Foundations of Agile Python Development

Copyright © 2008 by Jeff Younker

The source code for this book is available to readers at http://www.apress.com.

■ ■ ■

What Is Agile Development?

Agile development is a term given to an entire class of iterative development methodologies. Their unifying characteristic is a focus on short development cycles, on the scale of weeks rather than months. Each development cycle, referred to as an iteration or sprint, produces a working product. This chapter introduces the motivations for the movement to agile software development and surveys the practices that commonly constitute these methodologies.

These practices, in the order to be discussed, are as follows:

- Pair programming

- User stories

- The system metaphor

- On-site customers

- Unit tests

- Test-driven development (TDD)

- Refactoring

- Simple design

- Short iterations

- Collective code ownership

- Continuous reflection

- Continuous integration

- Documentation

Why More Methodologies?

Some projects succeed and some projects fail. This happens regardless of what development methods are used. Development is about much more than simply the techniques that are used. Good development depends upon a strong grounding in reality; not everything can be known before a project starts, and this must be taken into account when planning. Some of these new facts will be minor, and some will be major.

Accommodating major new facts often requires hard choices to be made; making these hard decisions requires sound judgment, and even then, the sound judgments are sometimes wrong. Making these judgments requires guts and integrity; a project leader who is unwilling to stand up and tell the truth potentially sacrifices the development organization's well-being, the product's quality, and possibly the whole organization's long-term viability. This holds true no matter how the project is developed.

These days, the waterfall methodology is a favorite whipping boy. If you're advocating something strongly, then it helps to have something else to demonize, and many agilists have fastened upon the waterfall methodology for that purpose. There's a lot of software out there that has been developed nominally using the waterfall method; whether the engineering staff actually followed the documents is open to question. The waterfall methodology reflects the aspirations of many toward producing better software, and it reflects the best understanding that was available at the time, but there are valid criticisms that have been leveled against it:

It assumes that all change can be predicted and described up front. Software is created to serve a purpose, but if the conditions in the world change, then development needs to change to reflect the new realities. This often happens in the middle of a project. A new disruptive technology is released. New versions of interoperating software are released, altering dependencies and interactions; the new software has features that duplicate functionality being implemented or changes how the existing functionality works. New competitors may come into the market, or the regulatory environment may change. The world is simply too complicated to anticipate all changes up front.

Exploratory tasks that should be part of development are pushed into the design phase. Many judgments about suitability can only be addressed through the creation of proto-types. Many times, determining how something can be designed most effectively requires building a substantial part of it. Why should this effort be wasted?

The waterfall methodology also assumes that documentation can be sufficient to describe a system. There are fields with far more detailed and elaborate documentation systems than are found in software development; mathematics, medicine, and the law are three examples. Nobody in these fields has the hubris to say that documentation alone is sufficient for achieving understanding. Instead, they recognize that a person cannot become an expert without tutelage.

A software specification detailed enough to unambiguously describe the system is spe-cific enough to be translated automatically to software. Such a process simply pushes the effort of coding into design, yet if this is done without feedback from operating models, the design will have errors.

Agile methods emphasize accommodating change, group communication, and iterative design and development. They attempt to cast off excess process. Some of it is just jettisoned; some of it is replaced by other practices. Agile methodologies range from extreme program-ming (XP), which focuses almost exclusively on the developer and development techniques, to the Dynamic Systems Development Method (DSDM), which focuses almost completely on processes—but they all have similarities.

A Little History

Although the term *agile*, as it relates to software development, dates from early 2001, agile methodologies have been in use for much longer. They used to be called iterative methodologies. Today's particular bunch were called lightweight development methodologies before the Manifesto for Agile Software Development was produced in February, 2001. (Seems someone didn't like being called a lightweight!)

> ***Individuals and interactions*** *over processes and tools*
> ***Working software*** *over comprehensive documentation*
> ***Customer collaboration*** *over contract negotiation*
> ***Responding to change*** *over following a plan*
> *That is, while there is value in the items on the right, we value the items on the left more.*[1]
>
> —Manifesto for Agile Software Development

A few years ago, people looked on agile development practices with great suspicion. The term was almost ridiculed in some circles. These days there is more respect paid, and these practices are making significant inroads. Most organizations I've worked with have flirted with agile methods. Developers are learning what works, either on their own projects or from experiences at other companies, and agile practices are spreading, often under the radar of the larger development organization.

Arguably, the wider adoption of agile methods reflects an underlying change in technology. This change began in the early '80s with the wide-scale introduction of personal computers. At that point, computing power was expensive and people's time was comparatively cheap. Experiments and prototyping were unknown. The ability to run hundreds or thousands of tests in a few seconds was fantasy. The idea of setting up and tearing down a SQL database in memory was absurd. A generation has passed, and that relationship has reversed. Development methods are finally catching up with the changes in technology, and the lessons learned from physical manufacturing in the '80s and '90s are also being felt.

While the various agile techniques are useful on their own, they have strong synergistic effects. One practice enables another to be used more effectively, so the payoff from using them in combination is often larger than for using them separately. I've tried to note such interactions in this chapter.

This chapter aims to show you what those methods are. I'll try to explain how they tie together. Some that relate to process won't be covered in this book, but those relating to tools will be. These are the same practices that are easiest to bring in the back door as a developer.

---

1. The Manifesto for Agile Software Development is available at `http://agilemanifesto.org/`. The authors are Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. It may be freely copied in any form, but only through to this notice.

