, because the influence of a price doesn't simply disappear from the average after length days. Rather the price's influence decays over time.

The general concept is essentially the same for both averages. The algorithms themselves aren't different. The build up period simply means that we don't want to display the prices. separate out compute and value. Compute returns undef. value blows up. is_ok or will_compute_ok? The two calls are inefficent, but the design is simpler. Show the gnuplot code to generate the graph. gnuplot reads from stdin? The only difference is that the two algorithms have different build up lengths. The easiest solution is therefore to add a field in the sub-classes which the base classes exposes via a method called `build_up_length`. We need to expand our tests first:

```
use strict;
use Test::More tests => 6;
```

```
BEGIN {
  use_ok(&#39;EMA&#39;);
}
ok(my $ema = EMA->new(4)); is($ema->build_up_length, 8);
is($ema->compute(5), 5);
is($ema->compute(5), 5);
is($ema->compute(10), 7);
```

The correct answer for `EMA` is always two times `length`. It's simple enough that we only need one case to test it. The change to `SMA.t` is similar.

To satisfy these tests, we add `build_up_length` to `MABase`:

```
sub build_up_length {
  return shift->{build_up_length};
}
```

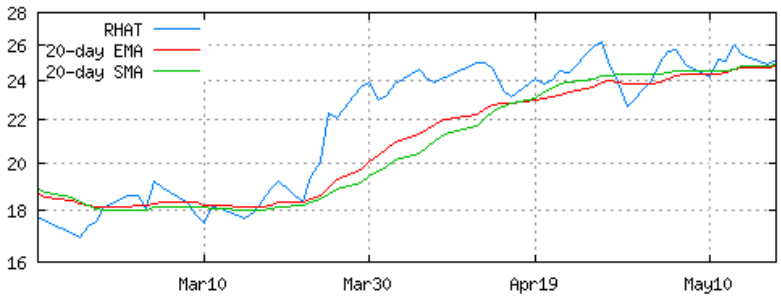
The computation of the value of `build_up_length` requires a change to `new` in EMA:

```
sub new {
  my($proto, $length) = @_;
  return $proto->SUPER::new($length, {
    alpha => 2 / ($length + 1), build_up_length => $length * 2,
  });
}
```

The change to SMA is similar, and left out for brevity. After we fix the plotting code to reference `build_up_length`, we end up with the following graph:

Moving average graph with correction for build up period

nagler@extremeperl.org



12.8 Global Refactoring

After releasing the build up fix, our customer is happy again. We also have some breathing room to fix up the design again. When we added `build_up_length`, we exposed a configuration value via the moving average object. The plotting module also needs the value of `length` to print the labels ("20-day EMA" and "20-day SMA") on the graph. This configuration value is passed to the moving average object, but isn't exposed via the `MABase` API. That's bad, because `length` and `build_up_length` are related configuration values. The plotting module needs both values.

To test this feature, we add a test to `SMA.t` (and similarly, to `EMA.t`):

```
use strict;
use Test::More tests => 8;
BEGIN {
  use_ok(&#39;SMA&#39;);
}
ok(my $sma = SMA->new(4));
is($sma->build_up_length, 3); is($sma->length, 4);
is($sma->compute(5), 5);
is($sma->compute(5), 5);
is($sma->compute(11), 7);
is($sma->compute(11), 8);
```

We run the test to see that indeed `length` does not exist in `MABase` or its subclasses. Then we add `length` to `MABase`:

```
sub length {
```

```
  return shift->{length};
}
```

`SMA` already has a field called `length` so we only need to change `EMA` to store the `length`:

```
sub new {
  my($proto, $length) = @_;
  return $proto->SUPER::new($length, {
    alpha => 2 / ($length + 1),
    build_up_length => $length * 2, length => $length,
  });
}
```

This modification is a refactoring even though external behavior (the API) is different. When an API and all its clients (importers) change, it's called a global refactoring. In this case, the global refactoring is backwards compatible, because we are adding new behavior. The clients were using a copy of `length` implicitly. Adding an explicit `length` method to the API change won't break that behavior. However, this type of global refactoring can cause problems down the road, because old implicit uses of `length` still will work until the behavior of the `length` method changes. At which point, we've got to know that the implicit coupling is no longer valid.

That's why tests are so important with continuous design. Global refactorings are easy when each module has its own unit test and the application has an acceptance test suite. The tests will more than likely catch the case where implicit couplings go wrong either at the time of the refactoring or some time later. Without tests, global refactorings are scary, and most programmers don't attempt them. When an implicit coupling like this becomes cast in stone, the code base is a bit more fragile, and continous design is a bit harder. Without some type of remediation, the policy is "don't change anything", and we head down the slippery slope that some people call Software Entropy.[4]

[4] Software Entropy is often defined as software that "loses its original design structure" (http://www.webopedia.com/TERM/S/software_entropy.html). Continuous design turns the concept of software entropy right side up (and throws it right out the window) by changing the focus from the code to what the software is supposed to do. Software entropy is meaningless when there are tests that specify the expected behavior for all parts of an

application. The tests eliminate the fear of change inherent in non-test-driven software methodologies.

nagler@extremeperl.org

12.9 Continuous Rennovation in the Real World

Programmers often use building buildings as a metaphor for creating software. It's often the wrong model, because it's not easy to copy-and-paste. The physical world doesn't allow easy replication beyond the gene level. However, continuous design is more commonplace than many people might think. My company had our office rennovated before we moved in. Here is a view of the kitchen (and David Farber) before the upgrade:

Before rennovation

After the rennovation, the kitchen looked like this:

After rennovation

12.10 Simplify Accessors

Software entropy creeps in when the software fails to adapt to a change. For example, we now have two accessors that are almost identical:

```
sub length {
  return shift->{length};
}

sub build_up_length {
  return shift->{build_up_length};
}
```

The code is repeated. That's not a big problem in the specific, because we've only done it twice. This subtle creep gets to be a bigger problem when someone else copies what we've done here. Simple copy-and-paste is probably the single biggest cause of software rot in any system. New programmers on the project think that's how "we do things here", and we've got a standard practice for copying a single error all over the code. It's not that this particular code is wrong; it's that the practice is wrong. This is why it's important to stamp out the practice when you can, and in this case it's very easy to do.

We can replace both accessors with a single new API called `get`. This global refactoring is very easy, because we are removing an existing API. That's another reason to make couplings explicit: when the API changes, all uses fail with method not found. The two unit test cases for EMA now become:

```
is($ema->get(&#39;build_up_length&#39;), 8);
is($ema->get(&#39;length&#39;), 4);
```

And, we replace `length` and `build_up_length` with a single method:

```
sub get {
  return shift->{shift(@_)};
}
```

We also refactor uses of `build_up_length` and `length` in the plotting module. This is the nature of continuous rennovation: constant change everywhere. And, that's the part that puts people off. They might ask why the last two changes (adding `length` and refactoring `get`) were necessary.

12.11 Change Happens

Whether you like it or not, change happens. You can't stop it. If you ignore it, your software becomes brittle, and you have more (boring and high stress) work to do playing catch up with the change. The proactive practices of testing and refactoring seem unnecessary until you do hit that defect that's been copied all over the code, and you are forced to fix it. Not only is it difficult to find all copies of an error, but you also have to find all places in the code which unexpectedly depended on the behavior caused by the defect. Errors multiply so that's changing a single error into N-squared errors It gets even worse when the practice of copy-and-pasting is copied. That's N-cubed, and that will make a mess of even the best starting code base. Refactoring when you see replication is the only way to eliminate this geometric effect.

Without tests, refactoring is no longer engineering, it's hacking. Even the best hackers hit a wall if they can't validate a change hasn't broken something. They'll create ad hoc tests if necessary. XP formalizes this process. However, unit testing is still an art. The only way to get good at it is by seeing more examples. The next chapter takes a deeper look at the unit testing in a more realistic environment.

Chapter 13

Unit Testing

> A successful test case is one that detects an as-yet undiscovered error.
>
> – Glenford Myers[1]

The second and third examples test a post office protocol (POP3) client available from CPAN. These two unit tests for `Mail::POP3Client` indicate some design issues, which are addressed in the Refactoring chapter. The third example also demonstrates how to use `Test::MockObject`, a CPAN module that makes it easy to test those tricky paths through the code, such as, error cases.

13.1 Testing Isn't Hard

One of the common complaints I've heard about testing is that it is too hard for complex APIs, and the return on investment is therefore too low. The problem of course is the more complex the API, the more it needs to be tested in isolation. The rest of the chapter demonstrates a few tricks that simplify testing complex APIs. What I've found, however, the more testing I do, the easier it is to write tests especially for complex APIs.

Testing is also infectious. As your suite grows, there are more examples to learn from, and the harder it becomes to *not* test. Your test infrastructure also evolves to better match the language of your APIs. Once and only once applies to test software, too. This is how `Bivio::Test` came about. We were tired of repeating ourselves. `Bivio::Test` lets us write subject matter oriented programs, even for complex APIs.

[1] *Art of Software Testing*, Glenford Myers, John Wiley & Sons, 1979, p. 16.

13.2 Mail::POP3Client

The POP3 protocol[2] is a common way for mail user agents to retrieve messages from mail servers. As is often the case, there's a CPAN module available that implements this protocol.

Mail::POP3Client[3] has been around for a few years. The unit test shown below was written in the spirit of test first programming. Some of the test cases fail, and in Refactoring, we refactor Mail::POP3Client to make it easier to fix some of the defects found here.

This unit test shows how to test an interface that uses sockets to connect to a server and has APIs that write files. This test touches on a number of test and API design issues.

To minimize page flipping the test is broken into pieces, one part per section. The first two sections discuss initialization and data selection. In Validate Basic Assumptions First and the next section, we test the server capabilities and authentication mechanisms match our assumptions. We test basic message retrieval starting in Distinguish Error Cases Uniquely followed by retrieving to files. The List, ListArray, and Uidl methods are tested in Relate Results When You Need To. Destructive tests (deletion) occur next after we have finished testing retrieval and listing. We validate the accessors (Host, Alive, etc.) in Consistent APIs Ease Testing. The final test cases cover failure injection.

13.3 Make Assumptions

```
use strict;
use Test::More tests => 85;
use IO::File;
use IO::Scalar;
BEGIN {
  use_ok(&#39;Mail::POP3Client&#39;);
}
```

[2] The Post Office Protocol - Version 3 RFC can be found at http://www.ietf.org/rfc/rfc1939.txt. The Mail::POP3Client also implements the POP3 Extension Mechanism RFC, http://www.ietf.org/rfc/rfc2449.txt, and IMAP/POP AUTHorize Extension for Simple Challenge/Response RFC http://www.ietf.org/rfc/rfc2195.txt.

[3] The version being tested here is 2.12, which can be found at http://search.cpan.org/author/SDOWD/POP3Client-2.12.

nagler@extremeperl.org

```
my($cfg) = {
  HOST => &#39;localhost&#39;,
  USER => &#39;pop3test&#39;,
  PASSWORD => &#39;password&#39;,
};
```

To access a POP3 server, you need an account, password, and the name of the host running the server. We made a number of assumptions to simplify the test without compromising the quality of the test cases. The POP3 server on the local machine must have an account `pop3test`, and it must support APOP, CRAM-MD5, CAPA, and UIDL.

The test that comes with `Mail::POP3Client` provides a way of configuring the POP3 configuration via environment variables. This makes it easy to run the test in a variety of environments. The purpose of that test is to test the basic functions on any machine. For a CPAN module, you need this to allow anybody to run the test. A CPAN test can't make a lot of assumptions about the execution environment.

In test-first programming, the most important step is writing the test. Make all the assumptions you need to get the test written and working. Do the simplest thing that could possibly work, and assume you aren't going to need to write a portable test. If you decide to release the code and test to CPAN, relax the test constraints after your API works. Your first goal is to create the API which solves your customer's problem.

13.4 Test Data Dependent Algorithms

```
my($subject) = "Subject: Test Subject";
my($body) = <<&#39;EOF&#39;;
Test Body
A line with a single dot follows
.
And a dot and a space
.
EOF

open(MSG, "| /usr/lib/sendmail -i -U $cfg->{USER}\@$cfg->{HOST}");
print(MSG $subject . "\n\n" . $body);
close(MSG)
```


nagler@extremeperl.org

```
  or die("sendmail failed: $!");
sleep(1);

my($body_lines) = [split(/\n/, $body)];
$body = join("\r\n", @$body_lines, &#39;&#39;);
```

The POP3 protocol uses a dot (.) to terminate multi-line responses. To make sure `Mail::POP3Client` handles dots correctly, we put leading dots in the message body. The message should be retrieved in its entirety, including the lines with dots. It's important to test data dependencies like this.

The test only sends one message. This is sufficient to validate the client implementation. Testing the server, however, would be much more complex, and would require multiple clients, messages, and message sizes.

