CHAPTER 8

Everybody Needs Feedback

Developers want to write good code. However, their code tends to be worse than they believe it is. They think their tests cover more cases than they really do, and they believe that more of the code is exercised than really is. They tend to believe that they understand the code better than they really do, and they believe they produce fewer bugs than they do.

This is because programmers are by and large healthy optimists. They have to be. Truly understanding all the details in even a simple program requires years of study and experience, so they gloss over most of the details.[1]

This isn't some kind of strange, aberrant behavior. Over and over again, psychological research has shown that normal, happy people believe that they have more control than they really do. Depressed people seem to have an absolutely accurate view of the control they have over situations. Does that make them better programmers? Probably not. The depressed tend to be less creative, and they have a really hard time motivating themselves. On balance, it's better to be a healthy and functional human being, even if it leads to objectively unjustified optimism.

An experienced programmer does, however, tend toward cynicism. Experience hopefully brings an understanding of one's faults and shortcomings. It's not necessary to conquer your faults, but it is necessary to see them and work with them. The first step is getting the feedback to understand what those flaws are. A story illustrates this.

Years ago I had a dear girlfriend. She was one of the most brilliant people I've ever met. She was nearing the end of her doctorate in computational molecular virology. (After that, she headed off to veterinary school.)

There was a problem in her lab. Biochemists label things left and right—centrifuge tubes, test tubes, beakers, Eppendorf tubes, and so on. Getting anything done in a lab requires a Sharpie—a kind of indelible magic marker. And someone in her lab was stealing all the Sharpies. Whenever she needed one, she'd have to go questing for markers, stalling her lab work, and derailing her train of thought.

She railed endlessly to anyone who would listen about the inconsiderate thief who was stealing all the lab's Sharpies. She loved venting about it. She couldn't figure out who it was either, which made it all the more mysterious.

---

1. One of my favorite interview questions for system administrators is "Describe what happens when you type `telnet www.google.com`." No matter how deep someone goes, you can always ask more detailed questions.

One evening she opened up a desk drawer in her bedroom, and she tossed a pen in. Unlike most days, she looked down at the drawer. She broke down laughing, and she brought me over to look at it. It was full of Sharpies. Tens upon tens, perhaps hundreds of them.

She was the Sharpie thief. Every day she left the lab with two or three Sharpies in her pants. Every day she returned home, mindlessly opened the drawer, tossed the Sharpies in, and closed it. (Her organizational instincts were incredible.) She never noticed doing it. With this discovery, the Sharpies stopped vanishing from the lab.

The moral of this story is that feedback is incredibly important. Without the appropriate feedback, she might never have realized that she was the source of the problem in the lab. Without feedback, you often can't see your own faults.

What do developers need feedback for? Well, developers have their own drawers of Sharpies. Each person has errors they tend to commit. I double space after periods, and I have to go back through my documents pulling them out. My editor appreciates that. I also have a tendency to write overly complicated and general code. I have to strive for simplicity. I have trouble choosing appropriate names, and my comments often lack enough depth. I tend to be either too pedantic or not pedantic enough. Sometimes I use tabs by reflex, and my lines tend to be way over 80 characters long. I tend to miss simple error checks, and I like mock objects too much. I have to keep an eye out for these things. It's good to have tools, procedures, and an environment that help to prevent these from happening.

This chapter looks at several measures of quality. Some are quantitative and some are qualitative. Among the qualitative measures are coding standards.

Fundamentally, there are two kinds of feedback for development. They are social and environmental. *Social feedback* includes structured criticism through procedures such as code reviews, and it includes cultural norms such as interpersonal communication patterns and documentation habits. Rewards are also a kind of explicit social feedback, and I'll talk a little bit about them.

Environmental feedback encompasses technological gadgetry. Your project's tooling should give you feedback where social feedback fails. It can produce precise, focused, and immediate feedback on small things:

- IDEs and compilers let you know when code is syntactically broken.
- The source code repository can check for malformed code and refuse to accept submissions.
- The build system can fail the build when conditions aren't met, which I've already demonstrated in connection with unit tests in previous chapters.

This is all very important because it affects software quality. Software quality is about keeping errors down while making the remaining errors easy to find. Put another way, it is about making software that is easy to maintain without introducing new errors.

There has been a great deal of research into the kinds of errors that developers make. Different studies report different results, and it's hard to come to a firm consensus. Some consider the hard numbers produced in this area to be highly suspect. Much effort has been focused on classifying bugs and their relative frequencies, and some general themes have been revealed.

The scope of most errors appears to be limited. Many are outside the domain of construction. Most are the programmer's fault, and a lot of those are typos and misspellings.

A recurrent theme is failure to understand the problem domain and the design of the software itself. Happily, most errors seem to be easy to fix.

One plausible reason for the difference in quantifiable results between studies is that different environments, both social and technological, lead to different errors. The individuals in the mix probably contribute, too, so it is important to build on your organization's experience. I suspect that collecting per-user and per-group information to build targeted defect profiles is an area that is ripe for research and/or commercialization.

There are some practices that make errors easy to find. The first of these is an extensive suite of tests, which I've already discussed in previous chapters. Tests provide feedback, but there is further feedback about the quality of those tests, which is explored here.

Simple design, a core agile practice, focuses on building only the minimal functionality that allows the program to meet the user's needs. There are measures that successfully capture and quantify various aspects of a program's complexity.

Writing clear code helps to pinpoint errors. Clear code is written with the intention that it will be read.[2] It focuses on communicating intent to the user, with the computer as a secondary concern. Various tools assist in writing clear code. They check conformance with coding standards and consistency of style.

Stylistic consistency is one of the hallmarks of easily read code. In such code, names are chosen well, and they are chosen in a way that reflects the underlying system metaphor. Those names and the choices they embody are propagated throughout the code base. Typographical conventions are the same throughout, blocks are indented the same way in the same situations, spaces are added or omitted in the same manner, and so on. These choices are made in a way that is both simple and self-consistent.

While tools can help with some aspects of these practices, human eyeballs and procedural or cultural practices are often the best ways of helping to achieve these goals. The problem with tools is determining which aspects of these practices can be measured.

Measuring Software Quality

Measurements give you feedback. Quantitative measures give you precise numbers characterizing an attribute, while qualitative measures describe the general properties of the subject you're studying. They tell you what you have, but not how much of it.

Quantitative measures are appealing in that they can often be automated. They tell you a precise value of a specific attribute, but their specificity limits their utility. The results can be rendered graphically, making them favorites for management. (There are some people who fall in love with anything that you put in a spreadsheet.) They invite abuse at times, and in the wrong hands they render discussion moot, even if there is a point to be discussed.

Qualitative measures are much fuzzier, but they can often lead to greater insight. They are judgments such as "the code stinks," or "the style is awful." They constrain the mind less, and their contemplation often leads to ideas for quantitative measures. Qualitative measures don't lend themselves to automation, so those with a penchant for automation often give them short shrift.

---

2. As Tom Welsh (my editor) said, "Indeed, the more successful your code, the more times it will be read—and by more people."

Measurements

With any measure, the first question is "What are we trying to measure?" There are several factors characterizing the measure:

There are *attributes* and *instruments*. The underlying phenomena may be characterized by attributes that can be measured. Those attributes are determined, and the instruments of measurement are decided upon.

The instruments' results must be *reported*. A means of storage and presentation must be decided upon. They may be dumped into a database and analyzed, or they may be spit to `sys.stderr`. The means of presentation doesn't have to be fancy, it just has to be effective.

Often you measure to *effect change*. Do the chosen measurements provide effective feedback? Do the measurements of code complexity result in less complex code? Does a measure of test coverage result in better test coverage? Does it tell you where the poor-quality code resides?

Measurements often have *side effects*. Are your programmers now competing to see who can get the highest cyclomatic complexity number? Are programmers just adding tests to increase coverage instead of really testing the code? Will this cause the measure to lose its effectiveness at identifying poor-quality code?

Before you begin measuring, there are some fundamental questions for which the answers need to be understood:

What is the purpose of the measurement? What are you trying to accomplish? Is this for your own use, or is it intended to change the way everyone codes? If the measurement is for your own use, then the variance may be high, and the technique doesn't need much justification; you can be sloppy. If the measurement is intended to change the way everyone codes, then you need to choose a well-understood measure, and you need to do it in a consistent manner, as you'll need to justify your choices.