Planning and Agile Development

Proponents of agile development methods often give short shrift to planning. I feel this is an overreaction to "big design up front" (BDUF), a practice often condemned by agile advocates. Planning is critical to any project. At the very least, the development team needs to know the broad scope and the intended form of the finished product; for example, a hosted solution is very different from shrink-wrapped software. It is important to defer coding until you have a basic grasp of what you are trying to build.

Agile methods aren't a license to go flying off in any direction. The admonition that an agile team should have expertise in the problem domain is often underplayed. This require-ment for experience allows advocates to underplay the role of planning, because if you've built it once before, you've already invested the effort in planning once, and doing the same thing again is a slam dunk. In the more interesting and challenging cases, this is not true. At these times, it pays to sit down and think about the voyage you're embarking upon and the path that will take you to your destination. Failure to do this leads to failure.

I've been witness to an agile project that reached a dead end. The architecture the team had evolved couldn't cope with the new requirements. The team scrapped what they had done, and they launched into the process of rewriting the application. Rather than building in the desired architecture from the beginning, they dogmatically pursued the same evolutionary process that they had used the first time. Unsurprisingly, they ended up at the same dead end again. (To be fair, the outcome was foreseen by at least one Cassandra on the team, but she was ignored.) Eventually, they dug themselves out, but at the expense of quite a bit of devel-oper time.

This leads to a conjecture that some recent work supports: agile development methods are excellent tools for producing locally optimal designs, but on their own they are insufficient to produce globally optimal designs. Development techniques are no substitute for a thor-ough understanding of the problem domain. You still need experts, and you still need to comprehend the big picture.

What Are Agile Methods?

Agile methods are a collection of different techniques that can be used in conjunction to achieve high software quality and accurate estimates of time and material with shorter devel-opment cycles. The laundry list includes pair programming, user stories, TDD, refactoring, simple design, rapid turnaround/short iterations, continuous integration, a consistent system metaphor, on-site customers, collective code ownership, continual readjustment of the devel-opment process, and believe it or not, documentation. The things relating to specific tools will be covered deeply in this book, but those relating to process will only be touched upon lightly.

The first insight into agile methods is that all software development is software mainte-nance. Feature development and feature maintenance are one and the same. Your software should always be functional. It may not have the full functionality of the finished application, but it should always be runnable and installable.

The second major insight is that source code is not a product. It is a blueprint for a prod-uct. The final product is packaged object code, or in some environments live code running on production hardware. What goes into the process is a design, and what comes out of the com-piler or the interpreter is the product. A single project may in fact produce multiple programs or multiple packagings for different architectures.

This is a somewhat provocative statement, but there is a good deal of literature to back it up.

It seems less absurd when you examine how other manufacturing processes are becoming more like software. Once upon a time, design was only a small part of producing an ornate steel scrollwork grill. Production required days if not weeks of work. The pattern was drawn or scraped into the metal. The metal was heated, banged out, banged back into shape, and then reheated. This was done over and over, and the process proceeded inch by inch. When completed, each edge had to be filed down to smoothness.

The process has gotten faster over the last 200 years. Oxyacetylene torches easily make gross cuts, eliminating the need to heat and reheat. Angle grinders dramatically sped up the filing process. Plasma cutters made gross cuts even easier; cutting steel with a plasma cutter is like cutting warm butter with a steak knife, but it's still a manufacturing skill requiring hand-eye coordination.

Today there are computer-controlled cutting tables. You feed in a blueprint, load a sheet of metal, and press a button, and a few minutes later the grillwork is complete. Design has become the primary activity.

Writing software is not producing new features, but instead designing them. Similarly, rewriting existing software is really redesigning old features. Every software developer is also an architect. The two roles are one and the same. Producing software becomes an entirely automatic process that is often maintained by specialists (often referred to as *release engineers*).

So what are these methods about? Well, I'm going to start with one that I don't cover elsewhere in this book: pair programming.

Pair Programming

Pair programming is the most controversial of the bunch. Quite simply put, most programmers aren't that productive. Let's face it, programming is lonely, and we're social creatures. So programmers end up wasting half their day. They spend time reading e-mail, whether personal or company. They surf the Web. They fall into conversations with coworkers. To some extent, they are just trying to engage with other human beings.

Working alone with a computer has a strange effect on the human mind. The computer gives rewards and feedback, but it doesn't engage our limbic system—that layer of gray matter that distinguishes the mammalian brain from that of a reptile. It's what allows us to be caring parents and friends; it's what lets us feel another's pain or love. Frequently, programmers find themselves in strange state of mind; many programmers I know refer to it simply as *code space*. As a profession, we don't talk about it much. It's a place isolated from the rest of the human race. It takes time to come back from code space, often hours, and those are the hours that we have to spend with our families and friends.

Put two programmers together and their work becomes a social activity. They work together. They don't get stuck, they keep each other from being distracted, they don't go into code space, and they're happier at the end of the day. At worst, you haven't lost any productivity, and you've increased employee morale.

Pair programming arguably improves code quality. It works because it is a form of constant *code review*. Code reviews are a process in which programmers review and make suggestions about another developer's code. Code reviews have been found to consistently

decrease the number of bugs per thousand lines of code, and they have been found to be more effective at doing this than any other single measure.

Pair programming transfers knowledge between team members. In typical development environments, programmer-to-programmer learning is limited to brief exchanges. Those exchanges may be meetings in the break room, conversations in the hall, or formal meetings. In pair programming, the exchanges extend through the entire day. When people spend time together asking questions, they get to know each other. They lower their barriers, and they're willing to ask stupid questions that they'd otherwise spend all day researching.

