

Chapter 8

Designing for Decentralized Development

The Importance of Software Design

Have you ever used a software application and after a while thought, "I like the design and architecture of this program, but the bugs are just making it worthless." Or maybe as a developer you have been in a situation where you were really proud of the code, but in the end needed to reprogram everything because of initial design errors. This clearly shows how design and coding can be quite different—and, in fact, quite often these two steps are done by different individuals altogether. These examples show that you can have bad design and good coding (hard to fix later), or good design and bad coding (easier to fix than the other situation). Or you can have bad design *and* bad coding, in which case it is easier just to scrap everything and start everything over—or buy or download a finished product. The best case, as is often true of open source, is good design and good coding. The Linux operating system is an example of both.

What is design? It can be defined in a number of ways, but for this book, let's just agree that software design is about *comprehensibility*—that is, it's the part of programming concerned with making programs understandable to and maintainable by humans.

Good design does not:

- Make programs run more efficiently in terms of memory usage or speed.
- Improve the quality or accuracy of their output.

Good design does, however:

- Make the general plan of the program easy to grasp. When the program is "graspable," it means that for any given behavior of the software, you can find the region of source code that controls it.
- Enable a programmer to figure out the cause of a bug quickly.
- Help design-sensitive programmers see immediately only one or two plausible ways to fit in a new feature, instead of an infinity of equally possible (and equally nonintuitive) implementations.

We cannot emphasize too much that comprehensibility is a concern of people, not code. The code couldn't care less whether, from a human's point of view, it is clean and beautiful or tangled and knotty. By the time the computer runs it, the code has been transformed (compiled) into a numbingly long string of machine instructions, from which only very patient programmers can discern the original plan of the source code with great effort. Software does not run any better because it has been divided into intuitive modules or because its data structures have been segregated by role and documented. These benefit only people, not machines.

Proprietary Software Design vs. Free Software Design

In some cases, it is possible to completely hide bad design from the end user (although bad design in most cases tends to surface in one way or another.) Of course, only proprietary software has hidden internals. The inner mechanisms of free software are constantly exposed and studied. You might think that this would make good design even more important for free software. Free programs do tend toward classically "good" design over the long haul, but their authors are usually less concerned with well-structured design at inception than authors of proprietary programs are.

Why is that? One reason is that a commercial software vendor can afford to make a high initial investment in the code, confident that it will be paid for by sales over the long run. That means the vendor can heavily subsidize the initial design stage. In addition, the more maintainable the program, the less costly it will be for the company to produce each upgrade. Another reason might be that the senior programmers at proprietary software companies often have degrees in computer science or software engineering, so they come with the traditional academic preference for solid foundations. In a sense, a program is a mathematical construct or proof, so if it does not proceed clearly from well-established premises, it is not quite academically acceptable.

Formal qualifications and theoretical correctness seem to hold less influence in the free software world, however. Admittedly, there is a never-ending discussion among free

software enthusiasts about what constitutes good design, but what one sees in chat groups and mailing lists is not always an accurate indication of what's happening on the coding front. When it comes to successful free software projects, clean design—although certainly desirable—is of secondary importance to the ability to simply run reliably on a wide variety of platforms. Some reasons for this are:

- In a free project, various contributors make many small initial investments at different times, instead of one organization making a single massive initial investment. These individual contributors became involved in the first place because of their interest in what the code does (as opposed to an interest in earning their salary or gaining the approval of the project leader). Their first job is to install and run the code, not peruse it. If they're able to get it up and running, they'll almost automatically be inclined to evaluate the program's design more charitably.

- The majority of contributors to free software don't need to carefully inspect the code, although that is really the only way to develop a truly informed opinion about a program's design. New contributors usually just want some small, incremental addition to the program's behavior (after all, if they could obtain their desired results only by making major changes to the program, they would probably write something from scratch themselves or find a more appropriate project to join). For such quick, minor changes, a good strategy is generally for them to dive in and start hacking. There usually isn't time—or, to be honest, inclination—to take a broad survey of the code and then make changes in the most theoretically pleasing way.