The `sleep(1)` is used to give `sendmail` time to deliver the message before the test starts.

13.5 Validate Basic Assumptions First

```
my($pop3) = Mail::POP3Client->new(HOST => $cfg->{HOST});
$pop3->Connect;
is($pop3->State, &#39;AUTHORIZATION&#39;);
like($pop3->Capa, qr/UIDL.*CRAM.*|CRAM.*UIDL/is);
ok($pop3->Close);
```

The first case group validates some assumptions used in the rest of the cases. It's important to put these first to aid debugging. If the entire test fails catastrophically (due to a misconfigured server, for example), it's much easier to diagnose the errors when the basic assumptions fail first.

`Bivio::Test` allows you to ignore the return result of conformance cases by specifying `undef`. The return value of `Connect` is not well-defined, so it's unimportant to test it, and the test documents the way the API works.

This case raises a design issue. Perl subroutines always return a value. `Connect` does not have an explicit `return` statement, which means it returns an arbitrary value. Perl has no implicit `void` context like C and Java do. It's always safe to put in an explicit `return;` in subroutines when you don't intend to return anything. This helps ensure predictable behavior in any

nagler@extremeperl.org

calling context, and improves testability.

The second case tests the server supports CAPA (capabilities), UIDL (unique identifiers), and CRAM (challenge/response authentication). The capability list is unordered so we check the list for UIDL then CRAM or the reverse. `Bivio::Test` allows us to specify a `Regexp` instance (`qr//`) as the expected value. The case passes if the expected regular expression matches the actual return, which is serialized by `Data::Dumper`.

13.6 Validate Using Implementation Knowledge

```
foreach my $mode (qw(BEST APOP CRAM-MD5 PASS)) {
  $pop3 = Mail::POP3Client->new(%$cfg, AUTH_MODE => $mode);
  is_deeply([$pop3->Body(1)], $body_lines);
  is($pop3->Close, 1);
}

$pop3 = Mail::POP3Client->new(%$cfg, AUTH_MODE => &#39;BAD-MODE&#39;);
like($pop3->Message, qr/BAD-MODE/);
is($pop3->State, &#39;AUTHORIZATION&#39;);
is($pop3->Close, 1);

$pop3 = Mail::POP3Client->new(
  %$cfg, AUTH_MODE => &#39;BEST&#39;, PASSWORD => &#39;BAD-PASSWORD&#39;);
like($pop3->Message, qr/PASS failed/);
is($pop3->State, &#39;AUTHORIZATION&#39;);
is($pop3->Close, 1);

$pop3 = Mail::POP3Client->new(
  %$cfg, AUTH_MODE => &#39;APOP&#39;, PASSWORD => &#39;BAD-PASSWORD&#39;);
like($pop3->Message, qr/APOP failed/);
is($pop3->Close, 1);
```

Once we have validated the server's capabilities, we test the authentication interface. `Mail::POP3Client` defaults to `AUTH_MODE BEST`, but we test each mode explictly here. The other cases test the default mode. To be sure authentication was successful, we download the body of the first message and compare it with the value we sent. POP3 authentication implies

nagler@extremeperl.org

authorization to access your messages. We only know we are authorized if we can access the mail user's data.

In `BEST` mode the implementation tries all authentication modes with `PASS` as the last resort. We use knowledge of the implementation to validate that `PASS` is the last mode tried. The `Message` method returns `PASS failed`, which gives the caller information about which `AUTH_MODE` was used.

The test doesn't know the details of the conversation between the server and client, so it assumes the implementation doesn't have two defects (using `PASS` when it shouldn't and returning incorrect `Message` values). We'll see in Mock Objects how to address this issue without such assumptions.

The authentication conformance cases are incomplete, because there might be a defect in the authentication method selection logic. We'd like know if we specify `APOP` that `Mail::POP3Client` doesn't try `PASS` first. The last case group in this section attempts to test this, and uses the knowledge that `Message` returns `APOP failed` when `APOP` fails. Again, it's unlikely `Message` will return the wrong error message.

13.7 Distinguish Error Cases Uniquely

```
sub _is_match {
  my($actual, $expect) = @_;
  return ref($expect) eq &#39;Regexp&#39;
    ? like(ref($actual) ? join(&#39;&#39;, @$actual) : $actual, $expect)
    : is_deeply($actual, $expect);
}

$pop3 = Mail::POP3Client->new(%$cfg);
foreach my $params (
  [Body => $body_lines],
  [Head => qr/\Q$subject/],
  [HeadAndBody => qr/\Q$subject\E.*\Q$body_lines->[0]/s],
) {
  my($method, $expect) = @$params;
  _is_match([$pop3->$method(1)], $expect);
  is($pop3->Message(&#39;&#39;), &#39;&#39;);
  is_deeply([$pop3->$method(999)], []);
  like($pop3->Message, qr/No such message|Bad message number/i);
}
```


nagler@extremeperl.org

The `Body` method returns the message body, `Head` returns the message head, and `HeadAndBody` returns the entire message. We assume that `999` is a valid message number and that there aren't 999 messages in the mailbox.

`Body` returns an empty array when a message is not found. Should `Body` return something else or `die` in the deviance case? I think so. Otherwise, an empty message body is indistinguishable from a message which isn't found. The deviance test identifies this design issue. That's one reason why deviance tests are so important.

To workaround this problem, we clear the last error `Message` saved in the `Mail::POP3Client` instance before calling the download method. We then validate that `Message` is set (non-blank) after the call.

The test case turned out to be successful unexpectedly. It detected a defect in `Message`: You can't clear an existing `Message`. This is a side-effect of the current test, but a defect nonetheless. One advantage of validating the results of every call is that you get bonuses like this without trying.

13.8 Avoid Context Sensitive Returns

```
foreach my $params (
  [Body => $body],
  [Head => qr/\Q$subject/],
  [HeadAndBody => qr/\Q$subject\E.*\Q$body/s],
) {
  my($method, $expect) = @$params;
  _is_match(scalar($pop3->$method(1)), $expect);
  is(scalar($pop3->$method(999)), undef);
}
```

When `Body`, `Head`, and `HeadAndBody` are invoked in a scalar context, the result is a single string, and `undef` is returned on errors, which simplifies deviance testing. (Note that `Bivio::Test` distinguishes `undef` from `[undef]`. The former ignores the result, and the latter expects a single-valued result of `undef`.)

`Bivio::Test` invokes methods in a list context by default. Setting `want_scalar` forces a scalar context. This feature was added to test non-

bOP classes like `Mail::POP3Client`.

In bOP, methods are invocation context insensitive. Context sensitive returns like `Body` are problematic.[4] We use `wantarray` to ensure methods that return lists behave identically in scalar and list contexts. In general, we avoid list returns, and return array references instead.

13.9 Use `IO::Scalar` for Files

```
foreach my $params (
  [BodyToFile => $body],
  [HeadAndBodyToFile => qr/\Q$subject\E.*\Q$body/s],
) {
  my($method, $expect) = @$params;
  my($buf) = &#39;&#39;;
  is($pop3->$method(IO::Scalar->new(\$buf), 1), 1);
  _is_match($buf, $expect);
}
```

`BodyToFile` and `HeadAndBodyToFile` accept a file glob to write the message parts. This API design is easily testable with the use of `IO::Scalar`, an in-memory file object. It avoids file naming and disk clean up issues.

We create the `IO::Scalar` instance in `compute_params`, which `Bivio::Test` calls before each method invocation. `check_return` validates that the method returned true, and then calls `actual_return` to set the return value to the contents of the `IO::Scalar` instance. It's convenient to let `Bivio::Test` perform the structural comparison for us.

13.10 Perturb One Parameter per Deviance Case

```
foreach my $method (qw(BodyToFile HeadAndBodyToFile)) {
  is($pop3->$method(IO::Scalar->new(\(&#39;&#39;)), 999), 0);
  my($handle) = IO::File->new(&#39;> /dev/null&#39;);
  $handle->close;
  is($pop3->$method($handle, 1), 0);
}
```

[4] The book *Effective Perl Programming* by Joseph Hall discusses the issues with `wantarray` and list contexts in detail.

nagler@extremeperl.org

We test an invalid message number and a closed file handle[5] in two separate deviance cases. You shouldn't perturb two unrelated parameters in the same deviance case, because you won't know which parameter causes the error.

The second case uses a one-time `compute_params` closure in place of a list of parameters. Idioms like this simplify the programmer's job. Subject matter oriented programs use idioms to eliminate repetitious boilerplate that obscures the subject matter. At the same time, idioms create a barrier to understanding for outsiders. The myriad `Bivio::Test` may seem overwhelming at first. For the test-first programmer, `Bivio::Test` clears away the clutter so you can see the API in action.

13.11 Relate Results When You Need To

```
foreach my $method (qw(Uidl List ListArray)) {
  my($first) = ($pop3->$method())[$method eq &#39;List&#39; ? 0 : 1];
  ok($first);
  is_deeply([$pop3->$method(1)], [$first]);
  is_deeply([$pop3->$method(999)], []);
}
```

`Uidl` (Unique ID List), `List`, and `ListArray` return lists of information about messages. `Uidl` and `ListArray` lists are indexed by message number (starting at one, so the zeroeth element is always `undef`). The values of these lists are the message's unique ID and size, respectively. `List` returns a list of unparsed lines with the zeroeth being the first line. All three methods also accept a single message number as a parameter, and return the corresponding value. There's also a scalar return case which I didn't include for brevity in the book.

The first case retrieves the entire list, and saves the value for the first message. As a sanity check, we make sure the value is non-zero (true). This is all we can guarantee about the value in all three cases.

[5] We use `IO::File` instead of `IO::Scalar`, because `IO::Scalar` does not check if the instance is closed when `Mail::POP3Client` calls print.

The second case requests the value for the first message from the POP3 server, and validates this value agrees with the value saved from the list case. The one-time `check_return` closure defers the evaluation of `$_SAVE` until after the list case sets it.

We cross-validate the results, because the expected values are unpredictable. Unique IDs are server specific, and message sizes include the head, which also is server specific. By relating two results, we are ensuring two different execution paths end in the same result. We assume the implementation is reasonable, and isn't trying to trick the test. These are safe assumptions in XP, since the programmers write both the test and implementation.

13.12 Order Dependencies to Minimize Test Length

```
my($count) = $pop3->Count();
ok($count >= 1);
is($pop3->Delete(1), 1);
is($pop3->Delete(999), 0);
$pop3->Reset;
is($pop3->Close, 1);
$pop3->Connect;
is($pop3->Count, $count);
# Clear mailbox, which also cleans up aborted or bad test runs
foreach my $i (1 .. $count) {
  $pop3->Delete($i);
};
is($pop3->Close, 1);
$pop3->Connect;
is($pop3->Count, 0);
is($pop3->Close, 1);
```

We put the destructive cases (`Delete`) near the end. The prior tests all need a message in the mailbox. If we tested delete first, we'd have to resend a message to test the retrieval and list methods. The case ordering reduces test length and complexity.

Note that we cannot guarantee anything about `Count` except that is at least one. A prior test run may have aborted prematurely and left another

nagler@extremeperl.org

message in the test mailbox. What we do know is that if we `Delete` all messages from one to `Count`, the mailbox should be empty. The second half of this case group tests this behavior.

The empty mailbox case is important to test, too. By deleting all messages and trying to login, we'll see how `Mail::POP3Client` behaves in the this case.

Yet another reason to delete all messages is to reset the mailbox to a known state, so the next test run starts with a clean slate. This self-maintaining property is important for tests that access persistent data. Rerun the entire test twice in a row, and the second run should always be correct.

The POP3 protocol doesn't remove messages when `Delete` is called. The messages are marked for deletion, and the server deletes them on successful `Close`. `Reset` clears any deletion marks. We cross-validate the first `Count` result with the second to verify `Reset` does what it is supposed to do.

13.13 Consistent APIs Ease Testing

```
$pop3 = Mail::POP3Client->new;
is($pop3->State, &#39;DEAD&#39;);
is($pop3->Alive, &#39;&#39;);
is($pop3->Host($cfg->{HOST}), $cfg->{HOST});
is($pop3->Host, $cfg->{HOST});
$pop3->Connect;
is($pop3->Alive, 1);
is($pop3->State, &#39;AUTHORIZATION&#39;);
is($pop3->User($cfg->{USER}), $cfg->{USER});
is($pop3->User, $cfg->{USER});
is($pop3->Pass($cfg->{PASSWORD}), $cfg->{PASSWORD});
is($pop3->Pass, $cfg->{PASSWORD});
is($pop3->Login, 0);
is($pop3->State, &#39;TRANSACTION&#39;);
is($pop3->Alive, 1);
is($pop3->Close, 1);
is($pop3->Alive, &#39;&#39;);
is($pop3->Close, 0);

$pop3 = Mail::POP3Client->new;
$pop3->Connect;
```


nagler@extremeperl.org

```
is($pop3->Alive, &#39;&#39;);
is($pop3->Login, 0);
is($pop3->State, &#39;DEAD&#39;);
```

This section not only tests the accessors, but also documents the `State` and `Alive` transitions after calls to `Connect` and `Login`.