A bullpen layout is often used with pair programming to facilitate exchanges between programmers. In a bullpen, there is no obstacle between you and the next person. There is nothing to stand between you and the person to your left when you need to ask a question. If you need help from someone who knows a given section of code, you can turn around and ask them.

A word often used in conjunction with pair programming is *collocation*. It refers to teams that are in the same location and can freely exchange ideas. It should be noted that a team that shares a single floor or building is not necessarily regarded as collocated. If there are physical barriers between programmers, then they are isolated. Walls impede the free exchange of ideas and information. The classic barrier is the cube wall.

This immediately brings to mind an office brimming with noise and distractions. Classically, these are death to programmer productivity, but in unpaired environments, programmers are trying to isolate themselves from other human beings. This isn't the case when pairing.

It's not that excessive noise and distractions aren't a problem with pair programming. It's that the programmers are engaged with their partners. As a species, we're very good at carrying on conversations with other people and ignoring our larger environment. When we're doing that, it takes much more to interrupt the flow of thoughts. Think of all the wonderful conversations that people have in restaurants and cafes. We can play immensely engaging games in such environments; chess and bridge come to mind. If the volume gets too high, then concentration will break down, but the environment has to get really raucous for that to happen. You want to work in a cafe rather than a night club, but that still leaves a wide range of environmental choices.

In any development organization, there is a huge amount of information stored in the heads of the developers. That knowledge will never be completely transferred to paper or bits. To do so would take more time and money than is available. Think of the difficulty of maintaining someone else's code when they are no longer around. What they could answer in seconds will take you minutes or hours to fathom out.

Code bases are full of questionable constructs. Pairing serves to spread the explanations from person to person. In order to understand what one person is doing, the other has to ask these questions.

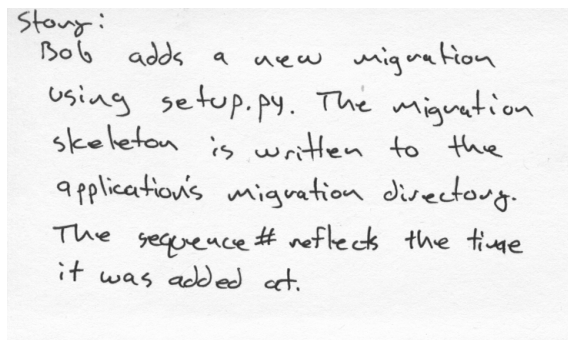Pairs are fluid. Programmers pair with different programmers every few days. This spreads the knowledge around. Knowledge spreads like a virus. One person knows something in the beginning. They pair with someone. Now two people know. They move on to different pairs, and now four people now know. The more pairings you have, the more it spreads. This protects the development group from the loss of any one programmer.

■**Caution** Viruses spread like viruses too. Presenteeism and pair programming are a bad combination. If you're pairing and you get sick, then please go home and rest. The rest of us want to stay well.

All professions involve a large element of social learning. Lawyers and doctors have internships in which they engage with mentors and peers. In some medical schools, people work together in teams. Mathematicians, members of the classic loner profession, actually spend a huge amount of time in front of blackboards hashing out ideas together. Coffee fuels them, but it's usually flavored with chalk dust. Pair programming recognizes our natural strengths as social creatures and works with them.

User Stories

User stories are short descriptions of features to be written. They describe a small piece of functionality that can be broken down into tasks whose durations can be quickly estimated. They should fit on an index card (see Figure 1-1). They determine what we are going to produce. If it's not in a user story, we shouldn't be coding it. In a perfect world, user stories would determine every feature that goes into the software.

Story:
Bob adds a new migration using setup.py. The migration skeleton is written to the application's migration directory. The sequence # reflects the time it was added at.

Figure 1-1. *Card with a user story*

User stories are produced in conjunction with the customer. They are a distillation of everything the customer knows. More importantly, they are the distillation of what the customer wants and what can be produced by the programmers. It is important for programmers and management to be involved in their creation, as they provide a technical check on customers' wild dreams. It is important that the dreams be those of the customer, though, as the programmers probably don't have as firm a grasp on the business problems as they'd like to believe.

Some user stories are produced just by the development organization. These relate to the internals of the software. They may describe a new data structure (say, a B-tree) or module (perhaps an object store) to be created.

User stories alone are often insufficient to specify the software's behavior fully. At those frequent points where clarification and elaboration are required, the on-site customer should be consulted. This results in a direct transfer of knowledge from the customer to the coder, bypassing the interpretation that would be imposed by more formal documentation.

The principal difference between a user story and a use case is that the effort required to complete a user story can be easily estimated.

The System Metaphor

The *system metaphor* allows you to talk about your design in a consistent and unambiguous way. There is just one way that the code, the developers, the managers, and the customers talk about the design. When everyone speaks the same vocabulary, meetings and discussions flow smoothly. (We've all been in those interminable meetings where everyone goes back and forth, confused over what a handful of words mean.)

The system metaphor should be used throughout the project. It should be used when discussing the project, in the user stories, and throughout the code base.

A variable or a function with a name that conflicts with the system metaphor should be treated as a bug. There is a children's game called telephone (also known as Chinese whispers). In it, a group of children sit in a line or a circle. The person at the beginning whispers a short phrase or sentence into the ear of the person next to them. That person repeats the phrase to the next person, and this continues until the phrase reaches the end, where the last person announces it. After the first person stops laughing at how far the phrase has been transmuted, they tell the group what the starting phrase was. Rarely if ever does a phrase make it through the line intact.