What is the scope? How widely will this measurement be used? The wider it is applied, the more impact it may have, both through positive control effects and unintended side effects.

What attributes are being measured? Imprecise ideas about what is being measured are likely to yield imprecise results.

What are the units? Unless you understand the units, you can't determine how it relates to other quantifiable values. Measuring an amoeba in feet is nearly useless. Measuring an elephant in angstroms is meaningless (although it does bring up the interesting question of where the elephant begins and ends).

What is the variability of the attribute? Unless you understand the variability of a measure, you can't determine how accurate your measurement is.

What is the measuring instrument? Don't use a micrometer to measure an elephant. Don't use a yardstick to measure an amoeba. Don't use line counts to measure program complexity. (Don't use a bathroom scale with an elephant either. It breaks.)

What are the units of the instrument? This ties in with the previous question. The units of the instrument must be compatible with the units of the attribute.

What is the reading's variability? Most instruments are imperfect. They have errors. Network problems cause sampling problems with remote probes. Statistical profilers can only give approximate usage reports.

How do the measurements and attributes interact? Retrieving page counts from a web server by making an HTTP connection increases the number of hits. For small, low-traffic web sites, this could be a problem. Measuring code coverage through execution affects how quickly tests run. Timing tests might exhibit failures while coverage is being examined.

What are the foreseeable side effects? Are the page hits artificially inflated? Are the timing tests dying mysteriously? Will reporting cyclomatic complexity result in an obfuscated code competition? Will reporting code coverage cause code coverage to improve? Will you be fired for stepping on your boss's turf?

Quantitative Measurements: How Much Is That Doggie in the Window?

There are common quantitative measures to which you've probably been exposed. These include the following:

- Test coverage
- Source lines of code (SLOC)
- Cyclomatic complexity
- Churn
- Recorded defect rates
- Development velocity

We'll be looking at three of these in detail: coverage, cyclomatic complexity, and development velocity.

Code Coverage

Code coverage is a family of measurements. There are many different kinds of code coverage. Cem Kaner covers 101 of them in his paper "Software Negligence and Testing Coverage" (`www.kaner.com/coverage.htm`). With 100 percent statement coverage, all statements in a program have been executed. This is not to say that all expressions have been executed. It is also not the same as saying that all branches have been executed. An if-then-else statement has been executed even if only the else block has been traversed.

With 100 percent branch coverage, every branch of every statement has been executed. In an if-then-else statement, both the then block and the else block have been traversed. Branch coverage is a much stronger metric, but it still doesn't guarantee that all expressions have been evaluated. In some definitions, a short-circuit logical operator is considered to have been executed even if the second operand has never been evaluated.

Branch coverage is appealing in several ways. It is easy to count. In many languages, the tracing mechanism can be used to obtain this number. It is unambiguous. When you say that 70 percent of statements have been executed, little further explanation is needed. The ease with which it is explained is part of its appeal. Anyone can grasp it in a moment.

Those factors make branch coverage seductive. There is a temptation to see it as a goal, but it is not. It is simply a tool, and like any other tool it has limitations.

Branch coverage tells you nothing about data flow. It doesn't tell you that a variable has never been initialized, that a constant is returned instead of the value your code spent hours calculating, or that an invariant value is being rewritten every time a loop is entered.

One hundred percent branch coverage only covers those branches that have been written, so it doesn't cover sins of omission. Necessary, but unwritten, code is invisible to this metric. According to one survey study, these kinds of errors account for between 22 and 54 percent of all bugs.[3]

Weak tests may hit all statements, but they don't hit them very hard. The tests don't exercise every predicate in the conditional clauses. Loops are only executed once, and many bugs don't occur until they're executed several times. Default values are modified, and the new values leak into subsequent calls, but the test framework clears them every time.

Mock objects short-circuit interactions between methods. In Python, they allow complete isolation, so it's possible that the real function is never called. Although the statement has been executed, it hasn't been executed with real data.

Branch coverage doesn't report errors that take a while to manifest. It doesn't catch environmental interactions. Table-driven code is inscrutable to branch coverage tools, which miss all of the embedded logic. They miss any place where work is done in data instead of code, and branch coverage completely misses the interactions between interrupts and signal handlers.

With all these problems, why use branch coverage? Because it yields useful information; but you have to be aware of that information's limitations. If you have low test coverage, then you probably have a problem. You should look at where the test coverage is missing, and then decide if it should be addressed. If it's old code that's well debugged and rarely changes, then it's probably not worth focusing efforts there. If it's in highly defective code, or code with a high churn, then it might be worth focusing testing efforts there.

What constitutes low test coverage? Below 85 percent is a number that's bandied about, but there seems to be little academic basis for it. It may be a number that someone picked out of a hat at some point and has been referenced ever since, like an urban legend.

What is the branch statement coverage of your code? Unless you're measuring it, you're probably overestimating it. Typically, unit tests only cover 50 to 60 percent of the code in a system. You can probably look at my code and get a feel for the coverage, but when you look at

3. Brian Marick, "Faults of Omission," *Software Testing and Quality Engineering Magazine* (January, 2000), `www.testing.com/writings/omissions.pdf`.

your own code your estimate will be too high. People tend to have a blind spot when it comes to their own weaknesses. There is likely to be a moment of shock the first time you wire up one of the tools described later in this chapter.

Some have an aversion to measurement. There's not really an excuse for this. Refusing to measure when you have the tools available means that you are willfully ignorant, but there are reasons to tread carefully. Measurement has motivational effects, and these effects can be good or bad. People tend to optimize for anything that they are being judged by. People like to look busy and productive, so measure and report carefully.

Complexity Measurements

As with code coverage measurements, there are many different kinds of complexity measurements. The example we'll be wiring up later is called cyclomatic complexity, or McCabe complexity, developed by Thomas McCabe in 1976. The measure was almost a side effect of the paper's larger achievement of defining what an "unstructured program" is. It determines complexity on a per-function or per-member basis by examining a program's control flow graph.

A control flow graph pictorially represents how execution passes through a program. In these diagrams, statements that don't affect execution paths are ignored. Only those statements that entail decision points are included. The flow graph for the following program is shown in Figure 8-1.

```
def foo(x):
    while x > 5:
        if x < 2:
           print "a"
        else:
           print "b"
```

The cyclomatic complexity algorithm adds a link from the end of the program to the beginning. This is shown as a dotted line in Figure 8-1.

In hard mathematical terms, cyclomatic complexity is the smallest number of linearly independent paths that it takes to span the flow graph. A set of partial paths span a graph when every possible path through the graph can be described using a combination of these partial paths. A set of paths that span this graph is shown in Figure 8-2. Every path through this graph can be described by combining these five graphs.

The next part of the definition is *linear independence*. If you've had a linear algebra class, this should be familiar. The paths through the graph are linearly independent if there is no way to combine all of them at the same time in such a way that they cancel each other out.

Imagine that you're dropping breadcrumbs as you walk one of the partial graphs. Then you try walking one of the other paths that connect to this one, and you continue doing this until you've walked through all of the partial paths. If you can pick up all the breadcrumbs you dropped, then your paths were not linearly independent.

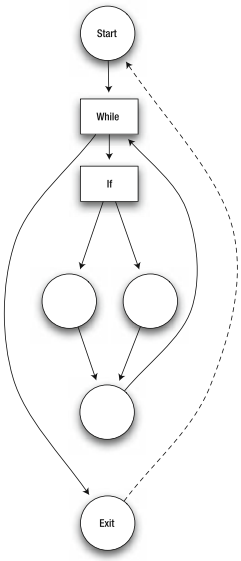

Figure 8-1. *The control flow for an if-then-else statement inside a while loop*

In Figure 8-2, the three paths on the right (3, 4, and 5) are not linearly independent. You can walk path 3 and then path 4 dropping breadcrumbs along the way, and you'll end up at the beginning. Then you can follow path 5 backward, picking up the breadcrumbs as you go, and you'll end up at the beginning having collected all the crumbs.