There's a minor design issue to discuss. The accessor `Pass` does not match its corresponding named parameter, `PASSWORD`, like the `Host` and `User` do. The lack of uniformity makes using a `map` function for the accessor tests cumbersome, so we didn't bother.

Also the non-uniform return values between `Alive` and `Close` is clear. While the empty list and zero (0) are both false in Perl, it makes testing for exact results more difficult than it needs to be.

13.14 Inject Failures

```
$pop3 = Mail::POP3Client->new(%$cfg);
is($pop3->POPStat, 0);
$pop3->Socket->close;
is($pop3->POPStat, -1);
is($pop3->Close, 0);
```

The final (tada!) case group injects a failure before a normal operation. `Mail::POP3Client` exports the socket that it uses. This makes failure injection easy, because we simply close the socket before the next call to `POPStat`. Subsequent calls should fail.

We assume error handling is centralized in the implementation, so we don't repeat all the previous tests with injected failures. That's a big assumption, and for `Mail::POP3Client` it isn't true. Rather than adding more cases to this test, we'll revisit the issue of shared error handling in Refactoring.

Failure injection is an important technique to test error handling. It is in a different class from deviance testing, which tests the API. Instead, we use extra-API entry points. It's like coming in through the back door without knockin'. It ain't so polite but it's sometimes necessary. It's also hard to do

nagler@extremeperl.org

if there ain't no backdoor as there is in `Mail::POP3Client`.

13.15 Mock Objects

Mock objects allow you to inject failures and to test alternative execution paths by creating doors where they don't normally exist. `Test::MockObject`[6] allows you to replace subroutines and methods on the fly for any class or package. You can manipulate calls to and return values from these faked entry points.

Here's a simple test that forces CRAM-MD5 authentication:

```
use strict;
use Test::More;
use Test::MockObject;
BEGIN {
  plan(tests => 3);
}
my($socket) = Test::MockObject->new;
$socket->fake_module(&#39;IO::Socket::INET&#39;);
$socket->fake_new(&#39;IO::Socket::INET&#39;);
$socket->set_true(&#39;autoflush&#39;)
  ->set_false(&#39;connected&#39;)
  ->set_series(getline => map({"$_\r\n"}
    # Replace this line with &#39;+OK POP3 <my-apop@secret-key>&#39; for APOP
    &#39;+OK POP3&#39;,
    &#39;+OK Capability list follows:&#39;,
    # Remove this line to disable CRAM-MD5
    &#39;SASL CRAM-MD5 LOGIN&#39;,
    &#39;.&#39;,
    &#39;+ abcd&#39;,
    &#39;+OK Mailbox open&#39;,
    &#39;+OK 33 419&#39;,
  ))->mock(print => sub {
    my(undef, @args) = @_;
    die(&#39;invalid operation: &#39;, @args)
      if grep(/(PASS|APOP)/i, join(&#39;&#39;, @args));
    return 1;
```

[6] Version 0.9 used here is available at: http://search.cpan.org/author/CHROMATIC/Test-MockObject-0.09/

```
  });
use_ok(&#39;Mail::POP3Client&#39;);
my($pop3) = Mail::POP3Client->new(
  HOST => &#39;x&#39;, USER => &#39;x&#39;, PASSWORD => &#39;keep-secret&#39;
);
is($pop3->State, &#39;TRANSACTION&#39;);
is($pop3->Count, 33);
```

In `BEST` authentication mode, `Mail::POP3Client` tries `APOP`, `CRAM-MD5`, and `PASS`. This test makes sure that if the server doesn't support `APOP` that `CRAM-MD5` is used and `PASS` is not used. Most POP3 servers always support `APOP` and `CRAM-MD5` and you usually can't enable one without the other. Since `Mail::POP3Client` always tries `APOP` first, this test allows us to test the `CRAM-MD5` fallback logic without finding a server that conforms to this unique case.

We use the `Test::MockObject` instance to fake the `IO::Socket::INET` class, which `Mail::POP3Client` uses to talk to the server. The faking happens before `Mail::POP3Client` imports the faked module so that the real `IO::Socket::INET` doesn't load.

The first three methods mocked are: `new`, `autoflush`, and `connected`. The mock `new` returns `$socket`, the mock object. We set `autoflush` to always returns true. `connected` is set to return false, so `Mail::POP3Client` doesn't try to close the socket when its `DESTROY` is called.

We fake the return results of `getline` with the server responses `Mail::POP3Client` expects to see when it tries to connect and login. To reduce coupling between the test and implementation, keep the list of mock routines short. You can do this by trial and error, because `Test::MockObject` lets you know when a routine that isn't mocked has been called.

The mock `print` asserts that neither `APOP` nor `PASS` is attempted by `Connect`. By editing the lines as recommend by the comments, you can inject failures to see that the test and `Mail::POP3Client` works.

There's a lot more to `Test::MockObject` than I can present here. It can make a seemingly impossible testing job almost trivial.

nagler@extremeperl.org

13.16 Does It Work?

As noted, several of the `Mail::POP3Client` test cases were successful, that is, they found defects in the implementation. In Refactoring, you'll see the fruits of this chapter's labor. We'll refactor the implementation to make it easier to fix the defects the test uncovered. We'll run the unit test after each refactoring to be sure we didn't break anything.

Chapter 14

Refactoring

> In refactoring, there is a premium on knowing when to quit.
>
> – Kent Beck

Refactoring is an iterative design strategy. Each refactoring improves an implementation in a small way without altering its observable behavior. You refactor to make the implementation easier to repair and to extend. The refactorings themselves add no business value.

Programming in XP is a four part process: listening, testing, coding, and designing. Design comes after you have a test for what the customer wants, and you have a simple implementation already coded. Of course, your simple implementation is based on your definition of simple. Your first cut may even be optimally coded in which the design is complete for the task you are working on. Your next task may require you to copy some similar code from another module, because there's no API to get at it. Copying code is usually the simplest way to see if that code actually works, that is, passes your tests. Copying code is not good design. After you get the copy working, you need to refactor the original module and your new code to use a common API. And, that's emergent design.

The designing emerges from the problem, completely naturally, like you organize a kitchen. The figure out where to put things depending on how frequently they are used. If you only use that fruitcake pan once a year, you probably tuck it away in the back of a hard to reach cabinet. Similarly, if a routine is used only in one place, you keep it private within a module. The first time it is used elsewhere, you may copy it. If you find another use for it, you refactor all three uses so that they call a single copy of the routine.

In XP, we call this the Rule of Three, [1] and basically it says you only know some code is reusable if you copy it two times. Refactoring is the process by which you make the code reusable.

This chapter demonstrates several refactorings in the context of a single example, `Mail::POP3Client`. We discuss how and when to refactor and why refactoring is a good design strategy. We show how refactoring makes it easier to fix a couple of defects found by the unit test in Unit Testing.

14.1 Design on Demand

XP has no explicit design phase. We start coding right away. The design evolves through expansions (adding or exposing function) and contractions (eliminating redundant or unused code).

Refactoring occurs in both phases. During contraction, we clean up excesses of the past, usually caused by copy-and-paste. During expansion, we modularize function to make it accessible for the task at hand. Expansion solves today's problems. Contraction helps mitigate future problems.

14.2 `Mail::POP3Client`

The examples in this chapter use the foundation we laid in Unit Testing. `Mail::POP3Client`[2] is a CPAN class that implements the client side of the POP3 mail retrieval protocol. Since refactoring does not change observable behavior, you don't need to understand how POP3 works to follow the examples in this chapter.

I chose `Mail::POP3Client`, because it is well-written, mature, and solves a real problem. Some of the code is complex. The meat of the example is highlighted in the changed version, so skip ahead if code complexity interferes with comprehension. I have included entire subroutines for completeness, but the differences are just a few lines of code.

14.3 Remove Unused Code

We'll warm up with a very simple refactoring. Here's the `Host` accessor from `Mail::POP3Client`:

[1] Don Roberts came up with this rule, and it is noted in *Refactoring: Improving the Design of Existing Code*, Martin Fowler, Addison Wesley, 1999, p. 58.

[2] http://search.cpan.org/author/SDOWD/POP3Client-2.12

nagler@extremeperl.org

```
sub Host {
  my $me = shift;
  my $host = shift or return $me->{HOST};
#   $me->{INTERNET_ADDR} = inet_aton( $host ) or
#   $me->Message( "Could not inet_aton: $host, $!") and return;
  $me->{HOST} = $host;
}
```

This first refactoring removes the extraneous comment. Unused code and comments distract the reader from the salient bits. Should we need to resurrect the dead code, it remains in a file's history available from the source repository. For now, we don't need it, and the refactored version is shorter and more readable:

```
sub Host {
  my $me = shift;
  my $host = shift or return $me->{HOST};
  $me->{HOST} = $host;
}
```

After each refactoring, even as simple as this one, we run the unit test. In this case, the test will tell us if we deleted too much. Before we check in our changes, we run the entire unit test suite to make sure we didn't break anything.

How often to check in is a matter of preference. The more often you check in, the easier it is to back out an individual change. The cost of checking in is a function of the size of your test suite. Some people like to check in after every refactoring. Others will wait till the task is complete. I wouldn't check in now unless this change was unrelated to the work I'm doing. For example, I might have deleted the dead code while trying to figure out how to use `Mail::POP3Client` in another class.

14.4 Refactor Then Fix

As discussed in Distinguish Error Cases Uniquely, the `Message` method has a defect that doesn't let us clear its value. Its implementation is identical

nagler@extremeperl.org

to `Host` and six other accessors:

```
sub Message {
  my $me = shift;
  my $msg = shift or return $me->{MESG};
  $me->{MESG} = $msg;
}
```

The method tests the wrong condition: `$msg` being false. It should test to see if `$msg` was passed at all: `@_` greater than 1. For most of the other accessors, this isn't a real problem. However, the `Debug` method only allows you to turn on debugging output, since it only saves true values:

```
sub Debug
{
 my $me = shift;
 my $debug = shift or return $me->{DEBUG};
 $me->{DEBUG} = $debug;
}
```

We could fix all the accessors with the global replacement, but the repetition is noisy and distracting. It's better to contract the code to improve its design while fixing the defects.

We refactor first, so we can test the refactoring with the existing unit test. Once we determine the refactoring hasn't changed the observable behavior, we fix the fault. Usually, refactoring first cuts down on the work, because we only need to fix the fault in one place.

The first step is to split out the existing logic into a separate subroutine, and update all accessors to use the new routine:

```
sub Debug {
  return _access(&#39;DEBUG&#39;, @_);
}
sub Host {
  return _access(&#39;HOST&#39;, @_);
}
sub LocalAddr {
  return _access(&#39;LOCALADDR&#39;, @_);
```


nagler@extremeperl.org

```
}
sub Message {
  return _access(&#39;MESG&#39;, @_);
}
sub Pass {
  return _access(&#39;PASSWORD&#39;, @_);
}
sub Port {
  return _access(&#39;PORT&#39;, @_);
}
sub State {
  return _access(&#39;STATE&#39;, @_);
}
sub User {
  return _access(&#39;USER&#39;, @_);
}
sub _access {
  my $field = shift;
  my $me = shift;
  my $value = shift or return $me->{$field};
  $me->{$field} = $value;
}
```

Perl allows us to pass on the actual parameters of a routine simply. Since the `$value` parameter is optional, we put the new parameter `$field` before the rest of the parameters. This simple trick reduces code complexity. By the way, it's not something I'd do if `_access` were public. If `_access` needed to be overriddable by a subclass, it would have been called as a method (`$me->access`). However, the XP rule "you aren't going to need it" applies here. We'll expose `_access` to subclasses when there's an explicit need to.

14.5 Consistent Names Ease Refactoring

The refactored accessors are repetitive. In Perl it's common to generate repetitive code, and we'll do so here.[3] First, we need to rename `MESG` to

[3] There are numerous CPAN classes for generating accessors. The purpose of this refactoring is to demonstrate the technique of generating code to reduced repetition. In the real world, you'd use one of the CPAN classes or, better yet, eschew accessors in favor

MESSAGE to simplify generation. There are only two references in Mail::POP3Client (one use not shown), so this refactoring is easy:

```
sub Message {
  return _access(&#39;MESSAGE&#39;, @_);
}
```

Next, we refactor the Pass accessor. As noted in Consistent APIs Ease Testing, the inconsistent naming (Pass method and PASSWORD configuration parameter) made testing a bit more repetitive than it needed to be. We don't change the parameter and instance field from PASSWORD to PASS, because abbreviations are less desirable. Rather, we refactor Pass as follows:

```
sub Pass {
  return shift->Password(@_);
}
sub Password {
  return _access(&#39;PASSWORD&#39;, @_);
}
```

As a part of this refactoring, we need document that Pass is deprecated (will eventually go away). Mail::POP3Client is available on CPAN, so we can't fix all the code that uses it. Deprecation is an important refactoring technique for very public APIs like this one.