Naming things is a bit like the game of telephone. You look for a name that is close, distinctive, and appropriate. The further your base name is from the system metaphor, the further your new names are going to be from that. The meaning drifts, and eventually it becomes unintelligible to everyone except you. When someone returns to your code six months from now, they're going to be lost in the mess of unfamiliar terminology. That someone may be you. Therefore, names inconsistent with the system metaphor should be fixed immediately, and you should refer to the system metaphor when even slightly in doubt.

Automatic refactoring tools in a modern IDE (integrated development environment) help with changing names across the system. The process is almost magical. Chapter 7 will cover the use of refactoring tools in the Eclipse IDE.

On-Site Customers

On-site customers allow you to get feedback from someone who will actually use the product. Nothing tells you that a feature is off track as fast as a user saying, "What's that for?" When you have questions about what a user story means (of course, written in terms of the system metaphor), you can get an answer from the customer rather than guessing.

Few specifications are ever complete, particularly when the specification is written on 3×5 index cards. The job of the programmer is to turn rough specifications into precise and unambiguous instructions. Much of that translation is based on knowledge about how the computer works, but much of it is based on domain knowledge, and that knowledge belongs to the customer.

Having a customer available saves the programmer from having to make guesses about the domain. Making guesses is expensive. A wrong guess is a bug. It is a well-substantiated

observation that the further a bug makes it through the development process, the more expensive it is to fix.

The customer also serves as broad functional QA. She gets to see things along the way. She can catch situations where the developers and customer didn't communicate quite well enough. This prevents functional misapprehensions (bugs) from getting further into the development process.

The customer and the development team should communicate using the system metaphor. If they don't, then confusion can ensue, and the developers can produce the wrong thing.

Having the customer on site speeds up this process of interaction. The programmers don't have to wait for the customer to get back to them. If the programmers have to wait, then they lose their train of thought or have to switch tasks. This takes time. The easier it is for the programmer to get hold of the customer, the more likely he is to do it.

In the real world, we sometimes have disagreeable customers, or customers who feel that interacting with the development team is a distraction from their real work. Often they have been burned in the past. Short iterations of work help with this because the customer gets feedback. Good old-fashioned social engineering can help too. Never underestimate the power of a random thoughtful gift.

The more familiar the programmers are with the customer, the more likely they are to go to the customer for assistance. The more familiar the customer is with the programmers, the more likely she is to be happy to offer assistance. It is important that the customer and the programmers feel comfortable with each other. For this reason, among others, the customer should be included in the development group's work and social activities as much as possible.

Unit Tests

Unit tests are also called programmer tests. These are the tests that you write to verify the operation of your code. These tests aren't at the level of features. They are at the level of methods and functions. They allow you to check assertions during development, like, "If I pass in an empty list, does this function raise an exception?" Or, "If I pass in an unknown user, does this method return `False`?"

One of the joys of working with an interactive language is getting rapid feedback. You type a command, and you can see its output. You can verify that it's what you expected. You make a change to a program, you run the program, and you can immediately see the result. In noninteractive languages, you write a small program just to test the new functionality. You run the test programs, and in your mind you check that the result is what you expect. You conclude that the code worked, and you move on to the next chunk of code.

Unit testing formalizes that. You put all of those little test programs in a common place. Instead of checking the results manually, you write code to check the results. You package all of that up inside a harness to run the tests and report all their results. This way, you have a record of everything that you expect the code to do, and you can verify conformance at any point by rerunning the tests.

Unit tests should be pervasive. If your code isn't being tested, then you don't actually know that it is working. You trust that it is working, but you have no precise means of verifying this. Unit tests provide that means. They need to be pervasive because many bugs are non-local.

Code in one section of the code base can interact with code in another section of the code base. If you're not testing the entire code base, then you can miss these far-flung effects. These errors pile up, and soon you reach a situation that is referred to as "playing whack-a-mole."[2] The cause is often an underlying set of bugs elsewhere in the code. Pervasive unit tests let you see all of the problems at once.

Unit tests should be run after every change. Bugs interact, and the number of interactions doesn't increase linearly. One bug doesn't gives you one error. Two bugs don't necessarily give you two errors. They can give you four errors or more. Three bugs can give you eight or more. This is where the whack-a-mole situation comes from. Every time you make a change, you can introduce a bug. Running unit tests immediately after you make a change allows you to see a new bug as soon as it is created and before it has a chance to interact with other bugs. Fixing the bug is quick and cheap at that point. The more changes between runs of the unit test, the harder it will be to fix the resulting bugs.

Unit testing intersects with *regression testing*. Regression tests identify bugs that have been seen before and that have been fixed. Unit tests are a means of accomplishing regression testing, but they serve a larger purpose. When writing new features, the primary goals of unit testing are confirming expected behavior and identifying bugs resulting from the new functionality. Unit test runs should be fast. If test runs take too long, then programmers won't run the tests as often. As noted previously, the less often the unit tests are run, the more errors will accumulate. The more errors that accumulate, the more expensive they are to fix and the more time is spent fixing them.

Unit tests shouldn't be too pervasive. Each unit test has a maintenance cost associated with it. If you change the code it tests, then the unit test will need to be changed. If more than one unit test examines a section of code, then all of the tests will need to be updated. This can quickly become onerous, and once it become onerous, developers will tend to stop maintaining the tests.

Unit tests shouldn't traverse code outside of the unit being tested. When this happens, a change in one package can cause a cascade of unit test failures. All of these tests need to be fixed before the test suite runs correctly again. A simple trip next door becomes an expedition. Finally, limiting the number of tests and depth of their reach promotes fast unit test runs. There are fewer of them, and they are doing less work.