However, the triumph of practice over theory in free software is indicative of something deeper than mere time constraints. Although everyone agrees that programs should be easy to understand and modify, no two people will agree on exactly what that means in terms of actual code layout and data structures. Often, a design that looked good when the project started turns out to be unwieldy and obstructive one year and several thousand lines of code later. Because of this, most programmers would postpone as many design decisions as possible until after the code has seen some real use.

In free projects, they can do just that—the program needs to run, but it doesn't need to be mature or settled. Commercial software has to be not only usable, but also polished and professional, right from the first release. Polished, professional software does not confuse (or stimulate) the user by suggesting alternative uses or unexpected applications; instead, it tries to do exactly what its marketing plan claims it does. Thus, its design must be relatively complete from the very beginning of the project. Free programs can postpone design decisions for as long as the developers are comfortable; commercial programs generally cannot.

Cost Issues

The vendor of commercial software is also concerned with controlling the total cost of producing the software over the long term, including post-release maintenance and upgrades. The additional effort and expense involved in creating a sound design is justified

because it minimizes expensive rewrites or workarounds later on. Even though a careful, comprehensive design probably means a steeper learning curve for developers (who will need to familiarize themselves with the master plan before doing anything to the code), this extra effort usually pays off for the vendor in the long run.

Free software projects, though, are not really concerned with minimizing the total "cost" over the entire lifetime of a project. Instead, they're concerned with minimizing the cost to any one contributor at any one time. Sharp, immediate impediments to modification—say, having to understand a complex master plan—impose a high initial cost on participation. However, constant but low-key hassles—such as dealing with a slightly disorganized code base in which it's relatively easy to find what you're looking for even when you can't tell quite how it relates to everything else—are not likely to discourage anyone from participating. Volunteers might be psychologically more comfortable with loose designs, too. People are more inclined to jump in and start hacking if they don't feel like they might be upsetting some delicate and intricate master plan.

Design Invariants

Free software projects tend to settle at the outset on a very small number of invariants ("We're going to organize everything around a text-buffer structure" or "The records will be kept in an SQL-compatible database"). This allows the project to get started without complete chaos, and then the rest of the project can grow around those decisions. Indeed, we'd go so far as to say that the two most valuable skills a designer can have in the free software world are:

- Knowing which invariants will prove productive rather than constricting
- Knowing what questions *don't* have to be answered right away

Refusing to design prematurely is an open acknowledgment that we just don't know what the world will do with our code. Our imaginations are limited and our intellects are puny when faced with such highly complex situations. Who can calculate the outcomes of hundreds or thousands of user-program interactions, especially when the users have access to the source code? It's better to devote our resources to increasing the code's adaptability than to anticipating every future direction of development.

The important thing, then, is not to come up with the perfect design at the start, but to design the software so that it can evolve. Partly, of course, this is simply a matter of the developers keeping an open mind, but it's also a matter of how the code is organized.

This organization constitutes a kind of message to future developers, in which you make clear which parts of the code are design invariants and which are not. Most parts of the code are not invariants—they are quick decisions made to get things up and running, and don't claim to be the best solution.

Even the so-called invariants are subject to reconsideration. But if the original developers' judgment was reasonably sound, then the assumptions they saw as structurally defining usually also turn out to be sound. Those assumptions could, theoretically, be changed, but changing them to accommodate some new design often requires an effort equivalent to—or more expensive than—rewriting the entire program from scratch. Therefore, in practice, the core of the code evolves much more slowly than the rest, and rarely loses its basic design.

Code Design

When it comes to concrete design organization, there are good practices, but no universally agreed-on best practices. So we're just going to describe some methods that we've seen work in the past—but don't hesitate to experiment liberally.

We're not going to discuss design principles that are universally accepted as equally applicable to both open and closed source projects. Thousands of books are available explaining the necessity of avoiding unnecessary duplication of code, breaking the project into manageable components with clear entry and exit points, and many other useful guidelines. This chapter is about open source software design, not design in general.