14.6 Generate Repetitive Code

With the refactored names, we can now generate all eight accessors with this simple loop:

```
foreach my $a (qw(
  Debug
  Host
  LocalAddr
  Message
  Password
```

a super-class which implements accessors via get and put methods.

nagler@extremeperl.org

```
  Port
  State
  User
)) {
  eval(qq{
    sub $a {
      return _access(&#39;@{[uc($a)]}&#39;, \@_);
    }
    1;
  }) or die($@);
}
```

We check `eval`'s result to be sure the generator works correctly. The syntax to insert the field name is a bit funky, but I prefer it to using a temporary variable. The `@{[any-expression]}` idiom allows arbitrary Perl expressions to be interpolated in double-quoted strings.

14.7 Fix Once and Only Once

With the refactoring complete, we can now fix all eight accessors with one line of code. `_access` should not check the value of its third argument, but should instead use the argument count:

```
sub _access {
  my $field = shift;
  my $me = shift; @_ or return $me->{$field};
  $me->{$field} = shift;
}
```

This is not a refactoring. It changes the observable behavior. Our unit test confirms this: three more test cases pass.

nagler@extremeperl.org

14.8 Stylin'

Coding style is important to consider when refactoring. In this chapter, I chose to maintain the style of the source.[a] To contrast, here's what `_access` would look like in bOP style:

`sub` $_a ccess$ `my(`$field$`,self,` $value$`) = @;` `return @` $_> = 3?$`self->`$field =$`value :` $self- >$

[a] There are some minor changes to whitespace, indentation, and brace placement for consistency and brevity within this book.

[b] Except in FORTRAN, where the name of the variable can determine its type.

14.9 Tactics Versus Strategy

When we begin a task, tactics are important. We want to add business value as quickly as possible. Doing the simplest thing that could possibly work is a tactic to reach our goal.

Copy-and-paste is the weapon of choice when task completion is our only objective. Generalizations, such as the refactorings shown thus far, are not easy to come up with when you're under pressure to get the job done. Besides, you don't know if the code you copy solves the problem until the implementation is finished. It's much better to copy-and-paste to test an idea than to invent, to implement, and to use the wrong abstraction.

As a design strategy, copy-and-paste poses problems. The code is more difficult to comprehend, because it's difficult to see subtle differences. Faults fixed in one place do not propagate themselves automatically. For example, there's an alternative fix to the problem already embedded in two other accessors, `Count` and `Size`:

```
sub Count {
  my $me = shift;
  my $c = shift;
  if (defined $c and length($c) > 0) {
    $me->{COUNT} = $c;
  } else {
    return $me->{COUNT};
  }
}
```

```
sub Size {
  my $me = shift;
  my $c = shift;
  if (defined $c and length($c) > 0) {
    $me->{SIZE} = $c;
  } else {
    return $me->{SIZE};
  }
}
```

These accessors behave differently from the other eight that we refactored and fixed above. `Count` and `Size` need to be resettable to zero (mailboxes can be empty), which is why the accessors have an alternate implementation.

The thought and debugging that went into fixing `Count` and `Size` could have also applied to the other accessors. Since the code wasn't refactored at the time of the fix, it was probably easier to leave the other accessors alone. And, under the principle of "if it ain't broke, don't fix it" any change like this is not justified.

XP legitimatizes fixing non-broke code. It's something programmers do anyway, so XP gives us some structure and guidelines to do it safely. We can refactor `Size` and `Count` to use `_access` without fear.[4] The unit test covers the empty mailbox case. If we didn't have a test for this case, we could add one. Again, XP saves us. Since the programmers are the testers, we're free to modify the test to suit our needs.

14.10 Refactor With a Partner

Pair programming supports refactoring, too. Two people are better than one at assessing the need for, the side-effects of, and the difficulty of a change. The tradeoffs between tactics versus strategy are hard, and discussing them with a partner is both effective and natural. Switching partners often brings new perspectives to old code, too.

Sometimes I look at code and don't know where to begin refactoring. The complexity overwhelms my ability to identify commonality. For example, here's some code which needs refactoring:

[4] When making changes to CPAN modules, XP, nor any other methodology, helps to validate uses in the (unknown) importers.

```
sub List {
  my $me = shift;
  my $num = shift || &#39;&#39;;
  my $CMD = shift || &#39;LIST&#39;;
  $CMD=~ tr/a-z/A-Z/;
  $me->Alive() or return;
  my @retarray = ();
  my $ret = &#39;&#39;;
  $me->_checkstate(&#39;TRANSACTION&#39;, $CMD) or return;
  $me->_sockprint($CMD, $num ? " $num" : &#39;&#39;, $me->EOL());
  my $line = $me->_sockread();
  unless (defined $line) {
    $me->Message("Socket read failed for LIST");
    return;
  }
  $line =~ /^\+OK/ or $me->Message("$line") and return;
  if ($num) {
    $line =~ s/^\+OK\s*//;
    return $line;
  }
  while (defined($line = $me->_sockread())) {
    $line =~ /^\.\s*$/ and last;
    $ret .= $line;
    chomp $line;
    push(@retarray, $line);
  }
  if ($ret) {
    return wantarray ? @retarray : $ret;
  }
}

sub ListArray {
  my $me = shift;
  my $num = shift || &#39;&#39;;
  my $CMD = shift || &#39;LIST&#39;;
  $CMD=~ tr/a-z/A-Z/;
  $me->Alive() or return;
  my @retarray = ();
  my $ret = &#39;&#39;;
  $me->_checkstate(&#39;TRANSACTION&#39;, $CMD) or return;
```

```
  $me->_sockprint($CMD, $num ? " $num" : &#39;&#39;, $me->EOL());
  my $line = $me->_sockread();
  unless (defined $line) {
    $me->Message("Socket read failed for LIST");
    return;
  }
  $line =~ /^\+OK/ or $me->Message("$line") and return;
  if ($num) {
    $line =~ s/^\+OK\s*//;
    return $line;
  }
  while (defined($line = $me->_sockread())) {
    $line =~ /^\.\s*$/ and last;
    $ret .= $line;
    chomp $line;
    my($num, $uidl) = split &#39; &#39;, $line;
    $retarray[$num] = $uidl;
  }
  if ($ret) {
    return wantarray ? @retarray : $ret;
  }
}

sub Uidl {
  my $me = shift;
  my $num = shift || &#39;&#39;;
  $me->Alive() or return;
  my @retarray = ();
  my $ret = &#39;&#39;;
  $me->_checkstate(&#39;TRANSACTION&#39;, &#39;UIDL&#39;) or return;
  $me->_sockprint(&#39;UIDL&#39;, $num ? " $num" : &#39;&#39;, $me->EOL());
  my $line = $me->_sockread();
  unless (defined $line) {
    $me->Message("Socket read failed for UIDL");
    return;
  }
  $line =~ /^\+OK/ or $me->Message($line) and return;
  if ($num) {
    $line =~ s/^\+OK\s*//;
    return $line;
```


nagler@extremeperl.org

```
  }
  while (defined($line = $me->_sockread())) {
    $line =~ /^\.\s*$/ and last;
    $ret .= $line;
    chomp $line;
    my($num, $uidl) = split &#39; &#39;, $line;
    $retarray[$num] = $uidl;
  }
  if ($ret) {
    return wantarray ? @retarray : $ret;
  }
}
```

Where are the differences? What's the first step? With a fresh perspective, the following stood out:

```
sub Uidl {
  my $me = shift;
  my $num = shift;
  return $me->ListArray($num, &#39;UIDL&#39;);
}
```

A partner helps you overcome familiarity and fear of change which make it hard to see simplifications like this one.

14.11 Sharing with Code References

It's clear that `List` and `ListArray` are almost identical. The problem is that they differ in the middle of the loop. Perl code references are a great way to factor out differences especially within loops:

```
sub List { return _list( sub { my $line = shift; my $retarray =
shift; push(@$retarray, $line); return; }, @_, ); } sub ListArray
{ return _list( sub { my($num, $value) = split &#39; &#39;, shift;
my $retarray = shift; $retarray->[$num] = $value; return; }, @_,
); } sub _list { my $parse_line = shift;
  my $me = shift;
```


nagler@extremeperl.org

```
  my $num = shift || &#39;&#39;;
  my $CMD = shift || &#39;LIST&#39;;
  $CMD =~ tr/a-z/A-Z/;
  $me->Alive() or return;
  my @retarray = ();
  my $ret = &#39;&#39;;
  $me->_checkstate(&#39;TRANSACTION&#39;, $CMD) or return;
  $me->_sockprint($CMD, $num ? " $num" : &#39;&#39;, $me->EOL());
  my $line = $me->_sockread();
  unless (defined $line) {
    $me->Message("Socket read failed for $CMD");
    return;
  }
  $line =~ /^\+OK/ or $me->Message("$line") and return;
  if ($num) {
    $line =~ s/^\+OK\s*//;
    return $line;
  }
  while (defined($line = $me->_sockread())) {
    $line =~ /^\.\s*$/ and last;
    $ret .= $line;
    chomp $line; $parse_line->($line, \@retarray);
  }
  if ($ret) {
    return wantarray ? @retarray : $ret;
  }
}
```

We pass an anonymous subroutine as `_list`'s first parameter, `$parse_line`, for the reason described in Refactor Then Fix.

14.12 Refactoring Illuminates Fixes

The `List`, `ListArray`, and `Uidl` methods are bimodal. When called without arguments, they return a list of values. When passed a message number, the value for that message number alone is returned. The message number cases failed in our unit test.

The code reference refactoring shows us where the fault lies: `$parse_line`

is not called when _list is called with an argument. It also needs to chomp the $line to match the behavior:

```
  if ($num) {
    $line =~ s/^\+OK\s*//; chomp $line; return $parse_line->($line);
  }
```

The anonymous subroutines in List and ListArray need to be bimodal for this refactoring to work:

```
sub List {
  return _list(
    sub {
      my $line = shift;
      my $retarray = shift or return $line;
      push(@$retarray, $line);
      return;
    },
    @_,
  );
}

sub ListArray {
  return _list(
    sub {
      my($num, $value) = split &#39; &#39;, shift; my $retarray
= shift or return $value;
      $retarray->[$num] = $value;
      return;
    },
    @_,
  );
}
```

By compressing the business logic, its essence and errors become apparent. Less code is almost always better than more code.

While advanced constructs like code references may be difficult to understand for those unfamiliar with them, dumbing down the code is not a

nagler@extremeperl.org

good option. Defaulting to the least common denominator produces dumb code and ignorant programmers. In XP, change occurs not only the project artifacts but also in ourselves. Learning and teaching are an integral part of the XP methodology.

14.13 Brush and Floss Regularly

This chapter presents a glimpse of design on demand. Each refactoring was implemented and tested separately. The trick to refactoring successfully is taking baby steps.

I like to compare refactoring to brushing your teeth. Your best shot at preventing tooth decay is to brush briefly after each meal and to floss daily. Alternatively, you could just wait for cavities to appear and have them filled. The short term cost in pain, money, and time is much greater if you do. In the long term, too many fillings create structural problems and the tooth has to be replaced.

Refactoring is preventative maintenance, like tooth brushing. Quick fixes are like fillings. Eventually, they create structural problems and require the implementation to be replaced. Like teeth, complete rewrites are unnecessarily painful and costly.

XP encourages you to do the simplest thing that could possibly work (tactics) to address the immediate problem . You refactor your code to express concepts once and only once (strategy) to prevent structural problems. And, don't forget that brushing and flossing support pair programming.

Chapter 15

It's a SMOP

> Representation *is* the essence of programming.
>
> – Fred Brooks

Implementing Extreme Perl is a simple matter of programming. Practicing XP clarifies its values. Perl's virtues come alive as you read and write it. The subject matter language crystallizes as the subject matter oriented program evolves along with the programmers' mastery of the problem domain.

This chapter coalesces the book's themes (XP, Perl, and SMOP) into a single example: a DocBook XML to HTML translator. We walk through the planning game, split the story into tasks, discuss coding style, design simply, pair program, test first, and iterate. The subject matter oriented program (SMOP) is a hash of XML to HTML tags interpreted by a declarative algorithm. Along the way, declarative programming is defined and related to other paradigms (object-oriented and imperative programming).