In Python, we have many tools to assist with writing unit tests. Two of the most common are unittest and Nose. I'll show these to you in Chapter 6. unittest, which is older, is based on a classic design called xUnit. Knowledge of unittest will carry over to many other languages. Nose subsumes unittest, and provides a mechanism for running many tests. I'll show you how these test frameworks can be automatically run both in the local development environment and by automated build systems.

Mock object frameworks allow you to separate packages and classes from resources that they depend on. Limiting the scope of your tests is much harder without them.

Test-Driven Development

Test-driven development (TDD) turns unit testing on its head. All programmers know what they expect code to do before they write it; TDD makes these expectations concrete. It has a

---

2. Named after the game where moles pop out of holes, and you whack them with a hammer as fast as you can, but as soon as you do, they pop up somewhere else.

unique yet satisfying rhythm. You write the test. You run the test, and it should fail. You write the fragment of code being tested. You run the test, and now it should succeed. At every step, you're performing experiments to verify that your code operates as you expect.

TDD is done at a very fine granularity. You don't write all of the tests for the program. You just write the test for the next few lines. Many of these tests will be thrown away in the end because they test intermediate states in the program's evolution. Writing the tests at a fine granularity results in methods that are very small and possess very limited functionality. The same happens with classes.

At every step of the program's development, you have to think about how you are going to test the results. The data on which each method depends must be passed to that method as part of the test. Each data transfer tends to take effort, so you tend to create fewer of them. You have to test all the side effects, so you produce fewer of them. You have to test the classes in isolation, so you produce lightly coupled classes.

In general, big nasty methods and classes produce big nasty tests. Writing nasty tests is painful, so developers don't do that. Writing cleaner code becomes the path of least resistance. Simply put, designing for testing forces us to produce better code. And because we write the test before writing the code, we end up with close to 100 percent testing coverage.

I took the opportunity to do an experiment at work once. I was working on two sets of scripts—one was a new script, and one was maintenance on one of our nastiest, longest-running, and most problematic pieces of code. I wrote the virgin scripts without TDD. I made the overhaul of the nasty scripts using TDD. The nasty scripts ended up working fairly painlessly on the first try, and subsequent changes were utterly painless. The new code wasn't bad, but it wasn't nearly as robust. I would hate to see what it's like today.

Refactoring

Refactoring is the practice of simplifying and clarifying code at every opportunity. It focuses on changes that alter the structure of the code without altering the meaning. I find refactoring quite fun, but it's dangerous if you don't have unit tests in place. Without unit tests, you don't know if you've unintentionally altered the meaning.

In some sense, refactoring is one of the most well-understood areas of software development. It is described extensively in the literature. Refactorings have names and precise definitions. Some refactorings are done with an eye to improving readability, some are done with an eye toward removing redundancy and duplication (known to some as "the death of code"), and some are done to improve modularity. Most are limited in scope.

Refactorings should be limited to small localized changes. Large refactorings tend to affect many files, classes, and methods, and entail changes to many unrelated pieces of code. They can break everything they touch, and they require rewriting many unit tests. I have personally seen a large one-shot reorganization go quite awry. It consumed over a week of a medium-sized development organization's time. Large reorganizations across the entire code base are risky.

Refactoring is the daily bread of agile programming. It's the primary tool in producing simple designs. Whenever we encounter a chunk of code that doesn't smell right, we refactor until the smell goes away. I'll cover refactoring in more detail in Chapters 6 and 7.

Refactorings are so well understood that many are downright mechanical in nature. As such, they can be done with the assistance of tools. Most IDEs these days include tools to help with refactoring. The tools are much better established in statically typed languages, since

more of the structure and semantics of the code can be inferred, but tools for Python are getting better. Eclipse running Pydev, which I'll show you in Chapter 2, has some of the best refactoring tools available for Python.

The presence of automatic refactorings is one of the primary reasons for switching to an IDE; this alone nearly drove me to abandon Emacs for Eclipse. At first, you're likely to be a little put off by using them. They seem a bit clunky, but they're worth persisting in and learning how to use correctly.

Simple Design

Simple design is about fulfilling the requirements of the user stories and no more. Your code is the design, and the design should be as simple as possible, but no simpler. Simple design is not an admonition that design is bad, or that some up-front design isn't necessary. It is an admonition that *too much* up-front design is bad. How much is a matter of judgment and experience. Simple design encompasses a number of principles, many with cute acronyms.

The first principle is *don't repeat yourself (DRY)*. There should be one and only one source of truth for any fact or assertion in your program. If there is more than one, then someone will eventually end up changing one and not the other. And if they don't catch that, then it can lead to mysterious bugs, and in the worst cases it leads to architectural mistakes. Replicated functionality is the death of good code. When you find duplicated code, refactor!

The second principle is *you're not going to need it (YAGNI)*. Useless code is expensive. Every little feature and whiz-bang gizmo has a cost associated with it, and that cost carries into the future forever. It makes the code base complicated. It derails your delivery schedule. It keeps you from going running at lunch, makes you late picking up the kids, and delays you from cooking dinner for your wife. Is that cool generalization of the argument-processing code you might use the next time you reuse this code really worth it? Probably not. You can write it when you actually need it three months from now. This afternoon, you've got better things to do, like completing that user story and then playing in the snow with your sweetie.

The third principle is *better raw than wrong*. Simple, direct, blunt communication is better than anything else. You should make your intent clear. Don't mince words. Don't look for the most elegant way of saying things. Just say things simply and directly.