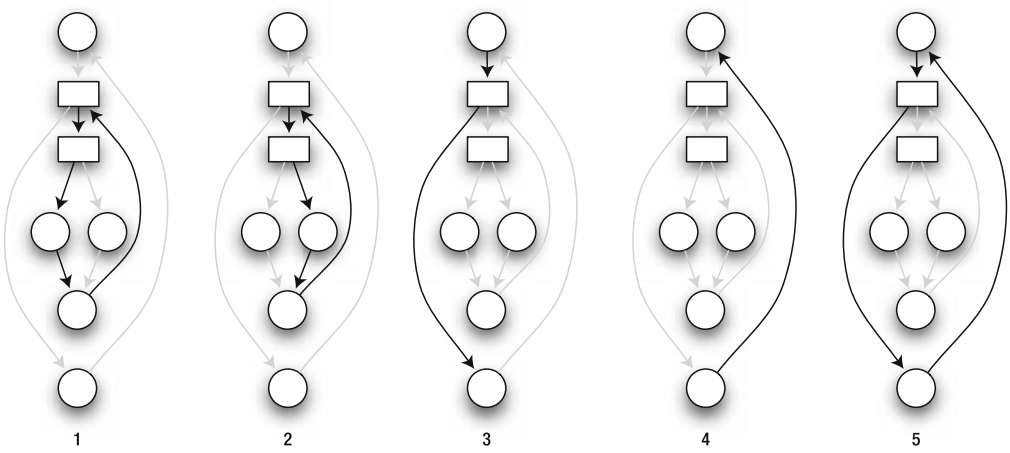

Figure 8-2. *A spanning set that is not linearly independent*

Figure 8-3 shows a linearly independent set of partial paths that span the graph. This is not the only one possible, but three is the smallest possible number for this graph. No set of two partial paths can be combined to describe all possible paths, and no complete set of four

or more partial paths is linearly independent if it spans the graph. Every closed directed graph can be characterized like this.

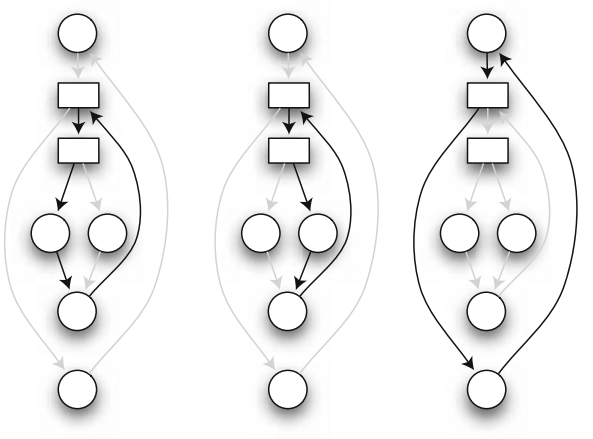

Figure 8-3. *A linearly independent minimal spanning set*

So that's a description of what cyclomatic complexity really means. It's how many different control flows are in a section of code. You could calculate it by drawing graphs and figuring out the spanning sets, but there's a much easier shorthand:

```
C = 1 + number of decisions
```

There is always at least one straight line through any graph, so cyclomatic code complexity starts with one. Each decision point adds another possible path. Some constructs add more than one. Every time a construct is encountered, the complexity index is increased by a specified amount. A simple calculation might use Table 8-1.

Table 8-1. *Sample Scores for Use in Calculating Cylcomatic Code Complexity*

Construct	Effect	Reason
If	+1	A decision is made.
Each elif	+1	A decision is made.
If-else	+1	A decision is made. (A plain If statement has an implicit, but empty else clause.)
While	+1	A decision is made.
For	+1	A decision is made.
Try	+1	Exceptions generate a new flow of control.
First except	0	The first is already accounted for by the try block.
Subsequence excepts	+1	A new choice is added.
Finally	0	All paths just rejoin here.
With	0	No control flow is visible to the routine.
Decorators	0	There is no alteration of the program flow.

There is more than one way to derive the cyclomatic complexity for a routine. The differences are based primarily on the control flow graph that is generated. Each logical comparison in an if-then or while statement can be viewed as generating an alternate condition. This can potentially result in much higher cyclomatic complexity numbers.

Generally, the lower the cyclomatic complexity, the better. Values in the range of 1 to 5 are considered to be trivial. Values from 6 to 10 are considered to be low risk. Numbers between 11 and 20 signify moderate risk. Numbers between 21 and 50 are considered high risk. At 50, you should consider submitting your routine to an obfuscated code contest rather than your source repository. The word *untestable* is often used in this context. These cutoffs are somewhat arbitrary, and as far as I know, there is no basis for their use other than experience and informed opinion.

Velocity: When Are We Done?

Velocity is a quantitative metric describing how much work a group can accomplish in a given time. Velocity is the primary measure for capacity estimation in most agile methodologies. It is most frequently used in development environments with well-defined iterations.

At the beginning of each iteration, the tasks available are placed on a board. Together, all the developers assign an effort estimate to each task. The estimates come from a small set of possible choices that correspond to point values. At the end of the iteration, the team sees how many points of work they've completed. The team's velocity is the number of points completed divided by the number of days worked.[4]

The first time through, the team is flying blind. They can make the estimates, but they can't convert those work estimates to time estimates. Velocity provides this conversion.

In successive iterations, the previous velocity measurements are combined to produce an average velocity, and this value should become more accurate over time. As the accuracy improves, the team can use this number to reschedule development or drop features as appropriate.

There are different methods for assigning estimates. Some use raw points, and some map between a natural language scale and points (e.g., small, medium, large, and that's-too-big-to-estimate). No matter what the details, they all use a small set of values, often no more than four or five.

Next, I'll describe two scales I've had direct experience with. One scale uses a raw point range of 0 to 3. A 0-point job is trivial. A 3-point job should probably be broken into smaller pieces. The numbers in this scale are not linear—a 2-point job takes much more than twice as much effort as a 1-point job, and 1-point job takes much more effort than a 0-point job.

Another scale uses the sizes extra small, small, medium, large, and epic. At one end of the spectrum are extra-small tasks, which are trivial, and at the other end are epic tasks, which are inestimable and need to be broken down into more manageable chunks. The rationale for using sizes rather than point values is that sizes can be mapped to a nonlinear scale, so that small might be 1 point and large might be 8.

4. Here the units are points per day. Always know your units.

The scrum methodology uses direct time estimates in hours, in which no task takes more than a day. All of these can be interchanged to some degree.

These values are purposefully fuzzy. Each group's definition will be a little different. What matters is that the team is consistent. Over time, the velocity calculations—whether in points, effort, or hours—become more accurate. The team works in small enough increments that daily stand-up meetings and periodic sprint retrospectives give them timely feedback, and this allows for improvements in estimation.

Qualitative Measurements: It's a Shih Tzu!

We are capricious beasts, and we are rarely as rational as we'd like to believe. Often, qualitative measures are the things that truly matter to us. I can have the best job in the world, but something goes wrong. My manager changes, and nothing else really changes about my job. The new guy is personable, in fact downright likable. By any measure of wage or work hours, or a listing of responsibilities, my job has remained the same, but suddenly I hate it. Getting to work is a chore. I'm constantly stressed. My ability to complete work declines.

Something has changed, but I can't say what. I can't measure the cause, but it's real and it matters. There has been a qualitative change, and its ruining my job.

IS THAT REALLY A MEASUREMENT?

Somewhere along the line, I was asked if qualitative judgment could really be called a measurement. I have a background in biochemistry, and I spent a small chunk of my life in a lab. Lab notebooks were full of qualitative measurements like this:

Tube A: Clear
Tube B: Cloudy
Tube C: Kind of murky
Tube D: Completely opaque

Each one of those is a measurement. You're determining some kind of data and recording it. You can describe code similarly:

Function A: Terrible code
Function B: Not too bad
Function C: Pretty good
Function D: Obviously Noah Friedman [5]

So I think it's fair to say that readability, continuity, and elegance are measurements, even if they don't have an obvious numerical representation. It's the systematic recording that makes something a measurement.

5. Noah Friedman wrote large chunks of Emacs. People who read his code have been known to laugh out loud with pleasure. My programming skill is measured in millifriedmans.

In the same way, there are many qualitative measures related to software. These are the things that you feel with your gut. These are the elements of judgment. When you read a program listing and you smile at the cleverness and clarity of the bounds checking, that's a qualitative judgment. In the same way, the sudden feeling of revulsion when you look at Bugzilla's code is also a qualitative judgment. These judgments are the measures of appropriateness of naming, agreement among those names, and elegance of control flow.