15.1 The Problem

The example in this chapter converts DocBook XML[1] (the source form of this book) to HTML. The example went beyond this chapter, and the version here was enhanced to produce the review copies for the entire book.[2]

DocBook is a technical document description language. I described the paragraphs, sections, terms, etc. of this book with tags as follows:

[1] *DocBook: The Definitive Guide* by Norman Walsh and Leonard Muellner, available online at http://www.docbook.org/tdg/en/html/docbook.html

[2] The most recent version is available with bOP.

```
<blockquote><para>
To learn about <firstterm>XML</firstterm>, visit
<systemitem role="url">http://www.w3c.org/XML</systemitem>.
</para></blockquote>
```

It's unreadable in source form, and my resident reviewer in chief pointed this out after I asked her to review the first chapter in source form. She was expecting something like this:

> To learn about XML, visit http://www.w3c.org/XML.

Since eating your own dog food is a great way to make sure it's palatable, my resident reviewer in chief agreed to act as the on-site customer[3] for a DocBook to HTML translator.

I am happy to say she is satisfied with the output of the program.[4]

15.2 Planning Game

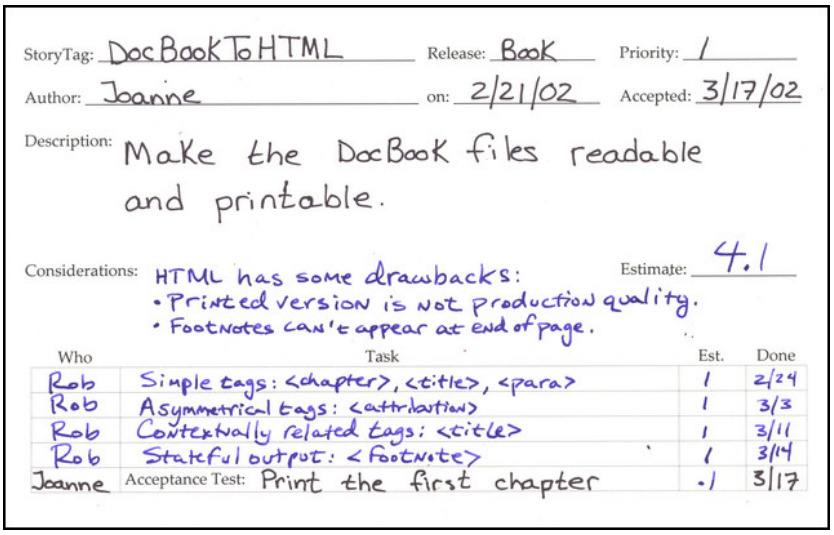
The planning game was brief. There was only one story card (shown completed):

The Story Card

[3] To be sure, I added it to her wedding vows *ex post facto*.

[4] One reviewer would have preferred Perl POD. However, XP only works when the customer speaks in one voice, so I ignored him for the sake of matrimonial harmony.

nagler@extremeperl.org



StoryTag: DocBookToHTML Release: Book Priority: 1

Author: Joanne on: 2/21/02 Accepted: 3/17/02

Description: Make the DocBook files readable and printable.

Considerations: HTML has some drawbacks: Estimate: 4.1
- Printed version is not production quality.
- Footnotes can't appear at end of page.

Who	Task	Est.	Done
Rob	Simple tags: <chapter>, <title>, <para>	1	2/24
Rob	Asymmetrical tags: <attribution>	1	3/3
Rob	Contextually related tags: <title>	1	3/11
Rob	Stateful output: <footnote>	1	3/14
Joanne	Acceptance Test: Print the first chapter	.1	3/17

Note the simplicity of the story. One sentence is usually sufficient to describe the problem.

During the planning game, we decided almost immediately the output would be HTML. We briefly discussed simply stripping the XML tags to produce a plain text document. We dropped this idea quickly, because the output would not be very easy to read, for example, footnotes would appear in the middle of the text. We touched on alternative formatting languages, such as, Rich Text Format (RTF), but the simplicity of HTML and its relatedness to XML made the decision for us. HTML provides enough formatting to give a sense of the layout, which is all Joanne needed to read and print the chapters.

15.3 Dividing the Story into Tasks

We already had a chapter as a sample input. This made it easy to define the tasks. I needed the tasks to be bite-sized chunks, because my programming partners were only available for brief periods.

The task split was simple. I scanned the chapter looking for problematic tags. The first task specifies simple tags. The other tasks specify one problematic tag each. Only DocBook tags used in the chapter were included, and each tag can be found in at least one test case.

15.4 Coding Style

The story card does not mention declarative programming. It also doesn't specify what language, operating system, or hardware is to be used. The customer simply wants readable and printable chapters. She doesn't care how we do it.

Too often we begin coding without an explicit discussion about how we're going to code, that is. what language and what style. For this project, we chose Perl for the following reasons:

- XML maps naturally to Perl's built-in data structures (lists and hashes),
- CPAN has several ready-to-use XML parsers,
- It's easy to generate HTML in Perl, and
- I needed an example for this book.

The last reason is important to list. One of the core values of XP is communication. By listing my personal motivation, I'm being honest with my team. Miscommunication often comes from hidden agendas.

15.5 Simple Design

The problem lends itself to simplicity. XML and HTML are declarative languages, and an important property of declarative languages is ease of manipulation. For example, consider the following DocBook snippet:

```
<para>
<classname>XML::Parser</classname> parses XML into a tree.
</para>
```

The relationships are clear, and the mapping to HML is simply:

```
<p>
<tt>XML::Parser</tt> parses XML into a tree.
</p>
```

One could translate the tags with a simple tag for tag mapping, such as:

nagler@extremeperl.org

```
s{<(/?)para>}{<$1p>}g;
s{<(/?)classname>}{<$1tt>}g;
```

This design is too simple, however. It assumes the XML is well-formed, which it often isn't when I write it. For example, if I were to leave off `</classname>`, the closing `</tt>` would be missing in the HTML, and the output would be in `tt` font for the rest of the document. The classic response to this is: garbage in, garbage out. However, we did better without added complexity[5], and the solution evolved with minimal changes.

We favored hubris and impatience over doing the simplest thing that could possibly work. A little chutzpah is not bad every now and then as long as you have benchmarks to measure your progress. If the implementation size grew too rapidly, we would have backed off to the simpler solution. If we blew our task estimates, we'd have to ask if we didn't under-estimate the complexity of the more radical approach.

The design we chose starts with the output of the CPAN package, `XML::Parser`. If the XML is not well-formed, `XML::Parser` dies. There is no output when the input is garbage.

`XML::Parser` preserves the semantic structure of the input. The translation is an in-order tree traversal, so the output is likely to be well-formed HTML, which also is a tree.

15.6 Imperative versus Declarative

To help understand the benefits of declarative languages, let's consider an alternate problem. Suppose I was writing this book in `troff`[6], an imperative text formatting language:

```
.PP
\fBXML::Parser\fR parses XML into a tree.
```

The commands in `troff` are not relational. The `.PP` does not bracket the

[5] One of the reviewers implemented the simple approach, and the two solutions are of comparable size and complexity.

[6] `troff` is a text-formatting language for UNIX man pages and other documents. After 25 years, it's still in use. For more information, visit http://www.kohala.com/start/troff/troff.html.

paragraph it introduces. `troff` interprets the `.PP` as a paragraph break, and what follows need not be a paragraph at all. The command is imperative, because it means do something right now irrespective of context.

The `\fB` and `\fR` commands do not relate to the value they surround, either. `\fB` turns on bold output, and `\fR` turns off all emphasis statefully. Drop one or the other, and the `troff` is still well-formed. `troff` commands are unrelated to one another except by the order in which they appear in the text.

Writing a `troff` parser is straightforward. The complete grammar is not much more complicated than the example above. Translating `troff` to HTML is much more difficult.[7] For example, consider the following `troff`:

```
\fBXML::Parser\fI is sweet!\fR
```

The equivalent HTML is:

```
<b>XML::Parser</b><i> is sweet!</i>
```

A simple command-to-tag translation is insufficient. The program must maintain the state of the font in use, and output the corresponding closing tag (`</b>`) when the font changes before appending the font tag (`<i>`). The same is true for font sizes, indentation level, line spacing, and other stateful `troff` commands. The program has to do two jobs: map commands to tags and emulate the state management of `troff`.

As you'll see, the XML to HTML translator does not maintain global input state to perform its job. The XML tags are translated based on minimal local context only. The only relational information required is the parent of the current tag. The mapping is stateless and therefore simpler due to XML's declarativeness.

[7] Eric Raymond's `doclifter` performs an even more herculean task: the program converts `troff` to DocBook. `doclifter` uses the implicit relations of common usage patterns to extract higher-level semantics, such as, knowing that man page references usually match the regular expression: `/\w+\(\d+\)/`. The 6,000 line program is written declaratively in Python, and can be downloaded from http://www.tuxedo.org/~esr/doclifter.

15.7 Pair Programming

Programming courses rarely mention declarative programming. Imperative programming is the norm. It is all too easy to use imperative forms out of habit or as a quick fix, especially when working alone under pressure. You may need to refactor several times to find appropriate declarative forms.

Dogmatic pursuit of declarative forms is not an end in itself, however. Sometimes it's downright counter-productive. Since Perl allows us to program in multiple paradigms, it is tricky to choose how or when to program using objects, imperatively, and declaratively.

For these reasons, it's helpful to program in pairs when coding declarative constructs. It takes time to learn how to code declaratively, just like it takes time to test-first, code simply, and refactor. The learning process is accelerated when you program in pairs.

All tasks and tests in this chapter were implemented in pairs. I would like to thank Alex Viggio for partnering with me on the last three tasks and Justin Schell for helping with the first. Thanks to Stas Bekman for being my virtual partner in the final refactoring-only iteration.

15.8 Test First, By Intention

The first task involves simple tags only. This allowed us to address the basic problem: mapping XML tags to HTML tags. Each XML tag in the first test case maps to zero, one, or two HTML tags.

The first test case input is the trivial DocBook file:

```
<chapter>
<title>Simple Chapter</title>
<simplesect>
<para>Some Text</para>
</simplesect>
</chapter>
```

Here's the expected result:

```
<html><body>
<h1>Simple Chapter</h1>
```


nagler@extremeperl.org

```
<p>Some Text</p>

</body></html>
```

The test case input and expected result are stored in two files named `01.xml` and `01.html`, respectively. In my experience, it pays to put the test cases in a separate subdirectory (`DocBook`), and to check the test cases into the collective repository. If the program runs amok and overwrites these files, you can always retrieve them from the repository. Also, storing all the test data and programs in the repository ensures programmer workstations are stateless. This allows you to switch tasks and/or workplaces easily.

The unit test program is:

```
#!perl -w
use strict;
use Bivio::Test;
use Bivio::IO::File;
Bivio::Test->new(&#39;Bivio::XML::DocBook&#39;)->unit([
  &#39;Bivio::XML::DocBook&#39; => [
    to_html => [
      [&#39;DocBook/01.xml&#39;] => [Bivio::IO::File->read(&#39;DocBook/01.html&#
    ],
  ],
]);
```

The function `to_html` takes a file name as an argument and returns a string reference, which simplifies testing. There is no need to create a temporary output file nor delete it when the test is over. A testable design is a natural outcome of test-first programming.

`Bivio::IO::File->read` returns the contents of the file name passed to it. The output is a scalar reference. If the file does not exist or an I/O error occurs, the function dies.

15.9 Statelessness

`Bivio::IO::File->read` is stateless, or idempotent. It always does the same thing given the same arguments. This isn't to say the file that it reads is stateless. Files and I/O are stateful. Rather, the operation retains no

state itself. If the underlying file does not change, exactly the same data is returned after each call.

Many of Perl's I/O functions are stateful. For example, Perl's `read` returns different values when called with the same arguments, because it maintains an internal buffer and file pointer. Each call returns the current buffer contents and advances an internal buffer index, possibly filling the buffer if it is empty. If the underlying file changes between calls, the old buffered contents are returned regardless. `read` buffers the file (uses state) to improve performance (decrease time), and we pay a price: the data read may not match a valid file state (old and new states may be mixed).

`Bivio::IO::File->read` cannot be used in all cases. Sometimes the file is too large to fit in memory, or the file may be a device that needs to be read/written alternately in a conversational mode. For our test program, `Bivio::IO::File->read` meets our needs, and the declarative operation simplifies the code and ensures data integrity.[8]

In terms of XP, stateless programming supports unit testing. It is easier to test stateless than stateful operations. Internal state changes, such as caching results in buffers, multiply the inputs and outputs implicitly. It's harder to keep track of what you are testing. Stateful test cases must take ordering into account, and tests of implicit state changes make unit tests harder to read.

Stateless APIs can be tested independently. You only need to consider the explicit inputs and outputs for each case, and the cases can be written in any order. The tests are easier to read and maintain.