The fourth principle is *do the simplest thing that could possibly work*. You can make it more complicated and robust later. That clever code is generally harder to maintain. It takes more work to write, and it takes more work to test. It's a waste of time. Make the unit tests pass, and get on to the next feature. Be Hemingway, don't be Melville (Melville was paid by the word).

Collective Code Ownership

Collective code ownership is about who fixes what. It is based on the idea that individuals should not maintain ownership of sections of code. It is one giant code base. When there are problems, everyone should feel empowered to go right in and fix the code.

Having one or two people lording control over a code fiefdom transforms it into a bottleneck. We've all experienced this: it may be good code, and it may be bad code, but it's not our code. At one time or another, we've found a bug in someone else's fiefdom. Or we've needed a new feature. Or we've needed to interface to their fiefdom in a way that just wasn't intended. We need them to make a change, and they're not available. They may be busy on a critical project. They may be on vacation. They may not like us, and they may be exercising control.

The only choices are to wait for them to become available or to fix the problem from the outside. I've seen far too many warts accumulate because people didn't want to dip into a chunk of code. Many lines of logic could be created to handle erroneous results from a buggy package when a one-line fix would be sufficient within the package itself. Sometimes arguments have to be converted to strange formats and then converted back when it would be easier to add a new input type to a case statement inside the module. Or even worse, an entirely new framework has to be adopted because nobody wants to fix the performance problems inside somebody else's package. (Yes, I actually did see this happen. And it was a two-line fix that needed to be made.) In every case, fixing the problem from the outside is the wrong solution.

If everyone is making improvements to the code base as a whole, then the code base shouldn't go rotten. When a developer finds something that smells, they are authorized to find that smell and fix it. They are not just authorized to do this; they are expected to. At every opportunity, they are expected to refactor stale code that affects their current task.

Other agile practices facilitate collective code ownership. Pair programming breeds familiarity with the code base as a whole. It forces programmers to give up sole control of their fiefdoms. Unit tests allow you to make changes with confidence. Frequent refactorings of the code base keep it from going smelly, and this encourages developers to continue to maintain it.

Short Iterations

Short iterations serve multiple purposes. They allow you to deliver a working product to your customer at regular intervals. They allow you to see how accurate your estimates are on a regular basis. Finally, they give you an opportunity to regularly reexamine your development processes. You get to plan for the future and look back at the past while everything is still fresh in your mind. Note that shorter iterations are not necessarily better. Time spans in the range of two weeks to one month seem to be values that people work with successfully.

Producing a functional product on a regular basis allows your customer to judge the outcome. Since you've produced a small number of features, the customer can examine all of them quickly. The review's coverage is complete. The review is kept short, so the customer is fresh and observant all the way through. His feedback is likely to be detailed and thorough. Miscommunications about his intent will be caught. He won't get to the point where he'll say, "*O$#&! it, it's good enough." The customer will be more satisfied with the final product as a result.

With a large number of features, some will invariably be skipped, and if they are not, the review becomes a painful slog. People are likely to get tired and bored, and tired and bored people get sloppy. Sloppy behavior results in inadequately reviewed features. Design bugs will be missed, or features won't be quite what was intended. These inadequacies will progress further into the development cycles, and may eventually be released in production software. The further they get into the development cycle, the more expensive fixing them will be.

If the iterations are short, there will never be a grand moment of shock for the customer when she asks, "What on earth did you build? That's not what I've asked for." (It's even less likely if your customer was on-site and interacting with you.) There may be times when the customer says, "That's not quite what I asked for. How about doing it like this?" and she continues to describe what she needs.

The review should be a pleasant experience for everyone. Keeping the number of new features to a minimum helps with this. The less pleasant a review, the more likely it is be put off, and the less effort is likely to be put into it. Keeping down the number of reviewed features keeps the review short and pleasant, and that keeps the customer happy.

In a classic waterfall process, estimates are made months out. On a ten-month project, a 20 percent underestimate is a two-month delay. This is the difference between finishing at Halloween and being stuck at work for Thanksgiving, Hanukkah, Christmas, and New Year.

Agile projects make short-term estimates. On a two-week iteration, a 20 percent under-estimate is only two work days. After the iteration is done, you get to produce another set of estimates. The causes of your delays will have happened days ago. The events that are going to happen in the next few weeks are likely to be known. Your estimates should be more accurate because of this, and you'll have many opportunities to learn from your misjudgments. Fewer inaccuracies should creep in, and you'll have many chances to tune the estimates before they damage the project.

Compare this with long-term estimates. The causes of inaccuracies are likely to have happened months ago. It will be hard to remember them when trying to learn how to estimate for the next project. You're not going to get many opportunities to learn how to estimate either. Many unforeseen events are likely to happen over the course of the new estimates, too. On a one-year project, it's likely that someone on your team may meet the love of their life, have a midlife crisis, or have a close relative die. These will all impact the project estimates.

Having accurate short-term estimates allows development leads to combat scope creep. When new features are requested, their effects on the schedule can be quickly and accurately determined. Management can be presented with this information, and they can be given options as to which features will have to be dropped in order to accomplish the new tasks. If estimates have been fed back to management throughout the development process, then they will have trust in the truth of these judgments. With long time spans between estimates, it is too easy for management to lose touch with the realities of software development and the direct effect that their actions can have on the process.

Iterations should produce a full product. All aspects of the production process should be exercised. There should be a working build, the build should be packaged, and that package should be deployed. It should go through all release tests, including load testing. If this has not happened, then there are surprises waiting for you. In places where I've worked, these have included

- A product that can't be deployed to production.

- An online product that only supports a few users on massive production hardware.