Qualitative changes often have quantitative effects. Poor architecture leads to a lower velocity, as do many other flawed development aspects. One of these is readability. Code that is hard to read is hard to modify. Code that is inconsistent is harder to read. The need for consistency leads to coding conventions.

Coding Conventions

There are three primary aspects to coding conventions. The typographical standards dictate the code's appearance. How many spaces is each block indented? Do spaces bracket the equal sign in an assignment? How about in a keyword assignment?

Naming conventions determine how names for variables, classes, and methods are chosen. They provide a common grammar and map the system metaphor into the programming language.

Structural conventions determine how a project is laid out. They determine where data and documentation can be found, and where the code and tests live. They provide a structure so that both developers and tools can examine the code base.

Coding conventions supply your project with a common language. They allow you to take more for granted. You don't have to decide where a file will go. You only have to decide what kind of a file it is. The convention supplies the location.

Coding conventions help to transfer knowledge across projects. When multiple projects share the same conventions, the developers can use each other's knowledge. They know where the unit tests are. They know that `ViewInterface` names a class rather than a method, and that `view_interface` defines a method or variable.

This consistency allows developers to learn new code more quickly.

Naming standards reduce name proliferation. A `linear_transformation` could reasonably be called a `linear_matrix`, a `linear_transform`, a `transform`, a `scaling_matrix`, or a `rotation`. Choosing one leaves less to remember.

Naming standards can compensate for language weaknesses. Unlike Java, Python doesn't provide interfaces as a language feature, but they can be simulated using various techniques. The standard can declare that classes used as interfaces shall be given names like `FooInterface`. Any time a developer sees `BarInterface`, they will know how it is being used. This lets developers bring along a useful feature from one language to another without formal support in the new language.

Naming standards can also emphasize the relationships between items. Declaring that collections must have plural names, and that each element in a collection must be referred to by the singular of that collection's name, instantly gives the user reading a loop a clear idea of the variables' relationships.

Almost any convention is better than none. Convention can be understood as a gradient, and the more that is specified, the less has to be deduced. Conversely, the less that is specified, the more must be deduced, and each deduction takes time and mental energy. Any reduction in variety will help.

The principle behind all this is that *structure is good*. Some argue that choice is a good thing, and it is, but only up to a point. Beyond that point, people are overwhelmed by choice. Too many options at a store actually decrease the chances that a customer will buy something. Our minds can't cope with any more information. It just becomes mental noise. Coding conventions reduce choice through clear guidelines.

This book demonstrates the utility of conventions—typographical, naming, and structural. When you see the font and location, you instantly know that this is the main narrative. When you see a double-lined section, you know it's a warning. When you see a sidebar, you know that it might be interesting, but that it's not essential. Imagine what this book would be like without consistent conventions.

Consistency is a good thing, but some consistencies are more important than others. Consistency with the world in general is good, but consistency within a project is more important, and consistency within one package is more important still. In turn, consistency within a module is more important than within a package, consistency within a class is more important than within the module, and consistency within a single function or method is paramount.

In general, when it comes to coding standards, questions are important. If you have questions, ask your partner. If she's not available, ask your teammates. Open and instant lines of communication are valuable. Physical proximity is always best, but often not possible. IRC or other shared-channel chat protocols are a distant second. Person-to-person chat is less useful. Hiking across the office is bad, and the phone is even worse. Sometimes, though, it may be the only choice.

There are very practical benefits conferred by consistent code conventions. When applied, they reduce churn due to reformatting, and diffs between file revisions become smaller, saving effort when integrating.

There are a variety of options for specifying coding conventions. At one end are flexible guidelines that are suited for very small groups or very large groups with diverse personalities. These may be little more than a collection of suggestions. At the other end are extremely detailed documents describing every *t* and *i* to be crossed and dotted. In either case, these are often best implemented as (or in conjunction with) a set of examples demonstrating the desired standards.

Coding conventions are desirable when the code is maintained or extended. In practical terms, this implies that anything other than disposable code for your own use should adhere to a well-defined coding convention. Certainly all agile projects fall into this category. Pair programming automatically involves review, and a well-defined system metaphor lays the foundations of a naming convention that should be extended through the entire code base.

The definition of good code varies from person to person, but a group needs a shared definition. Where should this shared definition come from? Answering this question requires some consideration of how software projects work.

Software projects depend on two kinds of authority: organizational authority and technical authority. The two are independent. *Organizational authority* comes from the structure of the company. *Technical authority* comes from experience and intelligence. Technical decisions need to be made with respect to technical authority.

Programmers often tend to despise management. Management, in turn, often disregards the programmer's expertise. But both groups contribute to the larger goals of the company, and each needs the other's know-how. Only when they respect each other and listen to each other can a project reach its maximum potential.

In all cases, managers should be involved with development to clearly communicate the company's goals. The agile practice of identifying a well-defined customer addresses part of this. The customer is brought into the development cycle and works closely with the developers. Short iterations with a predictable workload provide management with the feedback they need to understand development's progress.

So where do the coding conventions come from? If a coding style declaration comes from nontechnical management on the basis of organizational authority, then it will be rejected, so it must come from a position of technical authority within the development organization.

Several things make a coding standard work. First and foremost, the coding standard must be disseminated and understood. Clear examples of the coding standard must be available.

The code must be considered to be a shared asset, and everyone should expect their code to be read. Moreover, this expectation should be true. Every line should be seen by another pair of eyeballs.

All code should be seen by multiple people. There are aspects of a coding standard that can be verified by machine, all relating to typographical structure. The true meat must be verified by people, and that means people who weren't the sole author. Such outsiders don't have the same blind spot for the author's foibles.

When the code is submitted, it should meet quality guidelines. Some argue that the code should be signed off by those with more technical authority. There is a range of views as to how this should be done. At one end are those that suspect the programmer's competence and argue that all submissions must be approved. At another point along this continuum are organizations in which only one sizable chunk of code must be submitted for approval. Once the programmer has passed the "this is how we program here" test, they are allowed to submit code on their own. At the far end of the scale are those who implicitly trust those whom they hire without verification, which is a situation I have personally seen lead to bad code.

It has been suggested that if a project has a technical manager, then this manager should read all code. If the manager can't read the code, then it needs to be rewritten. In such cases, it is a boon to the project if the manager isn't a hotshot programmer, and the authors have to write down to his level.

There are different degrees of formality for coding standards. The degree of formality is directly related to the number of people on a project. The standard should be as informal as possible. Established projects should use the guidelines that they already have. If they don't have any, then guidelines should be established. Those guidelines should match the code base's existing style whenever possible.

Occasionally you'll have to break style. This is done primarily when the rules make code less readable, even for someone who is knowledgeable about the rules. This should be a rare occasion. The other reason is to be consistent with extant code that already breaks the rules, although in these cases, fixing the offending code is often a better solution. The longer it incubates, the further the stylistic infection is likely to spread through your code base's healthy tissues.

Welcome Back to Python

Unlike many other languages, Python has a natural authority for coding standards. This is Guido van Rossum, Python's creator. He is sometimes referred to as the Benevolent Dictator for Life (BDFL).

His wisdom is encapsulated in several Python Enhancement Proposals (PEPs). These conventions were laid out early in Python's development, so they're reasonably well disseminated. There are five documents related to coding conventions:

- PEP 7: Style Guide for C Code
- PEP 8: Style Guide for Python Code
- PEP 20: The Zen of Python
- PEP 257: Docstring Conventions
- PEP 287: reStructuredText Docstring Format

The first (PEP 7) relates only to C code, so I'm ignoring it in these discussions. The last two (PEPs 257 and 287) relate to docstring formatting. reStructuredText (RST) is a simple markup language similar to that used by many wikis. It is used for complicated docstrings or when the code is intended to be self-documenting.

PEPs 8 and 20 are the real meat. PEP 20 explains the philosophy behind many of the decisions made in the evolution of the language, and it helps in understanding why things are the way they are. It's written by Tim Peters, but he's channeling the BDFL. It's the best starting place. In fact, PEP 20 is embedded within the Python interpreter:

```
>>> import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea—let's do more of those!
```

This contrasts strongly with other languages that offer many ways of performing the same task. Here there's only one way to do it.