15.10 `XML::Parser`

Before we dive into the implementation, we need to understand the output of `XML::Parser`. It parses the XML into a tree that constrains our implementation choices. Given `01.xml`, the following data structure is returned by `parsefile`:

```
[
  chapter => [
    {},
    0 => "\n",
```

[8] For purists, the implementation of `Bivio::IO::File->read` does not lock the file, although it could. But `read` can't, because it is defined imperatively, and it cannot assume the file can be read into the buffer in its entirety.

```
    title => [
      {},
      0 => &#39;Simple Chapter&#39;,
    ],
    0 => "\n",
    simplesect => [
      {},
      0 => "\n",
      para => [
        {},
        0 => &#39;Some Text&#39;,
      ],
      0 => "\n",
    ],
    0 => "\n",
  ],
];
```

The tree structure is realized by nesting arrays. The first element in the array is a hash of attribute tags, which we can ignore for this first implementation, because `01.xml` doesn't use XML attributes. The special tag `0` indicates raw text, that is, the literal strings bracketed by the XML tags.

15.11 First SMOP

The implementation of the first task begins with the map from DocBook to HTML, which is the subject matter oriented program (SMOP):

```
my($_TO_HTML) = {
  chapter => [&#39;html&#39;, &#39;body&#39;],
  para => [&#39;p&#39;],
  simplesect => [],
  title => [&#39;h1&#39;],
};
```

The subject matter (tags and their relationships) is expressed succinctly

nagler@extremeperl.org

without much syntactic clutter.Perl's simple quoting comma (=>) describes the program's intention to replace the tag on the left with the list of tags on the right. As an exercise, try to translate this SMOP to another programming language. You'll find that Perl's expressiveness is hard to beat.

15.12 First Interpreter

The SMOP above is known as descriptive declarativeness, just like HTML and XML. The primary advantage of descriptive languages is that they are easy to evaluate. The first interpreter is therefore quite short[9]:

```
sub to_html {
  my($self, $xml_file) = @_;
  return _to_html(XML::Parser->new(Style => &#39;Tree&#39;)->parsefile($xml_file));
}

sub _to_html {
  my($tree) = @_;
  my($res) = &#39;&#39;;
  $res .= _to_html_node(splice(@$tree, 0, 2))
    while @$tree;
  return \$res;
}

sub _to_html_node {
  my($tag, $tree) = @_;
  return HTML::Entities::encode($tree)
    unless $tag;
  die($tag, &#39;: unhandled tag&#39;)
    unless $_TO_HTML->{$tag};
  # We ignore the attributes for now.
  shift(@$tree);
  return _to_html_tags($_TO_HTML->{$tag}, &#39;&#39;)
    . ${_to_html($tree)}
    . _to_html_tags([reverse(@{$_TO_HTML->{$tag}})], &#39;/&#39;);
}
```

[9] For brevity, I've excluded Perl boilerplate, such as `package` and `use` statements. The final version is listed in full regalia including header comments for the subroutines.

```
sub _to_html_tags {
  my($names, $prefix) = @_;
  return join(&#39;&#39;, map({"<$prefix$_>"} @$names));
}
```

The execution is driven by the XML, not our SMOP. `to_html` starts the recursive (in order tree traversal) algorithm by calling `parsefile` with the appropriate arguments. The XML tree it returns is passed to `_to_html`. The tags are translated by the SMOP as they are encountered by `_to_html_node`, the workhorse of the program. The tag names in the SMOP are converted to HTML tags (surrounded by angle brackets, `<>`) by `_to_html_tags`.

15.13 Functional Programming

The subroutine `_to_html_tags` is a pure function, also known as a function without side effects. A pure function is a declarative operation, which is defined formally as follows [10]:

> A declarative operation is *independent* (does not depend on any execution state outside itself), *stateless* (has no internal execution state that is remembered between calls), and *deterministic* (always gives the same results when given the same arguments).

`_to_html_tags` only depends on its inputs. Its output (the HTML tags) is the only state change to the program. And, we expect it to do exactly the same thing every time we call it.

Functional programming is the branch of declarative programming that uses pure functions exclusively. One of Perl's strengths is its functional programming support. For example, `map` and `join` allowed us to implement `_to_html_tags` functionally.

Other Perl operations, such as `foreach`, support imperative programming only. Code that uses `foreach` must rely on stateful side-effects for its outputs. To illustrate this point, let's look at `_to_html_tags` implemented with `foreach`:

```
sub _to_html_tags_imperatively {
```

[10] From *Concepts, Techniques, and Models of Computer Programming* by Peter Van Roy and Seif Haridi, draft version dated January 6, 2003, p. 109, available at http://www.info.ucl.ac.be/people/PVR/book.pdf

```
  my($names, $prefix) = @_;
  my($res) = &#39;&#39;;
  foreach my $name (@$names) {
    $res .= "<$prefix$name>";
  }
  return $res;
}
```

The `foreach` does its job of iterating through `$names`, and nothing more. It abdicates any responsibility for achieving a result. The surrounding code must introduce a variable (`$res`) to extract the output from inside the loop. The variable adds complexity that is unnecessary in the functional version.

15.14 Outside In

Like most programmers, I was trained to think imperatively. It's hard to think declaratively after years of programming languages like C and Java. For example, `_to_html` in our first interpreter uses the imperative `while`, because a functional version didn't spring to mind. This was the simplest thing that could possibly work. It was Stas who suggested the functional refactoring in Final Implementation.

Functional programming requires a paradigm shift from traditional imperative thinking. `_to_html_tags_imperatively` concatenates its result on the *inside* of the `foreach`. The functional `_to_html_tags` concatenates the result on the *outside* of the `map`. Functional programming is like turning an imperative program inside out.[11] Or, as some of my co-workers have noted, it's like programming while standing on your head.

15.15 May I, Please?

The inside out analogy helps us refactor. We can use it to simplify imperative programs. To program functionally from the outset, a different analogy may help: think in terms of requests, not demands. Paul Graham states this eloquently, "A functional program tells you what it wants; an imperative

[11] In *On Lisp*, Paul Graham explains and demonstrates this inside-out concept (page 34). The book is out of print, but you can download it from http://www.paulgraham.com/onlisp.html.

program tells you what to do."[12]

When we apply this analogy to the example, we see that `_to_html_tags_imperatively` tells us it formats tag names one at a time, and it appends them to the end of a string. When its done with that, it'll return the result.

The functional `_to_html_tags` has a list of tag names and wants a string to return, so it asks `join`, a function that conatenates a list into a string. `join` asks for a separator and a list of tags to concatenate. `map` wants to format tag names, so it asks for a formatter ({`"<$prefix$_>"`}) and a list of tag names.

All we're missing is some polite phrases like please and may I, and we can expand this analogy to familial relationships. Imperative kids tell their parents, "I'm taking the car." Declarative kids politely ask, "May I borrow the car, please?". By communicating their desires instead of demands, declarative kids give their parents more leeway to implement their requests.

Pure functions, and declarative programs in general, are more flexible than their imperative cousins. Instead of demanding a calling order that is implicitly glued together with state (variables), declarative programs define relationships syntactically. This reduces the problem of refactoring from an implicit global problem of maintaining state transitions to an explicit local one of preserving syntactic relationships. Functional programs are easier to refactor.

15.16 Second Task

The second task introduces asymmetric output. The test case input file (`02.html`) is:

```
<chapter>
<title>Chapter with Epigraph</title>
<epigraph>
<para>
Representation <emphasis>is</emphasis> the essence of programming.
</para> <attribution>Fred Brooks</attribution>
</epigraph>
<simplesect>
<para>Some Text</para>
</simplesect>
</chapter>
```

[12] *On Lisp*, Paul Graham, p. 33.

nagler@extremeperl.org

The output file (`02.xml`) we expect is:

```
<html><body>
<h1>Chapter with Epigraph</h1>

<p>
Representation <b>is</b> the essence of programming.
</p> <div align=right>-- Fred Brooks</div>


<p>Some Text</p>

</body></html>
```

The XML `attribute` tag doesn't map to a simple HTML `div` tag, so the existing SMOP language didn't work. But first we had to update the unit test.

15.17 Unit Test Maintenance

To add the new case to the unit test, we copied the line containing the first test case, and changed the the filenames:

```
#!perl -w
use strict;
use Bivio::Test;
use Bivio::IO::File;
Bivio::Test->new(&#39;Bivio::XML::DocBook&#39;)->unit([
  &#39;Bivio::XML::DocBook&#39; => [
    to_html => [
      [&#39;DocBook/01.xml&#39;] => [Bivio::IO::File->read(&#39;DocBook/01.html&#39;)],
      [&#39;DocBook/02.xml&#39;] => [Bivio::IO::File->read(&#39;DocBook/02.html&#39;)],
    ],
  ],
]);
```


nagler@extremeperl.org

Woops! We fell into the dreaded copy-and-paste trap. The new line is identical to the old except for two characters out of 65. That's too much redundancy (97% fat and 3% meat). It's hard to tell the difference between the two lines, and as we add more tests it will be even harder. This makes it easy to forget to add a test, or we might copy-and-paste a line and forget to change it.

We factored out the common code to reduce redundancy:

```
#!perl -w
use strict;
use Bivio::Test;
use Bivio::IO::File;
Bivio::Test->new(&#39;Bivio::XML::DocBook&#39;)->unit([
  &#39;Bivio::XML::DocBook&#39; => [
    to_html => [ map({ my($html) = $_; $html =~ s/xml$/html/; [$_]
=> [Bivio::IO::File->read($html)]; } sort(<DocBook/*.xml>))
    ],
  ],
]);
```

This version of the unit test is maintenance free. The test converts all `.xml` files in the `DocBook` subdirectory. All we need to do is *declare* them, i.e., create the `.xml` and `.html` files. We can execute the cases in any order, so we chose to sort them to ease test case identification.

15.18 Second SMOP

We extended the SMOP grammar to accommodate asymmetric output. The new mappings are shown below:

```
my($_TO_HTML) = _to_html_compile({ attribution => { prefix => &#39;<div
align=right>-- &#39;, suffix => &#39;</div>&#39;, },
  chapter => [&#39;html&#39;, &#39;body&#39;], emphasis => [&#39;b&#39;],
epigraph => [],
  para => [&#39;p&#39;],
  simplesect => [],
```


nagler@extremeperl.org

```
  title => [&#39;h1&#39;],
});
```

`attribution` maps to a hash that defines the prefix and suffix. For the other tags, the prefix and suffix is computed from a simple name. We added `_to_html_compile` which is called once at initialization to convert the simple tag mappings (arrays) into the more general prefix/suffix form (hashes) for efficiency.

15.19 Second SMOP Interpreter

We extended `_to_html_node` to handle asymmetric prefixes and suffixes. The relevant bits of code are:

```
 sub _to_html_compile { my($config) = @_; while (my($xml, $html)
= each(%$config)) { $config->{$xml} = { prefix => _to_html_tags($html,
&#39;&#39;), suffix => _to_html_tags([reverse(@$html)], &#39;/&#39;),
} if ref($html) eq &#39;ARRAY&#39;; } return $config; }

sub _to_html_node {
  my($tag, $tree) = @_;
  return HTML::Entities::encode($tree)
    unless $tag;
  die($tag, &#39;: unhandled tag&#39;)
    unless $_TO_HTML->{$tag};
  # We ignore the attributes for now.
  shift(@$tree);
  return $_TO_HTML->{$tag}->{prefix} . ${_to_html($tree)} . $_TO_HTML->{$tag}->{suffix};
}
```

`_to_html_compile` makes `_to_html_node` simpler and more efficient, because it no longer calls `_to_html_tags` with the ordered and reversed HTML tag name lists. Well, I thought it was more efficient. After performance testing, the version in Final Implementation turned out to be just as fast.[13]

The unnecessary compilation step adds complexity without improving

[13] Thanks to Greg Compestine for asking the questions: What are the alternatives, and how do you know is faster?

performance. We added it at my insistence. I remember saying to Alex, "We might as well add the compilation step now, since we'll need it later anyway." Yikes! Bad programmer! Write "I'm not going to need it" one hundred times in your PDA. Even in pairs, it's hard to avoid the evils of pre-optimization.

15.20 Spike Solutions

As long as I am playing true confessions, I might as well note that I implemented a spike solution to this problem before involving my programming partners. A spike solution in XP is a prototype that you intend to throw away. I wrote a spike to see how easy it was to translate DocBook to HTML. Some of my partners knew about it, but none of them saw it.

The spike solution affected my judgement. It had a compilation step, too. Programming alone led to the pre-optimization. I was too confident that it was necessary when pairing with Alex.

Spike solutions are useful, despite my experience in this case. You use them to shore up confidence in estimates and feasibility of a story. You write a story card for the spike, which estimates the cost to research possibilities. Spike solutions reduce risk through exploratory programming.