The Design Document

First, document the core assumptions of your code. Save this information in a file that is easily accessible (for example, in a file called "DESIGN" at the top level of the project). This file should describe the code only at a very general level. You should include detailed descriptions of the central data structures, for example, in comments in header or source files, so they can be updated as the structures evolve. The design document gives an overview of the project's overall shape as it was first conceived. If the assumptions stated there are general enough to remain more or less invariable, then you won't have to worry about the document becoming obsolete, and it might need to be changed only rarely, if at all.

A design document makes it much easier for new developers to grasp the project as a whole entity. Once they understand the basic design, everything else they examine in the project will have a ready-made framework to fit into.

Note

Be wary of the design document becoming too large. You might be including noncentric assumptions, which are likely to change quickly. Then that portion of your document would be out of date, which might be worse than not being there at all. Here's a good strategy for avoiding this: If you find yourself going back into the design document to revise a section in more than a minor way, consider removing that section entirely. The design document is intended to include only written-in-stone truths, not field reports that require revision from time to time.

The corollary to having a few clear central assumptions is that the rest of the project—the majority of the code—should *not* be written in stone. This means acknowledging (in comments in the code or perhaps via a mailing list) that some procedure could have been done a different way, and might even be better if done that way. When you've written a piece of code in the quickest way possible, it's rare for it to also be the most efficient implementation. That's okay for the first release—the quicker way is often preferable. However, including a comment in the code at that point might encourage people to try alternatives if that portion of the code becomes a bottleneck.

Here are some general points about writing for distributed developers:

- Write your code to be clear and self-explanatory, because you're not going to be there to train new arrivals. The clarity of the code has a direct bearing on how much other people will be able to understand and modify it. A navigable plan is better than an all-encompassing but complex one.
- Write every line of code as though you're taking part in a conversation with the other developers. After all, you are, although the conversation is stretched out across time, and some of the people you're conversing with might not even have joined the project yet. You are not on stage—your goal is not for them to see a flawless performance, but for them to understand what you were thinking when you wrote the code. You want to make them feel confident that they're as cognizant of all the major issues as you were when you wrote it. When people feel comfortable in a region of code, they'll feel at ease in changing it, which is exactly what you want.
- Don't be afraid to leave comments expressing doubt or second thoughts about what you've done in the code, any more than you would hesitate to say such things in a conversation. Later, when others are wondering why you wrote the code a particular way, they'll see your comments, revealing that you also entertained thoughts of doing it another way. This will reassure them that they're onto something and make them less hesitant to improve the code there. (Although comments might not traditionally be considered part of a program's design, they do interact with the design when they indicate how closely a certain part of the source is connected to the overall plan.)
- Use syntax-sensitive indentation throughout the source code. Most programmers expect this of publicly distributed code nowadays, and they might be confused or even misread parts of the code if it's not indented consistently.

Dividing Code into Files and Directories

Turning our attention to the next level of organization, let's consider the arrangement of files and directories.

How you divide up the code semantically (data structures, calling conventions, and so on) is very much a matter of taste, of course, or of style. But the best way to divide the code up *physically*, in our experience, is to use shallow directory hierarchies, at least at the beginning

of a project. Partly this is due to the odd way CVS removes obsolete directories: It doesn't remove them, it just has a special-case feature for making them vanish from the working copy once all their files have been removed. (The directories are still there, lurking in the repository, taking up space, and occasionally confusing people who forget to pass the **-P** option to **cvs update** to prune them out.) The fewer directories you create at the beginning of the project, the less frequently you will have to remove directories later on.

There is another reason to avoid creating a lot of subdirectories when starting a project. Sometimes, people are tempted to "over-hierarchize" as a way of making statements about the structure of the code (as an extreme example, putting all the networking code into a net subdirectory, all the printing code into a print subdirectory, and so on). The problem with this is that you usually can't *know* at the start of the project what parts will turn out to be natural modules later on. Maybe the printing code will turn out to be just one of many output drivers, and end up belonging in an outputs subdirectory that gets added to the project later, or maybe it will go away entirely, with the printing functions handed off to an external library instead. If an artificially imposed directory hierarchy biases contributors' organizational ideas, the contributors might be slow to see alternatives that would be immediately obvious under a looser arrangement. Absence of structure is a message that says, "We don't know yet how everything will turn out, so we're waiting for the proper arrangements to pop up by themselves."