PEP 8 is the global coding standard. All groups should attempt to adhere to it. Python is refreshing (some might say maddening) in its approach to indenting. In most languages, the placement of block delimiters results in a variety of gross layout decisions. Since Python uses leading whitespace alone, there are no block delimiters to inspire theological debate.

PEP 8's summation of block formatting boils down to this: each level is indented four spaces—don't use tabs. The document is well written, and it is available at `www.python.org/dev/peps/pep-0008/`.

Naming is the area where Python is least consistent. The styles advocated in PEP 8 are shown in Table 8-2. For the most part, new packages adhere to these. Unfortunately, the recommendations were taken up long after a variety of naming styles had infiltrated the core libraries. There are now aberrations like the string buffer I/O module `StringIO.StringIO`. Attribute names using camel case with an initial lower capital (à la Java) are often seen, but this style is advised against. Still, it's hard to escape this style because several notable packages, among them SQLObject, use it.

PEP 8 only addresses the most basic aspects of writing docstrings. PEP 257 fills in the details, and it's well worth the read. It explains conventions for formatting docstrings. PEP 258 goes even further. It defines a simple and attractive markup language for use in Python docstrings and documentation.

Table 8-2. *Python Typographical Naming Conventions*

Entity	Name Convention	Example
Package	Short lowercase underscore separated	`my_package`
Class	Camel case	`MyClass`
Attribute	Lowercase underscore separated (LCUS)	`my_var`
Function	LCUS	`my_func`
Private class	Camel case with leading underscore	`_MyPrivateClass`
Private attribute	LCUS with leading underscore	`_my_private_var`
Private function	LCUS with leading underscore	`_my_private_func`
Really private attribute	LCUS with two leading underscores	`__my_var`

All of these standards address the typographical aspects of coding conventions, but they don't deal with the semantic aspects of naming. To some extent, I've tried to provide a basis for structural decisions in earlier chapters, but when it comes to the semantic aspects, you'll have to develop your own. The particulars of your naming scheme will depend heavily upon the system metaphor you develop in your project.

Never Try to Fix a Social Problem with a Technical Solution

Technical fixes to social issues are usually failures. They frustrate people. The people imposing the technical limitations pay for the failure, and they usually pay in the currency of respect.

Before people can comply with a standard, it must be available. A secret standard is useless, so it must be published, and it should be published in a well-known location. Often a

code catalog containing snippets is better than an English document. A catalog consisting of real code from the project is even better. New samples can be taken from the code base as part of a motivational system.

Rewarding good code is a mixed idea. It opens up the entire topic of rewards and penalties. Rewards are a form of measure, and like all measures they must be used carefully, for they are particularly prone to unintended side effects.

The goal of any reward should be increasing the productivity of the entire team. A reward that emphasizes individual achievement but undermines the team is worse than no reward at all. If individuals are rewarded, they should be rewarded for contributing to the achievement of group goals, and in such a way that anyone can achieve the reward by doing their job well. Choose goals that you want to amplify and that won't stomp on other goals.

If you are a manager and you don't understand what good code is, then don't make that judgment, for rewarded code should be exceptionally good. If it is not, then you'll look like an idiot. No matter what, the reward should not reflect any other attributes of the programmer receiving the award. If the programmer is an arrogant, foul-smelling antisemite working for an Israeli security firm, but he writes a chunk of code that makes people gasp at its beauty, then he gets the reward, no matter what the political consequences.

The rewards should be something the programmer wants. "Attaboy" rewards are distasteful. Rewards for the entire team for individual contributions are an interesting thought, but I have little experience with them. If you do, then I'd love to hear from you about them. The problem with individual rewards is that they are at some level in conflict with the principle of collective ownership.

Code Reviews

Code reviews are an extremely valuable tool for achieving code quality. They are among the most important forms of qualitative measurement, taking in all aspects of the code. Moreover, code reviews tie in very closely with agile principles. They fall into two broad categories: formal and informal reviews.

There is nothing magical about pairing. It involves a constant code review process. A second developer gets to comment on every line of code as it is written. Pairing provides a second set of eyeballs to help enforce coding standards. It's like taking a formal review and rolling it into the daily development process. Pairing does convey other social benefits, but they're not responsible for the primary quality effects.

There are some important things to remember about pairing. It shouldn't be done 100 percent of the time. There are some things best done solo. Researching new technologies, writing summaries, and just reviewing the code are often done best on an individual level.

Human interactions are a delicate thing. Most people in an office will be able to pair with most others, but there will almost certainly be some combinations that don't work well together. Some people may get on each other's nerves, be unable to communicate with each other, or just dislike each other in a deep, visceral way. These people shouldn't be forced to pair with each other. That is simply cruel and probably unproductive.

This is not to say that an individual who refuses to pair with anyone or offends everyone should be tolerated—but I am speaking specifically about particular pairwise interactions.

If a project doesn't use pair programming, then official code reviews should be instituted.

Renaming

Naming conventions help, but you (or others) will sometimes choose inappropriate names. Rename as soon as you realize that there is a better name. The longer a name remains in the code base, the more calcified it becomes. Other names are chosen to relate meaningfully and consistently with the extant names, and these names must also be changed when renaming.

When changes are limited in scope, refactoring tools are immensely useful. However, they can't find related names (yet), so they can't help locate all the interdependent names in a well-established project. However, they do make changing each name trivial, and making the early change becomes far less onerous. Get familiar with your IDE's renaming tools. Rename early, and rename far less often in aggregate.

Communication

Documentation should be available for the project. It doesn't have to be comprehensive, but it should guide developers through the overall system. Many errors are due to developers not understanding the problem domain, or the project's architecture and implementation. Reference materials for these should be available.

Written documentation is not sufficient for expertise in any field. A recipe prepared by Thomas Keller of French Laundry fame will taste very different from the same recipe prepared at home.[6] A great chef depends upon skills and judgments that have taken years to acquire. Doctors and veterinarians go through lengthy internships, and musicians spend endless hours practicing with each other. Programmers must communicate, too.

Curiosity is good. Questions should be encouraged, and opportunities for information exchange should be encouraged. Mentors help when a person first enters a company. Brown bag lunches on major subsystems are a good idea, too, but if I had my druthers the company would always supply the brown bags.[7] It's really a minimal expense compared to the salary of a skilled developer.

---

6. Thomas Keller is considered one of the best chefs, and his restaurant French Laundry is consistently listed among the top four in the world. See `http://en.wikipedia.org/wiki/Thomas_Keller`.

7. Brown bag lunches are (typically informal) talks given on the company premises. Often these happen at lunch time, and *brown bag* refers the fact that everyone brings their own lunch. Such arrangements are common at companies in the San Francisco Bay Area.

Technological Feedback: Bad Programmer, No Cookie

There are three places in the development cycle where tools can be used to provide feedback. At the keyboard, IDE or command-line tools can suggest changes according to coding conventions. When the defective code is committed, revision control hooks (a.k.a. triggers) can prevent it from reaching the code base or report questionable practices. When the code is built, analysis requiring the entire code base is performed. Running unit tests is one such example.

There are things that must never be allowed, and there are things that should not be done. The things you must never do can be prevented, and the things you should not do should raise warnings. Treating a "should not" as a "must never" is a crime. It maddens and frustrates people, and it makes your project unpleasant to work on.

There are some things that should never be allowed on a Python project. Some are typographical conventions. Unparsable code, leading tabs, inconsistent indenting, and questionable trailing whitespace should never be allowed into the code base. A build must never succeed if the unit tests fail, and significantly decreasing test coverage should never be allowed.

There are things that are looked down upon or that are indicative of other problems. Overly complex methods probably need to be simplified. Low code coverage suggests inadequate testing. Bad style probably indicates poorly reviewed or insufficiently thought-out code. These things should be checked and advised, but they should not trigger commit or build failures.

Coercion at the Keyboard

Pydev offers a variety of features to assist with writing and analyzing code. These features include the following:

- Autoformatting
- Code analysis (with Pydev)
- Templating
- Unit test execution

All of these features are configured from the Eclipse Preferences window. On the Mac, you open it by selecting Eclipse ➤ Preferences. On Windows, it is under Window ➤ Preferences.