15.21 Third Task

The third task introduces contextually related XML tags. The DocBook `title` tag is interpreted differently depending on its enclosing tag. The test case input file (`03.xml`) is:

```
<chapter> <title>Chapter with Section Title</title>
<simplesect>
<programlisting>
print(values(%{{1..8}}));
</programlisting>

<para>
Some other tags:
<literal>literal value</literal>,
<function>function_name</function>, and
<command>command-name</command>.
</para>
```


nagler@extremeperl.org

```
<blockquote><para>
A quoted paragraph.
</para></blockquote>
</simplesect>

<sect1> <title>Statelessness Is Next to Godliness</title>
<para>
A new section.
</para>
</sect1>
</chapter>
```

The expected output file (`03.html`) is:

```
<html><body> <h1>Chapter with Section Title</h1>

<blockquote><pre>
print(values(%{{1..8}}));
</pre></blockquote>

<p>
Some other tags:
<tt>literal value</tt>,
<tt>function_name</tt>, and
<tt>command-name</tt>.
</p>

<blockquote><p>
A quoted paragraph.
</p></blockquote>


 <h2>Statelessness Is Next to Godliness</h2>
<p>
A new section.
</p>

</body></html>
```


nagler@extremeperl.org

The `chapter title` translates to an HTML `h1` tag. The `section title` translates to an `h2` tag. We extended our SMOP language to handle these two contextually different renderings of `title`.

15.22 Third SMOP

We discussed a number of ways to declare the contextual relationships in our SMOP. We could have added a `parent` attribute to the hashes (on the right) or nested `title` within a hash pointed to by the `chapter` tag. The syntax we settled on is similar to the one used by XSLT.[14] The XML tag names can be prefixed with a parent tag name, for example, `"chapter/title"`. The SMOP became:

```
my($_XML_TO_HTML_PROGRAM) = _compile_program({
  attribution => {
    prefix => &#39;<div align=right>-- &#39;,
    suffix => &#39;</div>&#39;,
  }, blockquote => [&#39;blockquote&#39;], &#39;chapter/title&#39;
=> [&#39;h1&#39;],
  chapter => [&#39;html&#39;, &#39;body&#39;], command => [&#39;tt&#39;],
  emphasis => [&#39;b&#39;],
  epigraph => [], function => [&#39;tt&#39;], literal => [&#39;tt&#39;],
  para => [&#39;p&#39;], programlisting => [&#39;blockquote&#39;,
&#39;pre&#39;], sect1 => [], &#39;sect1/title&#39; => [&#39;h2&#39;],
  simplesect => [],
});
```

15.23 Third SMOP Interpreter

We refactored the code a bit to encapsulate the contextual lookup in its own subroutine:

[14] The XML Stylesheet Language Translation is an XML programming language for translating XML into XML and other output formats (e.g., PDF and HTML). For more info, see http://www.w3.org/Style/XSL/

nagler@extremeperl.org

```
sub to_html {
  my($self, $xml_file) = @_;
  return _to_html( &#39;&#39;,
    XML::Parser->new(Style => &#39;Tree&#39;)->parsefile($xml_file));
}

sub _eval_child {
  my($tag, $children,  $parent_tag) = @_;
  return HTML::Entities::encode($children)
    unless $tag;
  # We ignore the attributes for now.
  shift(@$children);
  return _eval_op( _lookup_op($tag, $parent_tag), _to_html($tag, $children));
}

 sub _eval_op { my($op, $html) = @_; return $op->{prefix} . $$html
. $op->{suffix}; } sub _lookup_op { my($tag, $parent_tag) = @_; return
$_XML_TO_HTML_PROGRAM->{"$parent_tag/$tag"} || $_XML_TO_HTML_PROGRAM->{$tag}
|| die("$parent_tag/$tag: unhandled tag"); }

sub _to_html {
  my($tag, $children) = @_;
  my($res) = &#39;&#39;;
  $res .= _eval_child(splice(@$children, 0, 2), $tag)
    while @$children;
  return \$res;
}
 # Renamed _compile_program and _compile_tags_to_html not shown for
brevity.
```

The algorithmic change is centralized in `_lookup_op`, which wants a tag and its parent to find the correct relation in the SMOP. Precedence is given to contextually related tags (`"$parent_tag/$tag"`) over simple XML tags (`$tag`). Note that the root tag in `to_html` is the empty string (`''`). We defined it to avoid complexity in the lower layers. `_lookup_op` need not be specially coded to handle the empty parent tag case.

15.24 The Metaphor

This task implementation includes several name changes. Alex didn't feel the former names were descriptive enough, and they lacked coherency. To help think up good names, Alex suggested that our program was similar to a compiler, because it translates a high-level language (DocBook) to a low-level language (HTML).

We refactored the names to reflect this new metaphor. `$_TO_HML` became `$_XML_TO_HTML_PROGRAM`, and `_to_html_compile` to `_compile_program`. and so on. An `$op` is the implementation of an operator, and `_lookup_op` parallels a compiler's symbol table lookup. `_eval_child` evokes a compiler's recursive descent algorithm.

The compiler metaphor helped guide our new name choices. In an XP project, the metaphor subsitutes for an architectural overview document. Continuous design means that the architecture evolves with each iteration, sometimes dramatically, but a project still needs to be coherent. The metaphor brings consistency without straitjacketing the implementation. In my opinion, you don't need a metaphor at the start of a project. Too little is known about the code or the problem. As the code base grows, the metaphor may present itself naturally as it did here.

15.25 Fourth Task

The fourth and final task introduces state to generate the HTML for DocBook `footnotes`. The test case input file (`04.xml`) is:

```
<chapter>
<title>Chapter with Footnotes</title>
<simplesect>

<para>
Needs further clarification. <footnote><para> Should appear at the
end of the chapter. </para></footnote>
</para>

<itemizedlist>
<listitem><para>
First item
</para></listitem>
```


nagler@extremeperl.org

```
<listitem><para>
Second item
</para></listitem>
</itemizedlist>

<para>
Something about XML. <footnote><para> Click here <systemitem role="url">http://www.w3c.org
</para></footnote>
</para>

<para>
<classname>SomeClass</classname>
<varname>$some_var</varname>
<property>a_property</property>
<filename>my/file/name.PL</filename>
<citetitle>War &amp; Peace</citetitle>
<quote>I do declare!</quote>
</para>

</simplesect>
</chapter>
```

The expected output file (`04.html`) is:

```
<html><body>
<h1>Chapter with Footnotes</h1>


<p>
Needs further clarification. <a href="#1">[1]</a>
</p>

<ul>
<li><p>
First item
</p></li>
<li><p>
Second item
</p></li>
```


nagler@extremeperl.org

```
</ul>

<p>
Something about XML. <a href="#2">[2]</a>
</p>

<p>
<tt>SomeClass</tt>
<tt>$some_var</tt>
<tt>a_property</tt>
<tt>my/file/name.PL</tt>
<i>War &amp;amp; Peace</i>
"I do declare!"
</p>

 <h2>Footnotes</h2><ol> <li><a name="1"></a><p> Should appear at
the end of the chapter. </p></li> <li><a name="2"></a><p> Click
here <a href="http://www.w3c.org/XML/">http://www.w3c.org/XML/</a>
</p></li> </ol>
</body></html>
```

The footnotes are compiled at the end in a `Footnotes` section. Each footnote is linked through HTML anchor tags (`#1` and `#2`). Incremental indexes and relocatable output were the new challenges in this implementation.

15.26 Fourth SMOP

We pulled another blade out of our Swiss Army chainsaw for this task. Perl's anonymous subroutines were used to solve the footnote problem. The subroutines bound to `chapter` and `footnote` use variables to glue the footnotes to their indices and the footnotes section to the end of the chapter. Here are the additions to the SMOP:

```
 chapter => sub { my($html, $clipboard) = @_; $$html .= "<h2>Footnotes</h2><ol>\n
if $clipboard->{footnotes}; return "<html><body>$$html</body></html>";
}, citetitle => [&#39;i&#39;], classname => [&#39;tt&#39;], footnote
=> sub { my($html, $clipboard) = @_; $clipboard->{footnote_idx}++;
$clipboard->{footnotes} .= qq(<li><a name="$clipboard->{footnote_idx}"></a>$$html
return qq(<a href="#$clipboard->{footnote_idx}">) . "[$clipboard->{footnote_idx}]
```


nagler@extremeperl.org

```
}, itemizedlist => [&#39;ul&#39;], listitem => [&#39;li&#39;], property
=> [&#39;tt&#39;], quote => { prefix => &#39;"&#39;, suffix => &#39;"&#39;,
}, systemitem => sub { my($html) = @_; return qq(<a href="$$html">$$html</a>);
}, varname => [&#39;tt&#39;],
```

We didn't see a simple functional solution. Although it's certainly possible to avoid the introduction of `$clipboard`, we let laziness win out over dogma. There was no point in smashing our collective head against a brick wall when an obvious solution was staring right at us. Besides, you've got enough functional programming examples already, so you can stop standing on your head and read this code right side up.

15.27 Fourth SMOP Interpreter

The interpreter changed minimally:

```
sub to_html {
  my($self, $xml_file) = @_;
  return _to_html(
    &#39;&#39;,
    XML::Parser->new(Style => &#39;Tree&#39;)->parsefile($xml_file),
{});
}

sub _eval_op {
  my($op, $html, $clipboard) = @_; return $op->($html, $clipboard)
if ref($op) eq &#39;CODE&#39;;
  return $op->{prefix} . $$html . $op->{suffix};
}
```

`$clipboard` is initialized as a reference to an empty hash by `to_html`. If `$op` is a `CODE` reference, `_eval_op` invokes the subroutine with `$clipboard` and the html generated by the children of the current tag. The anonymous subroutines bound to the tags can then use all of Perl to fulfill their mapping obligation.

nagler@extremeperl.org

15.28 Object-Oriented Programming

`$clipboard` is a reference to a simple hash. An alternative solution would be to instantiate `Bivio::DocBook::XML`, and to store `footnote_idx` and `footnotes` in its object fields.

Objects are very useful, but they would be overkill here. To instantiate `Bivio::DocBook::XML` in Perl, it's traditional to declare a factory method called `new` to construct the object. This would clutter the interface with another method. We also have the option in Perl to `bless` a hash reference inline to instantiate the object. In either case, an objectified hash reference is more complex than a simple hash, and does not add value. The semantics are not attached to the hash but are embedded in the anonymous subroutines. Objects as simple state containers are unnecessarily complex.

Additionally, object field values are less private than those stored in `$clipboard`. An object has fields to enable communication between external calls, for example, a file handle has an internal buffer and an index so that successive `read` calls know what to return. However, it's common to abuse object fields for intra-call communication, just like global variables are abused in structured languages (C, FORTRAN, Pascal, etc.). In most pure object-oriented languages, there's no practical alternative to object fields to pass multiple temporary values to private methods. Choice is one of Perl's strengths, and a simple hash localizes the temporary variable references to the subroutines that need them.

Hashes and lists are the building blocks of functional programming. Perl and most functional languages include them as primitive data types. It's the simple syntax of a Perl hash that makes the SMOPs in this chapter easy to read. In many languages, constructing and using a hash is cumbersome, and SMOP languages like this one are unnatural and hard to read, defeating their purpose.

In object-oriented programming, state and function are inextricably bound together. Encapsulating state and function in objects is useful. However, if all you've got is a hammer, every problem looks like a nail. In functional programming, state and function are distinct entities. Functional languages decouple function reuse from state sharing, giving programmers two independent tools instead of one.

15.29 Success!

The first iteration is complete. We added all the business value the customer has asked for. The customer can translate a complete chapter. Time for a victory dance! Yeeha!

Now sit down and stop hooting. We're not through yet. The customer gave us some time to clean up our code for this book. It's time for a little refactoring. We missed a couple of things, and the code could be more functional.

15.30 Virtual Pair Programming

The second iteration evolved from some review comments by Stas. I wrangled him into partnering with me after he suggested the code could be more functional. The one hitch was that Stas lives in Australia, and I live in the U.S.

Pair programming with someone you've never met and who lives on the other side of the world is challenging. Stas was patient with me, and he paired remotely before.[15] His contribution was worth the hassle, and I learned a lot from the experience. The fact that he lived in Australia was an added bonus. After all, he was already standing on his head from my perspective, and he was always a day ahead of me.

[15] Stas Bekman co-wrote the book *Practical mod_perl* with Eric Cholet who lives in France. Stas is also an active contributor to the mod_perl code base and documentation (http://perl.apache.org).

15.31 Open Source Development with XP

Correspondence coding is quite common. Many open source projects, such as GNU, Apache, and Linux, are developed by people who live apart and sometimes have never met, as was the case with Stas and me. Open source development is on the rise as result of our increased communications capabilities. The Internet and the global telecommunication network enables us to practice XP remotely almost as easily as we can locally.

Huge collective repositories, such as http://www.sourceforge.net and http://www.cpan.org, enable geographically challenged teams to share code as easily as groups of developers working in the same building. Sometimes it's easier to share on the Internet than within some corporate development environments I've worked in! Open source encourages developers to program egolessly. You have to expect feedback when you share your code. More importantly, open source projects are initiated, are used, and improve, because a problem needs to be solved, often quickly. Resources are usually limited, so a simple story is all that is required to begin development.