Placing portions of the code into subdirectories is helpful when the time is ripe, just don't be overeager to feel that the time is ripe. A file's position in a hierarchy is really less important than an understanding of what the file actually does.

Also, a minor technical advantage to having most of the project in one directory is that the files are slightly more convenient to navigate. Fewer keystrokes are required to call up a new source file, and you can easily invoke basic search tools (such as grep in Unix) on all the files in the project.

Dividing Code into Modules

The semantic layout of the code—the division of the code into modules or subprograms, each performing a discrete portion of the whole task for which the software was designed—is related to the physical layout. However, this process is harder to generalize about because it can be so different from project to project.

Modules can help keep the code robust, even code that is in the busy hands of random contributors who might have varying levels of experience and competence. The point of a module is to group all of the code for a given subsystem into a definite set of files, so that when you change one part of that code, you know where to look for dependencies that might need to compensate for the change. Commonly, all the code in a given module will share certain data structures, and those data structures will be used nowhere else in the program—the module is entirely responsible for maintaining their integrity and understanding what each of those data structures means.

Access to a module is usually done through designated entry points—functions (or, sometimes, exported variables) that the rest of the program knows to use when it wants something from the module. All of the code outside the module then scrupulously avoids ever referring to any part of the module not explicitly advertised as an entry point. Indeed, many modern programming languages make it possible to render the module's internals actually invisible to the rest of the program, and you should take advantage of those features whenever possible. Although this might seem restrictive, it actually frees you because it means anyone can rewrite the internals of the module to an arbitrary degree (say, to use better algorithms) without disturbing the external code that depends on the module. As long as the entry points remain the same—that is, keeping the same names, taking the same arguments, and returning the same sorts of values—what goes on inside the module is nobody's business but the module's.

As far as it goes, this advice is applicable to any kind of software project. What's different for open source projects is how modules are treated *politically*. In a closely managed project with a programming team that is more or less set (which is true of most proprietary software projects), responsibility for the various modules tends to be fairly formalized, with most or all changes to a given module going through that module's maintainer. From a management perspective, this makes a lot of sense: You know the programmer is going to be there for a long time, because it's her job, so she becomes the house expert in the intricacies of that module. Meanwhile, the rest of the team can treat the module as a "black box" and be spared the necessity of learning its code. Also, it's easier to measure a programmer's contribution when his or her coding isn't mixed and diluted with everyone else's.

A free software team is not optimized for this particular kind of efficiency, however, because they don't know how long a given member will be around nor how much time that member will be able to devote to the project this week (or month, or whatever). Although there is sometimes a "responsible party" for a certain module, it's a much looser kind of responsibility, and anyone on the team will generally feel free to make changes to the module (or they will if the project is socially healthy and the programmers are free of destructive territorial instincts). When the module doesn't provide quite the interface needed by outside code, the person working on the calling code will often jump into the module and just *make* it provide the missing feature.

We are indebted to Jim Blandy—cofounder (with Karl Fogel, one of the authors of this book) of Cyclic Software—for identifying this "borderlessness" as one of the distinguishing characteristics of free software. Several years ago he was discussing some technical issues with another programmer while Karl listened. The programmer was complaining that one of the major modules of a certain free program didn't provide a well-designed interface to the rest of the code and this deficiency was causing him to have to write things in an awkward way. Jim raised his eyebrows archly and said with righteous fervor, "Well, change the provider, then! Isn't that the free software way?" Absolutely. This is especially true when the code is kept in CVS or some other version control system; the worst thing that can happen if someone makes an ill-advised change to a module is that the change will have to be reverted. It's no big deal—and probably a valuable learning experience for the programmer.