Autoformatting is one of the most useful but least contemplated features. Pydev offers several settings, all of which have defaults conforming with PEP 8. The most important of these are the indentation settings, which are specified under General ➤ Editors ➤ Text Editors, as shown in Figure 8-4.

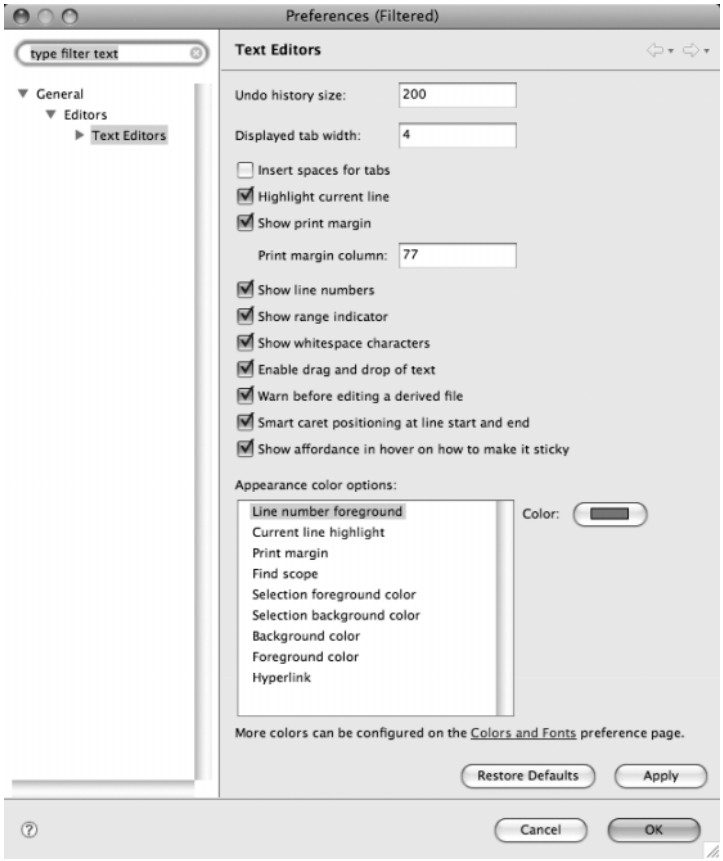

Figure 8-4. *Changing indentation settings in the editor properties*

To match PEP 8, "Displayed tab width" should be set to 4, and "Insert spaces for tabs" should be checked.

Under Pydev ➤ Code Style, there are four panels: Block Comments, Code Formatter, Docstrings, and File Types. These affect the way Pydev assists with code generation. It is clear how both Block Comments and Docstrings modify the formatting. Code Formatter only affects two changes: it determines if spaces are used after commas and if spaces are used both before and after parentheses. File Types determines which file name extensions the Pydev formatting tools operate, but these do not affect the association between the editor and Python files.

Under Pydev ➤ Typing, you will find more settings affecting the completions and formatting actions that Pydev performs. This panel is shown in Figure 8-5.

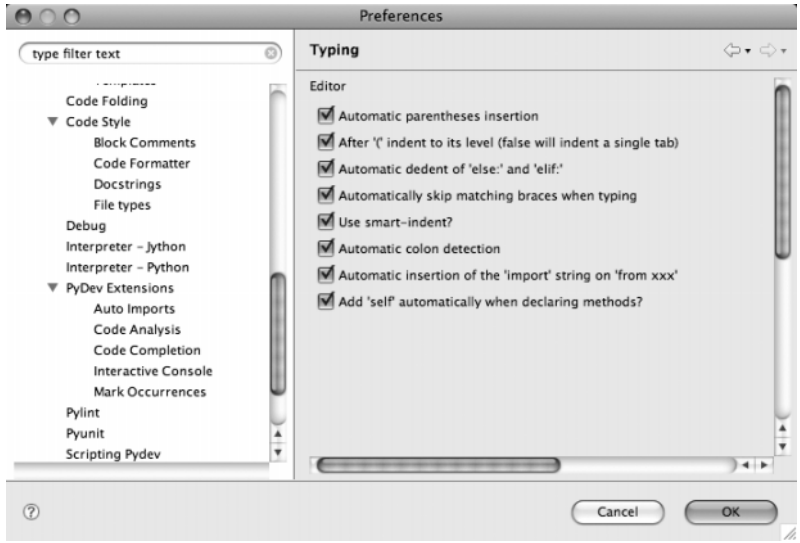

Figure 8-5. *The Pydev typing settings*

The commercial Pydev extensions add a significant number of code analysis features. These are accessed from Pydev ➤ PyDev Extensions ➤ Code Analysis. The tabs available are Options, Unused, Undefined, Imports, and Others. The odd man out is Options, which specifies when the analyzer runs. The rest of the tabs allow control of specific aspects of the analyzer itself. Each feature listed can signal an error or a warning, or be ignored. The Unused tab is shown in Figure 8-6.

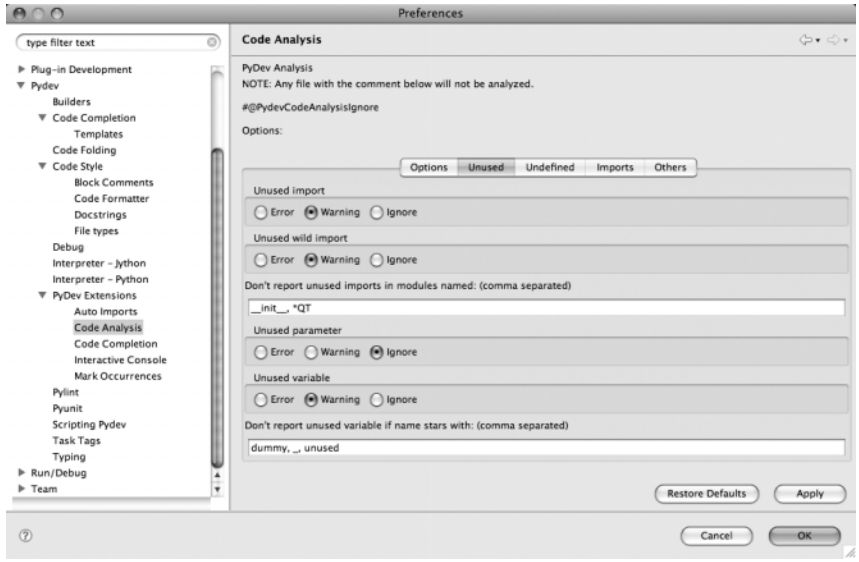

Figure 8-6. *The Unused tab on the Pydev Code Analysis panel*

The code analysis features provide the clearest examples of immediate feedback about errors in the code. These are only at the most rudimentary syntactic level, but they help identify issues that would otherwise be found when running the unit tests.

Templates are macro expansion mechanisms. When you press Ctrl+Enter, Pydev replaces the word you are currently typing with a template. Pydev defines macros for most Python control structures, but you can also define your own. The definitions can contain fields that must be supplied by the user.

In my environment, I have a template named `pym` that starts a PyMock Nose test. Templates are defined through the panel shown in Figure 8-7, who's path is Pydev ➤ Code Completion ➤ Templates.