- A massive online system in which nobody asked the question, "How do we bill customers?"

All of these resulted in massive employee overtime and schedule slips. They were huge emergencies, and they reflected very poorly on the development organization. Had any of these projects used short iteration processes that moved from development through to production deployment, then these issues would have been caught.

Producing short iterations depends on automation. This automation chain runs from the developer's desk to the production facility. Human interaction should only be required at points where human judgment is required.

Manual processes should be avoided, because they are error prone. We're not built for doing the same thing over and over again. We get bored, and when we get bored we make mistakes. Mistakes take time and effort to find and fix. This causes unpredictability in estimates and scheduling.

Manual processes result in nonconformity. Each person invariably interprets instructions in a slightly different way, and every person makes mistakes. The result is that manual processes always introduce varying results. A netmask may be recorded incorrectly, the software may be copied to the wrong directory, or files might be named in a way that subtly invalidates chosen conventions. Automation must be able to cope with this variation, and this is a daunting task. Some might say that it is an impossible task. The amount of work involved in making the software flexible enough to handle these variations is often far larger than simply automating the manual processes.

People are slow. People are unpredictable. People are limited in the amount of work they can perform. People don't scale. Automation is how we get around these issues. Automation itself has the potential to go horribly wrong, though, and this is why it has to be exercised as part of the build process. The same automation should be used in development as is used in production. That way, the build process itself verifies the integrity of the automation.

Much of this book is about building that automation chain. It shows how your development environment can automatically run your tests. It shows how to build your changes automatically, how to package your application, and how to upgrade your database schemas repeatedly and reliably. All of these facilitate short iterations.

Continuous Reflection

Continuous reflection is the ongoing analysis of the development process. The development process is not something that is set in stone. At every opportunity it should be subjected to scrutiny and refined. There should be just enough process and no more. This reflection is often done at the end or beginning of an iteration.

Process exists to coordinate activities between people. It has benefits, and it has costs. Within an organization, it protects some parts from abuse by others, setting boundaries and responsibilities. It provides a framework for communication, and it's also the communication itself, allowing large numbers of people to work together in ways that they might otherwise not. At the grandest scale, there are international treaties and processes for dispute resolution that coordinate certain activities for billions of people. At the other end are simple rules for individuals greeting one another on the street.

Within a development organization, processes often set expectations between management and development. They define who is doing work, what sort of work they are doing, how that work will be performed, and when the work will be done. This gives a degree of predictability and transparency.

Process often exists to coordinate activities between groups. It lets QA know what development intends to deliver. It lets development know when QA needs the first release. It lets release engineering know when it should schedule a deployment test, and it lets system operations know when they should schedule the deployment. In these cases, it replaces clear communication and personal relationships between groups. With large groups this is necessary. With smaller groups it may not be.

It also gives each party a means of lowering their risks if something goes wrong. Even though a project may fail, the documents mandated by the process can be used to show that

expectations were met. When this happens, something is wrong with the organization. The process has become more important than actual accomplishments.

Too much process is maddening. People can see when formal processes impede work rather than facilitating it. They begin to resent the time they spend on the process, and it becomes a point of frustration. If the process takes away responsibilities, then they often feel powerless. They will feel even more powerless if they can't alter or bypass the process. There is little in the world that is as devastating to mental health as a high-stress environment in which people feel they have little control.

At best, the onerous process will be circumvented and become nothing more than a small waste of people's time. At worst, it will become a point of contention that will breed employee dissatisfaction and lead to turnover.

The only thing worse than too much process is no process at all. When no expectations are set, when no procedures are defined, or when anything goes if you ask nicely enough, the development process can run off the rails. People head in different directions. Individuals become points of control, and losing them can then be devastating to the organization.

Agile development processes try to find the sweet spot between these two extremes, and continuous reflection is the means. At the end of every iteration, the development teams look at their processes. If something is giving benefit, then it can be kept. If it has more cost than benefit, then it can be abandoned by the group. If there is not enough process, then the minimal amount of process necessary can be self-imposed. There is clear discussion so that everyone knows why the process exists and why it is retained. The team members are left with a feeling of control.

Agile teams feel they can get away with less process. The focus on automation reduces the number of people needed. That reduces the need for coordination. The frequent feedback from the short iterations increases visibility into the development process. It also produces more accurate estimates, increasing predictability. Pairing and collocation reduce the need for formal exchanges and meetings. Thus the various agile processes compensate for the reduced process load and, in doing so, actually accomplish some of the very goals that more formal processes strive to achieve.

Continuous Integration

Continuous integration is one of the most general practices. Code lives in the source repository. The longer your code is away from this repository, the further it will diverge from the repository version. When the code is merged back in, you will find bugs. The odds of any two changes conflicting go up nonlinearly, so the longer you wait, the more conflicts you will find. Resolving conflicts will become more painful, and more unit tests will have to be rewritten.

The solution is checking your code into the repository as frequently as possible. Every hour your code is out of the repository is another hour in which it can diverge from other developers' work. Every day your code is out of the repository means another day's salary that has been sunk into untracked changes. Should your machine crash, those changes will be lost, and that money and (more importantly) development time will also be lost.

These submissions should not break the build, nor should they break the application. Developers need to have a way to verify that they haven't broken the build. They need to do this in their development environment. The obvious choice is running the build and unit tests locally. Optimally, this should be the same build that is run for production, as every difference is a possible source of failure. The code should always compile, and all of the unit tests should succeed before code is checked into the source code repository.