In some ways, it might be less efficient when everyone has to know a little bit about the module, as opposed to having one person be the official expert and gatekeeper. However, free projects optimize in favor of a distributed burden, lessening the vulnerability of the module to any one person's schedule (or lapse in judgment, for that matter). Over time, the module and its callers will slowly find their way to a polished balance that lies somewhere between a perfectly opaque black box that reveals nothing about its inner workings and a perfectly controllable engine that needs to be told how to do every little thing. This balance will be the right one, because the same people who use the module also have the opportunity to tune it. There can be no self-delusion about the intuitiveness of its interface or the efficiency of its implementation, because any sufficiently dissatisfied user of the module will do whatever is necessary to fix it.

Evolution-Centered Design

There is an obvious analogy between software development and biological evolution. Free programs often end up resembling living organisms and can be compared to nonfree programs in roughly the same way that natural organisms can be compared to manufactured machines. Anyone with an engineering bent who spends some time studying living creatures is usually surprised by the extreme inconsistency of natural selection's design sense. Nature might use a beautiful and intuitive—to our sensibilities—solution (say, having an internal skeleton) in a totally inappropriate way (say, the reshaping of ancient jawbones into part of our hearing apparatus, where they serve a purpose completely unrelated to issues of rigidity or load-bearing). But beauty and appropriateness are only meaningful to us as we consider the results from an aesthetic point of view. For the evolved creature, the only thing that matters is that things work well enough for the creature to survive and reproduce (and we do hear, after all).

The dynamics of free software survival are much like the natural evolution theory: If the code runs well, it will be copied more; if it doesn't, it will be copied less or not at all. If you work with such software long enough, you might even find your ideas about good design changing to accommodate all sorts of solutions that, while they might seem odd and unexpected at first glance, turn out in practice to work very well. Although the idea of letting the software largely design itself is not new, it also has not received the kind of attention we think it deserves. We even made some attempts to coin catchy new words for evolution-centered design, but none of them turned out very well So we'll have to discuss the fundamental concept of evolutionary design using an eight-syllable phrase to refer to it (unless someone wants to contribute a new one).

Evolutionary design really is qualitatively different from top-down, master-plan–based design, the kind that dominates the proprietary software industry. It's not that top-down programs don't also evolve, but their authors aren't depending on evolution to make major decisions for them. They have other sources of guidance (contractual requirements, focus groups, magazine reviews, company politics, and presumably even user feedback) for the

program's future. As an evolutionary force, however, these sources pale in comparison to having users indicate their preferences by directly hacking on the code. Even if a user botches a modification and someone else has to rewrite it, the fact that they tried at all sends the developers a clear message that the change was very important to someone. When hordes of random strangers start using your code, you soon discover where the preference lines truly fall. How do people *really* want the software to behave? You'll find out just by watching what they code; it's better than any focus group.

Comprehensibility—good design—is just one more evolving property. Although immediate usefulness might cause people to start working on a program, overall maintainability probably does a lot to keep them there. If the program didn't slowly become more navigable and more developer-friendly over time, you'd have to wonder what was wrong. The people working on it have every reason to organize it well, and experience with the code gives them the knowledge they need to do so.

The most beautiful example we know of evolutionary design has nothing to do with software; in fact, it probably never happened. You might have heard the story about the architect of a university campus who deliberately left pedestrian paths out of his design and specified that grass be planted everywhere. When the university's board of directors inquired about this seemingly major omission, he smiled and said, "Wait." After a year had passed, he returned to campus, observed the routes along which the grass had been most trampled by students, and ordered concrete paths to be built along those routes.

We have heard this tale now about three different architects and three different campuses, which leads us to believe it's only an urban legend. Anyway, everyone knows that professional architects don't behave like that. Nevertheless, it always sounded like a brilliant idea to us: Don't tell the students where to walk; instead find out where they *do* walk and make it more convenient for them to do what they do already. Don't tell reality how you want it to behave, but *ask* it how it behaves instead. The architect laid down some core assumptions—the buildings—and then let a supposedly major design choice essentially decide itself, by watching how the system flowed around the invariants.