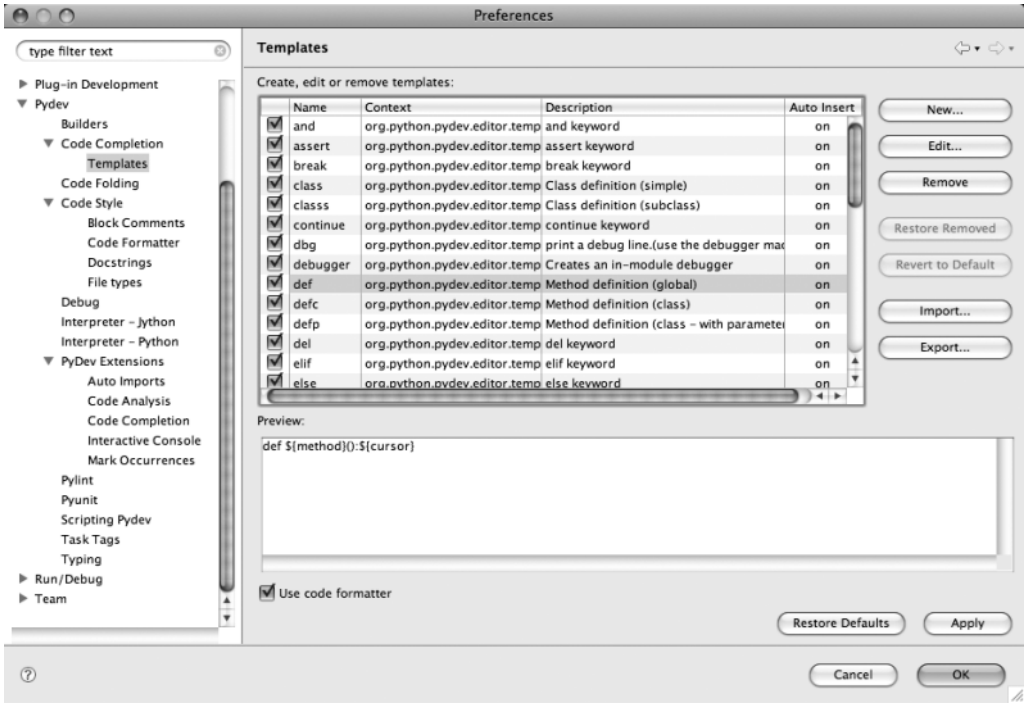

Figure 8-7. *The Pydev Templates panel showing the def macro*

The Edit button brings up the New Template window (shown in Figure 8-8). The Name field is the string used to choose the template when expansion happens. The Description field is just used in the Templates panel.

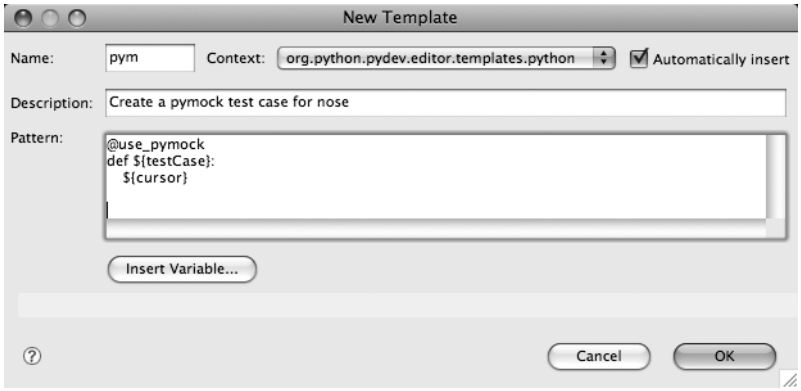

Figure 8-8. *Creating a new template*

The Pattern field defines the template body. Variables are denoted with the syntax `${foo}`. There are a handful of predefined variables that can be listed via the Insert Variable button. The most commonly used is `${cursor}`, which is the point the cursor will be left at when the macro expansion is complete. Unknown variables are filled in at expansion time, as shown in Figure 8-9.

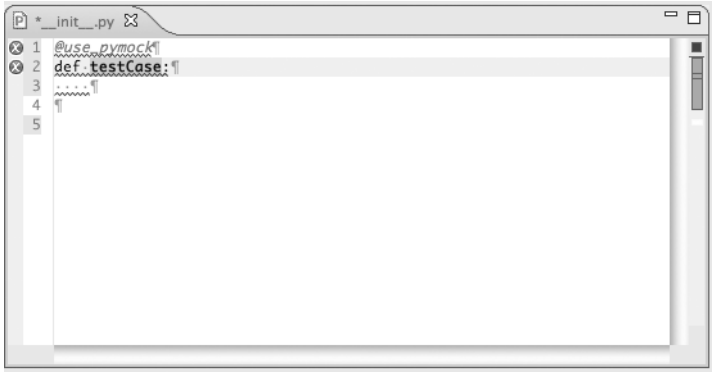

Figure 8-9. *The pym template being expanded*

The developer replaces `testCase` with the new method's name. When Enter is pressed, the focus passes to the next undefined variable. At the end, the cursor is placed where the cursor variable indicates. The templates can be exported and imported. A master templates file for the project should be exported to the root and checked in with the source, and it should be imported when it changes.

When Code Is Submitted

Subversion supplies hooks to validate committed code. These hooks happen after a transaction has been committed and the files have been copied to the server, but before the change has been committed. If the hook fails, the commit fails, and `stderr` is reported to the submitter.

This is the mechanism used to prevent atrociously malformed code from reaching the repository, and for supplying reports of offensive code.

To set this up, a `pre-commit` file is created in the Subversion `hooks` directory. On my development system, this is `/usr/local/svn/respos/hooks`. The file should be executable. If it is not, then it will fail. If the commit succeeds, this script returns a zero exit code. If it fails, the script returns a nonzero exit code and `stderr` is reported to the submitter. `stdout` is ignored. The precommit script acts as a dispatcher for other scripts that perform the real work.

Writing precommit hooks is a formulaic process, much the same in any language. The script must finish quickly to prevent the repository from blocking, so running a full compile to verify integrity is out of the question.

Precommit hooks are always passed the same two arguments: the repository path and the transaction number. These are used in conjunction with `svnlook` to get information about the transaction. The list of files is retrieved with `svnlook changes`. These files are iterated through, and the contents of each file are printed using `svnlook cat`. The log message can be accessed with `svnlook log`.

While the hooks should examine the code written in the project, there are some Python files that should be ignored. These are third-party files checked in to facilitate builds, so the precommit scripts must ignore these subtrees.

Amazingly, this hasn't been packaged up until now. I've done it as part of this chapter, however. The package is named svnhooks, and it can be installed with `easy_install`. It provides a simple framework for producing your own hooks. It can terminate the build or send notifications, and it comes with hooks to check for the following:

- Leading tabs
- Mismatched leading whitespace
- Windows line endings
- Trailing whitespace after \ at the end of the line
- Syntactically correct Python
- Suspiciously complex code via PyMetrics
- Questionable semantics via PyChecker or PyLint

PyChecker and PyLint are semantic verifiers. They check for constructions that are legal but questionable. Such things include redefined methods or locals overriding Python built-ins.

■ **Caution** Svnhooks supports PyLint, but the package is in questionable condition for use under Python 2.5. As of version 0.14.0, it won't correctly install unless the source is altered by hand.

My project's precommit script fails if indenting, syntax, or line lengths are invalid. It sends warnings to dev@sample.org if the code is too complex or if it fails the lint check. The file follows:

```
$ cat  /usr/local/svn/repos/hooks/pre-commit
```

```
#!/bin/sh

# PRE-COMMIT HOOK
#
# The pre-commit hook is invoked before a Subversion txn is
# committed. Subversion runs this hook by invoking a program
# (script, executable, binary, etc.) named 'pre-commit' (for which
# this file is a template), with the following ordered arguments:
#
#   [1] REPOS-PATH   (the path to this repository)
#   [2] TXN-NAME     (the name of the txn about to be committed)
#
# The default working directory for the invocation is undefined, so
# the program should set one explicitly if it cares.
#
# If the hook program exits with success, the txn is committed; but
# if it exits with failure (non-zero), the txn is aborted, no commit
# takes place, and STDERR is returned to the client. The hook
# program can use the 'svnlook' utility to help it examine the txn.
#

IGNORE='[^\/]+/(?!thirdparty)/.+'
ADDR='dev@sample.org'

/Users/svn/bin/whitespace_check "$REPOS" "$TXN" "$IGNORE" || exit 1
/Users/svn/bin/syntax_check "$REPOS" "$TXN" "$IGNORE" || exit 1
/Users/svn/bin/length_check "$REPOS" "$TXN" "$IGNORE" || exit 1
/Users/svn/bin/complexity_check "$REPOS" "$TXN" "$IGNORE" -m $ADDR
/Users/svn/bin/lint_check "$REPOS" "$TXN" "$IGNORE" -m $ADDR

# All checks passed, so allow the commit.
exit 0
```

For testing purposes, the checks can be run against known revisions using the -r flag. I frequently ran the whitespace check against revision 29 on my system when I wanted a successful test:

```
$ whitespace_check -r 29 /usr/local/svn/repos '[^\/]+/(?!thirdparty)/.+'
$ echo $?
```

```
0
```

Buildbot and Coverage

When I introduced Buildbot, I showed how to use it in conjunction with Nose to run unit tests. However, Nose can do much more. Together with the coverage package, it generates coverage reports. Coverage works with `easy_install` . . . sort of. At the moment (version 2.78), it is broken ever so slightly. The problem is known, and there is a patch for it.