Open source and XP are a natural fit. As I've noted before, Perl–one of the oldest open source projects–shares many of XP's values. CPAN is Perl's collective repository. Testing is a core practice in the Perl community. Simplicity is what makes Perl so accessible to beginners. Feedback is what makes Perl so robust. And so on. Open source customers may not speak in one voice, so you need to listen to them carefully, and unify their requests. But, pair programming is possible with practice.

Geographically challenged programmers can communicate as effectively as two programmers sitting at the same computer. It's our attitude that affects the quality of the communication. Stas and I wanted to work together, and we communicated well despite our physical separation and lack of common experience. Open source works for the same reason: programmers want it to.

To learn more about the open source development, read the book *Open Sources: Voices from the Open Source Revolution*, edited by Chris DiBona et al. available in paperback and also online at http://www.oreilly.com/catalog/opensources.

nagler@extremeperl.org

15.32 Deviance Testing

We forgot to test for deviance in the prior iteration. `XML::Parser` handles missing or incomplete tags, so we don't need to test for them here. The unit test should avoid testing other APIs to keep the overall test suite size as small as possible. However, `XML::Parser` treats all tags equally, and `Bivio::XML::DocBook` should die if a tag is not in the SMOP. We added the following test (`05-dev.xml`) to validate this case:

```
<chapter>
<unknowntag></unknowntag>
</chapter>
```

The case tests that `_lookup_op` throws an exception when it encounters `unknowntag`.

The unit test had to change to expect a `die` for deviance cases. We also made the code more functional:

```
#!perl -w
use strict;
use Bivio::Test;
use Bivio::IO::File;
Bivio::Test->new(&#39;Bivio::XML::DocBook&#39;)->unit([
  &#39;Bivio::XML::DocBook&#39; => [
    to_html => [ map({ ["$_.xml"] => $_ =~ /dev/ ? Bivio::DieCode->DIE
: [Bivio::IO::File->read("$_.html")]; } sort(map(/(.*)\.xml$/, <DocBook/*.xml>))),
    ],
  ],
]);
```

The `map` inside the `sort` returns the case base names (`DocBook/01`, `DocBook/05-dev`, etc.), and the outer `map` reconstructs the filenames from them. This purely functional solution is shorter than the previous version.

If the case file name matches the `/dev/` regular expression, the `map` declares the deviance case by returning a `Bivio::DieCode` as the expected value. Otherwise, the input file is conformant, and the `map` returns the expected HTML wrapped in an array.

`Bivio::Test` lets us declare deviance and conformance cases similarly.

nagler@extremeperl.org

When picking or building your test infrastructure, make sure deviance case handling is built in. If it's hard to test APIs that `die`, you'll probably write fewer tests for the many error branches in your code.

15.33 Final Implementation

The final SMOP and interpreter are shown together with comments, and POD, and changes highlighted:

```
package Bivio::XML::DocBook;
use strict;
our($VERSION) = sprintf(&#39;%d.%02d&#39;, q$Revision: 1.10 $ =~ /\d+/g);

=head1 NAME

Bivio::XML::DocBook - converts DocBook XML files to HTML

=head1 SYNOPSIS

  use Bivio::XML::DocBook;
  my($html_ref) = Bivio::XML::DocBook->to_html($xml_file);

=head1 DESCRIPTION

C<Bivio::XML::DocBook> converts DocBook XML files to HTML.  The mapping is only
partially implemented.  It&#39;s good enough to convert a simple chapter.

=cut

#=IMPORTS
use Bivio::IO::File ();
use HTML::Entities ();
use XML::Parser ();

#=VARIABLES
my($_XML_TO_HTML_PROGRAM) = {
  attribution => {
    prefix => &#39;<div align="right">-- &#39;,
    suffix => &#39;</div>&#39;,
```

```
},
blockquote => [&#39;blockquote&#39;],
&#39;chapter/title&#39; => [&#39;h1&#39;],
chapter => sub {
  my($html, $clipboard) = @_;
  $$html .= "<h2>Footnotes</h2><ol>\n$clipboard->{footnotes}</ol>\n"
    if $clipboard->{footnotes};
  return "<html><body>$$html</body></html>";
},
citetitle => [&#39;i&#39;],
classname => [&#39;tt&#39;],
command => [&#39;tt&#39;],
emphasis => [&#39;b&#39;],
epigraph => [],
filename => [&#39;tt&#39;],
footnote => sub {
  my($html, $clipboard) = @_;
  $clipboard->{footnote_idx}++;
  $clipboard->{footnotes}
    .= qq(<li><a name="$clipboard->{footnote_idx}"></a>$$html</li>\n);
  return qq(<a href="#$clipboard->{footnote_idx}">)
    . "[$clipboard->{footnote_idx}]</a>";
},
function => [&#39;tt&#39;],
itemizedlist => [&#39;ul&#39;],
listitem => [&#39;li&#39;],
literal => [&#39;tt&#39;],
para => [&#39;p&#39;],
programlisting => [&#39;blockquote&#39;, &#39;pre&#39;],
property => [&#39;tt&#39;],
quote => {
  prefix => &#39;"&#39;,
  suffix => &#39;"&#39;,
},
sect1 => [],
&#39;sect1/title&#39; => [&#39;h2&#39;],
simplesect => [],
systemitem => sub {
  my($html) = @_;
  return qq(<a href="$$html">$$html</a>);
```


nagler@extremeperl.org

```
  },
  varname => [&#39;tt&#39;],
};

=head1 METHODS

=cut

=for html <a name="to_html"></a>

=head2 to_html(string xml_file) : string_ref

Converts I<xml_file> from XML to HTML.  Dies if the XML is not well-formed or
if a tag is not handled by the mapping.  See the initialization of
$_XML_TO_HTML_PROGRAM for the list of handled tags.

=cut

sub to_html {
  my($self, $xml_file) = @_;
  return _to_html(
    &#39;&#39;,
    XML::Parser->new(Style => &#39;Tree&#39;)->parsefile($xml_file),
    {});
}

#=PRIVATE SUBROUTINES

# _eval_child(string tag, array_ref children, string parent_tag, hash_ref clipboa
#
# Look up $tag in context of $parent_tag to find operator, evaluate $children,
# and then evaluate the found operator.  Returns the result of _eval_op.
# Modifies $children so this routine is not idempotent.
#
sub _eval_child {
  my($tag, $children, $parent_tag, $clipboard) = @_;
  return HTML::Entities::encode($children)
    unless $tag;
  # We ignore the attributes for now.
  shift(@$children);
```

```
  return _eval_op(
    _lookup_op($tag, $parent_tag),
    _to_html($tag, $children, $clipboard),
    $clipboard);
}

# _eval_op(any op, string_ref html, hash_ref clipboard) : string
#
# Wraps $html in HTML tags defined by $op.  If $op is a ARRAY, call
# _to_tags() to convert the simple tag names to form the prefix and
# suffix.  If $op is a HASH, use the explicit prefix and suffix.  If $op
# is CODE, call the subroutine with $html and $clipboard.  Dies if
# $op&#39;s type is not handled (program error in $_XML_TO_HTML_PROGRAM).
#
sub _eval_op {
  my($op, $html, $clipboard) = @_; return &#39;ARRAY&#39; eq ref($op)
? _to_tags($op, &#39;&#39;) . $$html . _to_tags([reverse(@$op)],
&#39;/&#39;) : &#39;HASH&#39; eq ref($op) ? $op->{prefix} . $$html
. $op->{suffix} : &#39;CODE&#39; eq ref($op) ? $op->($html, $clipboard)
: die(ref($op) || $op, &#39;: invalid $_XML_TO_HTML_PROGRAM op&#39;);
}

# _lookup_op(string tag, string parent_tag) : hash_ref
#
# Lookup $parent_tag/$tag or $tag in $_XML_TO_HTML_PROGRAM and return.
# Dies if not found.
#
sub _lookup_op {
  my($tag, $parent_tag) = @_;
  return $_XML_TO_HTML_PROGRAM->{"$parent_tag/$tag"}
    || $_XML_TO_HTML_PROGRAM->{$tag}
    || die("$parent_tag/$tag: unhandled tag");
}

# _to_html(string tag, array_ref children, hash_ref clipboard) : string_ref
#
# Concatenate evaluation of $children and return the resultant HTML.
#
sub _to_html {
  my($tag, $children, $clipboard) = @_; return \(join(&#39;&#39;,
```


nagler@extremeperl.org

```
map({ _eval_child(@$children[2 * $_ .. 2 * $_ + 1], $tag, $clipboard);
} 0 .. @$children / 2 - 1), ));
}

# _to_tags(array_ref names, string prefix) : string
#
# Converts @$names to HTML tags with prefix (&#39;/&#39; or &#39;&#39;), and con
# the tags into a string.
#
sub _to_tags {
  my($names, $prefix) = @_;
  return join(&#39;&#39;, map({"<$prefix$_>"} @$names));
}

1;
```

To keep the explanation brief and your attention longer, here are the list of changes we made in the order they appear above:

- The `attribution` mapping was not fully compliant HTML. Values must be surrounded by quotes.

- The compilation of `$_XML_TO_HTML_PROGRAM` was eliminated. This version is less complex, and is not perceptibly slower.

- `_eval_op` implements the SMOP operator based on its type. Stas and I had a (too) long discussion about the formatting and statement choices. Do you prefer the version above or would you like to see a if/elsif/else construct? The former is functional, and the latter is imperative.

- `_to_html` was refactored to be a pure function by replacing the `while` and `$res` with a `join` and a `map`. The implementation is no longer destructive. The `splice` in the previous version modified `$children`. `_eval_child` is still destructive, however.

- `_to_tags` was renamed from `_compile_tags_to_html`.

nagler@extremeperl.org

15.34 Separate Concerns

This completes the second and final iteration of our DocBook XML to HTML translator. The second iteration didn't change anything the customer would notice, but it improved the program's quality. Pride of craftsmanship is a motivating factor for most people. The customer benefits directly when programmers are giving the freedom to fix up their code like this. Quality is the intangible output of motivated people.

Craftsmanship plays a role in many professions. For example, one of my favorite pastimes is baking bread. It's hard to bake well. There are so many variables, such as, ambient temperature, humidity, and altitude. A skilled baker knows how to balance them all.

Anybody can bake bread by following a recipe. Just buy the ingredients, and follow the step-by-step instructions. These days even inexpensive kitchen appliances can do it. While fresh bread from a bread machine tastes fine, it wouldn't win a competition against a skilled baker. My bread wouldn't either, but I might beat out a bread machine.

Becoming a skilled baker takes practice. Following a recipe isn't enough. Indeed, most good bakers instinctively adjust the recipe for temperature, humidity, altitude, and so on. They probably won't follow the instructions exactly as written either. A simple recipe tells them what the customer wants, that is, the ingredient combination, but the know how of a good baker would fill a book.

When you separate the what from the how, you get qualitative differences that are impossible to specify. In the case of our translator, the SMOP is the what and the interpreter is the how. The quality is the succinctness of the mapping from DocBook to HTML. The program is less than 100 lines of Perl without documentation, and we can add new mappings with just one line. You can't get more concise than that.

XP achieves quality by asking the customer what she wants and allowing programmers to implement it the best they know how. The feedback built in to XP gives the customer confidence that she's getting what she wants, much like the feedback from testing tells the programmers the code does what they want.

In plan-driven methodologies, the lines between the what and the how are blurred. Specifications are often an excuse for the customers and analysts to attempt to control how the programmers code. While the aim is to ensure quality, the result is often the opposite. The programmers act like unthinking automatons following the specifications to the letter, even when they know the spec is wrong.

nagler@extremeperl.org

The programmers are craftsmen, and XP respects their knowledge and experience. The customer is also a craftsmen, and XP teaches programmers to respect her skills, too. XP separates concerns to allow people to excel at their jobs.

15.35 Travel Light

When two craftsmen communicate, you don't hear much. Acronyms abound. Their common experience lets them skip over the details lay people need spelled out for them, like a recipe.

Perl, XP, and SMOP are terse. In Perl, you don't call the `regular_expression` function, you say `//`. A skilled Perl programmer reads and writes `//` instinctively. An XP customer writes brief story cards without a thought about whether the programmers will understand them. She knows the programmers will ask for elaboration. There's no need for big fat stateful specifications and programs to slow down the pipeline from the customer to the programmers to the computer to deliver value to the users.

An experienced traveler knows that the more baggage you carry, the harder it is to change planes, switch trains, and climb mountains. Extreme Perl works best when you drop those bags, and hit the ground running. Brief plans change easily when reality happens. Concise code adapts to changed plans rapidly. Travel light, and get there faster.

nagler@extremeperl.org

Index