With software, which is far more malleable and cooperative than grass and concrete, you have the opportunity to do this sort of thing every day. But we're trained not to. We're taught to try and hold the entire project in our minds at once, to have a master plan, and to shape the project to fit the plan. We think a lot of software would be better off if people let it design itself (or rather, if they paid more attention to what reality, in the form of users and contributors, told them about their software).

Principles of Free Software Design

Having made our case for free evolution, we now turn around and offer some inviolable principles of design, ones that are especially important for free software.

Don't Limit Input

When accepting input, tolerate no arbitrary limits on the size of the data (this applies to all kinds of input: streams, file contents, file names, prompted command lines from the user, and so on). This is one of the best known of the GNU Coding Standards (which are all worth reading—you can find them at **www.gnu.org/prep/standards_toc.html**). It's partly motivated by security concerns. (Nonobvious limits on input length can result in buffer-overflow bugs—if you don't know what that means, just take our word for it that it's something you want to avoid.)

A ban on arbitrary limits is important for your program to remain evolvable. You can't predict what kind of input people will send or for what purpose. What if the description slot in some data structure ends up being used to hold a full JPG image instead of the short textual description of a person's face it was originally meant to hold? At the time the data structure was designed, no one could foresee that it would one day be used to store images (which typically require much more storage room than text). Nevertheless, if the receiving area is dynamically allocated to be as large as the incoming data, there won't be a problem. Avoidance of arbitrary limits automatically makes the code more robust in the face of unexpected input, and the code becomes more extensible because developers don't have to worry whether parts of the input will inexplicably disappear when stored in certain locations.

In addition to tolerating data of any size, the program should also, for the same reasons, tolerate multiple-type data. In other words, it shouldn't modify a binary data stream (by zeroing the high bit of each byte, for instance, as some programs have been known to do) unless munging the data in that way is part of the program's purpose. Always leave the data as you found it.

Note

Neither of these injunctions should be taken to mean that the program has to know how to interpret any kind of data it sees. If the code expects plain text and doesn't know what to do with a JPG image, that's fine. The issue is not so much that people will suddenly start storing images where they formerly stored text, as that they might make modifications to the program that depend on it being able to store an image where it formerly stored text. If they proceed under this assumption, only to discover that most of each image is missing because the data structure silently truncated it, they'll feel—rightly—that an implicit promise has been broken. The software doesn't have to handle every conceivable input; it just shouldn't close any doors to the possibility of receiving every conceivable input.

Use a Consistent Interface

When the program has to interact with the outside world—for example, via file formats or network protocols—it's important that it present a consistent interface. These interfaces must actually be more carefully planned for free software than for proprietary software. Free third-party tools tend to spring up very quickly to augment free programs, and the authors of

those tools don't want to constantly rewrite them to cope with changing interfaces. (Commercial software companies can afford to have someone responsible for tracking format changes in other vendors' products and advising programmers on how to compensate for them. This won't happen with volunteer programmers—instead, they'll just stop interfacing with your software and start interfacing with someone else's.)

This means that formats and protocols need to be planned thoroughly in advance; otherwise, you might discover later that you have unintentionally built unexpected and inconvenient limitations into them. For example, one of the first programs Karl Fogel wrote was an extension to the Emacs text editor that allowed users to set "bookmarked" locations in text files (just like the bookmark feature found in most Web browsers). Obviously, the bookmarks had to be stored in a file on disk between Emacs sessions, so Karl whipped up what he thought was a sufficiently powerful file format and wrote code to write it out and read it back in.

As the software acquired a respectable user base, Karl began to get emails from people suggesting additional information that could be stored with each bookmark (annotations, for example). Certain of these suggestions were made with such regularity that he knew the software really ought to incorporate them. Unfortunately, Karl hadn't left any space in his file format for storing annotations or any of the other new information. He hadn't even taken the most basic step of leaving an "escape hatch" slot in each entry that he could toggle on to indicate the presence of additional information not accounted for by the original format.