There should be one source code repository for a project. All of the project's code should be checked in here. All artifacts necessary to produce the build should be included, too. This includes build scripts, tool scripts, properties files, installation scripts, third-party libraries, tests, and tool configurations (e.g., IDE configuration files).

The repository itself can enforce certain policy decisions. Submission triggers that validate code can be placed in the repository. If a file is not successfully validated, then the code submission will fail. Frequent validations include style analysis and syntactic analysis. With Python programs, style analysis checks frequently verify correct whitespacing. Syntactic analysis can be as simple as verifying that the file parses successfully. While the checks are being performed, submissions to the repository are blocked, so submission validation does not extend to significant functional checks.

Builds should happen automatically and as frequently as resources will allow. This is done to discover bugs as soon as possible. At the very least, builds should be run nightly, but with today's computing resources, there is very little reason not to run a build whenever new source code is added to the repository.

No human intervention should be necessary to go from source to finished and tested product. The build system should check out a clean copy from revision control, and then build from it. This ensures that the build does not depend on previously generated artifacts, and it tests that the build can be done on a machine other than the developer's desktop. This also goes a long way to ensuring that any developer can sync the code tree down to a new machine, issue a single build command, and have the build succeed. This allows desktops to be replaced quickly in case of failure, and it helps new developers on a project to come up to speed quickly.

The build should test all components, construction of the database, and initialization of any external services. The build must run all of the unit tests. Any unit test failures should cause the build to fail. I'll argue that a minimal set of functional tests should be run, but these may be restricted to certain kinds of builds. Often these are referred to as official builds.

Build failures should be quickly communicated to the team so that they can be quickly fixed. Typically, this entire process will be done on dedicated build machines, so a remote notification system should be used. This might be mail, chat, or some external means of notification (such as a lava lamp or a siren).

Much of this book focuses on continuous integration in Python. The package Setuptools forms the core of a replicable build system in Python. It provides dependency management, building, packaging, and a test harness. The testing package Nose will actually run the unit tests. The unit tests will run both from the build and from within the Eclipse IDE. A set of custom scripts will handle database management. The repository we'll be using is called Subversion. We'll validate Python source with Subversion triggers. Finally, we'll set up a build server using Buildbot.

Documentation

Documentation should be minimal. It should be limited to that which is necessary to ensure that participants in the development process can communicate. You should probably keep your system metaphor available somewhere. You should have the documentation necessary so that your system can be used and maintained.

You do not need extensive design documentation. The code is the design. You don't need extensive commenting. The code and unit tests should be clear and readable. Extensive design

documentation is the pinnacle violation of DRY. The impression of the design invariably falls out of sync with the reality of the design.

These practices have a common focus. They are about predictability. They are about minimizing waste, both in process and in design effort, and continuous feedback to identify the sources of waste.

While it's important to understand how these techniques fit together, we're not going to be looking at all of them. Our focus is going to be those agile techniques that are abetted by supporting software. These areas are continuous integration, TDD, unit tests, refactoring, and simple design. None of the products we're looking at are commercial. Like Python, they're open source of one sort or another. These products (and the corresponding chapters they're covered in) are as follows:

- Eclipse (Chapter 2), an IDE

- Pydev (Chapter 2), a Python Eclipse component

- Subversion (Chapter 3), a revision control system

- Setuptools (Chapter 4), a build harness

- Buildbot (Chapter 5), a continuous build system

- Nose (Chapter 6), a test runner

- pMock (Chapter 7), a mock object framework

- PyMock (Chapter 7), another mock object framework

- SQLObject (Chapter 9), an object-relational mapper

- SQLAlchemy (Chapter 9), another object-relational mapper

- JSUnit (Chapter 10), a JavaScript unit testing tool

- PyFit (Chapter 11), a functional testing tool

Summary

In this chapter, I have introduced the methods that constitute much of agile development. You've seen how they tie together and how they assist each other. Some of these methods relate to process, but many have more to do with concrete programming practices. Some are strictly developer tasks, and others are often seen as part of release engineering. By now, you probably realize that many agile methods aren't so alien. If you're a professional developer, you've probably used several of them, quite possibly automated builds and unit tests.

Much of what you will learn here can be taken back to work. If you decide to do that, then be careful. The word *agile* scares some people. Even though the techniques are hardly earth shattering in their novelty, the term has become a four-letter word in some places; it turns people off immediately.

If you're a developer, and you decide to bring an agile method into work, then treat it like a strange and somewhat skittish pet. Bring it into work, show it around your desk, let it become comfortable with its immediate surroundings, and let it interact with the local

environment. If it seems to be happy with you and your code base, then introduce it to some of your more inquisitive coworkers. Let them play with your friend. When it becomes at home and comfortable, see if it expands its home range on its own. Once it's comfortably ensconced in the new home, then bring in one of its friends in a similar way.

If you're in management, it is often best to lead your team gently into the new practices. While many find the prospect exciting, some will be skeptical. Saying, "We're going agile now" is a good way to lose face. Find a developer or team that is interested in trying out the techniques. Choose a practice with a low barrier to entry and a high payback for the effort, and get the developer or group to try it out as a pilot. There will be teething problems, so don't talk it up until the practice has proven itself. Changes are always most successful if the developers believe they instituted the changes themselves. If the practice succeeds, make sure that the developers involved get the credit. If it fails, take the responsibility yourself.

I use the command line for much of the material in this book but, whenever possible, I try to present the same material within an IDE. too. You can certainly do agile development without an IDE, but there are some tasks that are far more difficult. One that springs immediately to mind is refactoring.

In the next chapter, I'll show you the Eclipse IDE and how to set it up with the tools that we'll be using throughout the rest of this book.