All packages in the build we constructed are stored locally. This means that a locally patched version of coverage can be used. Retrieving the package from its distribution site is the first step. This is located by consulting `http://pypi.python.org`, the Python package repository.

The package is pulled down, unpacked locally, checked into source control, and subsequently patched. The version number is changed from 2.78 to 2.78p1 to designate that it has been patched.

A new package is generated with `python ./setup.py sdist`, and the resulting package file `dist/coverage-2.78p1.tar.gz` is copied into the project's `thirdparty` directory. The dependency on `coverage-2.78p1` is added to the project's `requires` attribute in `setup.py`.

The project is installed locally with `python ./setup.py install`, and the coverage tests are successfully run through Nose. This is done simply by adding the `--with-coverage` option when calling `nosetests`. The changes to the package are now known to function, so the patched files and the new third-party bundle are committed to their respective repositories.

Running coverage through Nose against an early version of svnhooks produces the following report:

```
$ nosetests -w src/test --with-coverage
```

```
.............................
Name                 Stmts   Exec  Cover   Missing
------------------------------------------------
decorator               40     32    80%   62, 76-83, 123
getopt                 103     15    14%   43-45, 48, 79-93, 110-143, 146-162,
168-186, 189-201, 204-207, 210-211
pickle                 854    262    30%   84, 95-96, 197-207, 218, 222-225,
242-247, 251-257, 261-267, 271-331, 335, 339-343, 350-420, 427, 431-434, 438-458,
...
1358-1359, 1362, 1365-1367, 1370, 1373-1374, 1379-1380, 1383
pymock                  73     52    71%   31, 47-48, 66-67, 74-75, 80-81, 86-87,
92, 95, 98, 101, 106, 109, 112, 117, 122, 127
pymock.pymock          493    329    66%   78-81, 92-93, 101-103, 106, 116-123, 127,
144, 150, 153-154, 159, 162, 165-166, 171, 176, 181-182, 185-188, 191-193, 206,
...
734-735, 739-743, 747-750, 754-755, 759-760, 781, 789, 797, 806, 818, 824, 835-836
sets                   286     67    23%   60-79, 93-94, 101, 108, 114-117, 124,
132, 150-153, 156-159, 165-167, 178-185, 201-203, 210-212, 219-221, 228-235,
...
493-495, 499-505, 511, 515, 524-530, 537-543, 550-553, 557, 561, 565, 573-574, 577
subprocess             496     42     8%   368-369, 371, 375-398, 410, 415-417,
422-429, 443, 456-462, 493-530, 540-622, 626-628, 632-639, 653-667, 674-909,
...
```

```
1083-1089, 1095-1103, 1109-1112, 1116-1181, 1188-1222, 1229-1239, 1243-1246
svnhooks                  0      0   100%
svnhooks.indent          46     23    50%   17, 23-24, 28-31, 39-46, 49-51, 54-60
svnhooks.precommit       78     70    89%   32-33, 47-48, 51-52, 90, 99
svnhooks.syntax          18     17    94%   32
tabnanny                173     29    16%   28, 36-40, 44-58, 66, 68, 70, 72, 84-130,
156-176, 181-182, 199-203, 208, 215-223, 239-249, 256-264, 267-271, 274-325, 329
term                      0      0   100%
term.framework          814      0     0%   3-1123
----------------------------------------------------
TOTAL                  3474    938    27%
----------------------------------------------------------------------
Ran 31 tests in 0.089s

OK
```

The first column is the name of the package. The second is the number of statements in the file, followed by the number executed, and then the percentage calculated from those two. The final column is a list of lines and line ranges that were not covered.

The coverage report has one noticeable weak point. It doesn't distinguish between built-in packages and subject packages. The report simply includes all the modules that are imported.

The code coverage report is easily patched into Buildbot. The necessary changes to the configuration created in Chapter 5 are shown here in bold:

```
def python_(version):
   return  "../../python%s/bin/python" % version

def nosetests_(version):
   return "../../python%s/bin/nosetests" % version

def site_bin_(version):
   return  "../../python%s/site-bin" % version

def site_pkgs_(version):
    subst = {'v': version}
    path = "../../python%(v)s/lib/python%(v)s/site-packages"
    return path % subst

def pythonBuilder(version):
    python = python_(version)
    nosetests = nosetests_(version)
    site_bin = site_bin_(version)
    site_pkgs = site_pkgs_(version)
```

```
f = factory.BuildFactory()
f.addStep(SVN, baseURL="svn://repos/rsreader/",
   defaultBranch="trunk",
   mode="clobber",
   timeout=3600)
f.addStep(ShellCommand,
          command=["rm", "-rf", site_pkgs],
          description="removing old site-packages",
          descriptionDone="site-packages removed")
f.addStep(ShellCommand,
          command=["mkdir", site_pkgs],
          description="creating new site-packages",
          descriptionDone="site-packages created")
f.addStep(ShellCommand,
          command=["rm", "-rf", site_bin],
          description="removing old site-bin",
          description="site-bin removed")
f.addStep(ShellCommand,
          command=["mkdir", site_bin],
          description="creating new site-bin",
          descriptionDone="site-bin created")
f.addStep(ShellCommand,
          command=[python, "./setup.py", "setopt",
                    "--command", "easy_install",
                    "--option", "allow-hosts",
                    "--set-value", "None"],
          description="Setting allow-hosts to None",
          descriptionDone="Allow-hosts set to None")
f.addStep(Compile, command=[python, "./setup.py", "build"])
f.addStep(ShellCommand,
          command=[python, "./setup.py",  "install",
                   "--install-scripts", site_bin],
          description="Installing",
          descriptionDone="Installed")
f.addStep(ShellCommand,
          command=[python, "./setup.py", "test"],
          description="Running unit tests",
          descriptionDone="Unit tests run")
f.addStep(ShellCommand,
          command=[nosetests, "./src/test", "--with-coverage"],
          description="Determining code coverage",
          descriptionDone="Code coverage determined")
return f
```

Summary

People are error prone, and they're prone to inflated judgments about their own capabilities (of course, this excludes you and me). Programmers are people, and so they carry over the same faults. Without adequate feedback, they can't get an accurate assessment of their performance, so it is critical to get feedback one way or another.

Getting feedback translates to measuring and reporting some aspect of the development process. These measurements fall into two broad categories. Quantitative measurements answer the question, "How much?" They are the sorts of things that produce hard numbers, and are easily analyzed by computers. Qualitative measurements describe what something is, and they tend to be things that people are good at determining.

Measurements have many aspects, and they need to be considered before investing time and effort. You must understand the characteristics of what you are measuring, the characteristics of the instrument you are measuring with, and the interactions between the two. You must understand what the measurements mean, what the impact of measuring will be, and what the possible adverse outcomes are.

Quantitative measurements tend to have very narrow definitions, and are only applicable in limited scopes. Many quantitative measurements are required to get an accurate assessment of a software project's overall condition. Among those commonly encountered are code coverage, cyclomatic complexity, and velocity.

Qualitative measurements tend to have much broader application, and they give a much deeper picture of a project's condition. They're also much harder to determine since they require human intervention, and collecting them systematically often requires procedural support. The granddaddy of all qualitative measurement regimes is the code review, although in agile development environments, pair programming often takes the place of formal reviews.

Such measurement regimes provide feedback about our behavior and work. Feedback can be achieved by social or technological means, but you should be wary of the temptation to use technology to solve social problems, as cultural norms and peer pressure are often more effective ways of shaping behavior.

Technological solutions have limited domains, but are often effective when used appropriately. There are a number of tools that help to provide feedback. Eclipse, Pydev, Subversion, and Buildbot can all provide useful feedback when used correctly.

Chapter 9 looks at databases. Databases are their own little world, and they present thorny issues when it comes to agile development. The issues of incremental upgrades and downgrades to live databases are largely solved, but only in ad hoc ways. Moreover, these mandate very different machinery from the object-relational mappers that are used to access databases today.