Eventually, Karl resorted to an inelegant solution that at least allowed compatibility in both directions: He made up a special "version stamp" that would appear at the beginning of the record file, telling the software what version of the format this file used. An absent version stamp was decreed to mean version 1, corresponding to the first release of the software. The newer versions of the software were given special code to convert version 1 format files to version 2. (And if that all sounds like a great pain in the neck, let Karl assure you that it was!)

Version 2 was a much better designed format, allowing essentially arbitrary numbers of key-value pairs. For the sake of forward-compatibility, the software was now free to simply ignore any keys it didn't understand. Because of this flexibility, version 3 of the format will probably never be needed. But Karl could have saved many hours of time and effort by doing things the version 2 way right from the start. (Then again, he seems to be in august company. Microsoft Word's native document format apparently changes quite a bit from one release to the next of the software, to the point where the latest versions of Word actually cannot read some files written by the earliest versions.)

Document Data Structures

When commenting, it's usually better to document the data structures rather than the code that uses the data structures. It might seem counterintuitive to document the "nouns" of a

program before the "verbs," but most people seem to be able to figure out the dynamic portions of the code, once they know the layout and purpose of the data on which it operates. So if you're writing code, put a description by each field in a data structure, directly in the file that defines the structure. When reading code, always look at the header files first. The rest of the code will then usually make much more sense.

Make It Portable

Portability is a big priority—quite naturally, with more participants testing the software in more environments. However, it also means that you'll have to take their word for it when they tell you that a certain way of doing things won't work on their computer. Normally, this is no big deal, but sometimes it can mean substantial changes are required to make the software run in that environment. For example, such a situation might occur if you release a thread-based program that makes use of a thread library peculiar to your operating system. In that case, it simply won't run anywhere else and must be changed to use some version of threading that works on both your machine and on the other person's.

When this happens—and the platform is one you hoped to support eventually, anyway—don't hesitate at all to accept the necessary changes. Every time a new platform is added to the list of where the code can run, you gain everyone who works on that platform as a potential user. This is too good an opportunity to pass up, no matter how fond you were of the original, nonportable implementation.

If your project is somewhat complex and you want it to be very widely used, in the Unix world consider using GNU **autoconf**. This allows you to "abstract out" nonportable aspects of your code and test, at build time on each platform, how each aspect is implemented there. The initial auto-confiscation (yes, that's the right word) of a project can be a bit daunting, but once it's done, many thorny issues are handled for you, because **autoconf** already knows about the most commonly encountered differences between various flavors of Unix. See **www.gnu.org/software/autoconf/autoconf.html** for more information.

The preceding points are more about adhering to certain well-tested conventions than about having any grand architectural vision for the program. But it is about as far as we're willing to go in suggesting how to do robust large-scale design in a world where programs frequently end up with both unexpected uses and unexpected users. Naturally, if you really want to write a program in a certain way, do it—part of the reason for writing software is the pleasure of testing out ideas in running code. However, don't ignore evidence that the design might not be working exactly as expected—with any reasonably complex program, there are bound to be discrepancies between your vision and its realization in software.

If you're set on making your program perfect and beautiful, don't expect anyone else to hack on it. They'll be too afraid of making a mess of your beauty. On the other hand, if your goal is a useful program and not a beautiful one, you can expect contributors to constantly shake up your preconceptions of the project's future. When a lot of developers participate in a

project, the resulting design often ends up representing a compromise, a mixture of styles and solutions, weighted in favor of those who do the most coding. This might seem like rough justice, but it has the advantage of tuning the implementation to those who must interact with it the most—that is, the biggest contributors.

When in Doubt, Abstain

A good principle in open source design is the one of abstaining. Design only what you absolutely must to get the code running or to express a firm belief you have about how the software must be in order to succeed. Leave the rest to evolution. Sure, you might end up with something very different from what you originally set out to produce, but is that so bad? You will be freed from the limits of your imagination, and that's a substantial reward in itself.