CHAPTER 2

Why Software Is Different

When we try to find out what's different about software and software development, the first question that comes to mind is "Different from what?" So let's compare software development to road building. We've all used roads; we know what they're for and how they work. Roads are a very different product from software. In their massive and immobile simplicity, they're as unlike software as it is possible to be.

There are many differences between road building and software development. Software development is rarely affected by the weather, for example, whereas you shouldn't begin excavating a cutting if the slope is soaking wet and subject to landslip. But is this a fundamental difference? No. Software projects are also affected by external events. If a third-party software component isn't available on time, for example, then similar delays can occur.

The following sections introduce 12 distinct but interrelated differences between software development and other common business endeavors (see Figure 2-1). Road building has been chosen as the example to compare against because it displays none of these characteristics, so distinctions can be drawn as clearly as possible. However, this may not be the case for any other activities that come to mind, which might exhibit one or two, or even a few of these characteristics. What makes software development unique is that it encompasses them all.

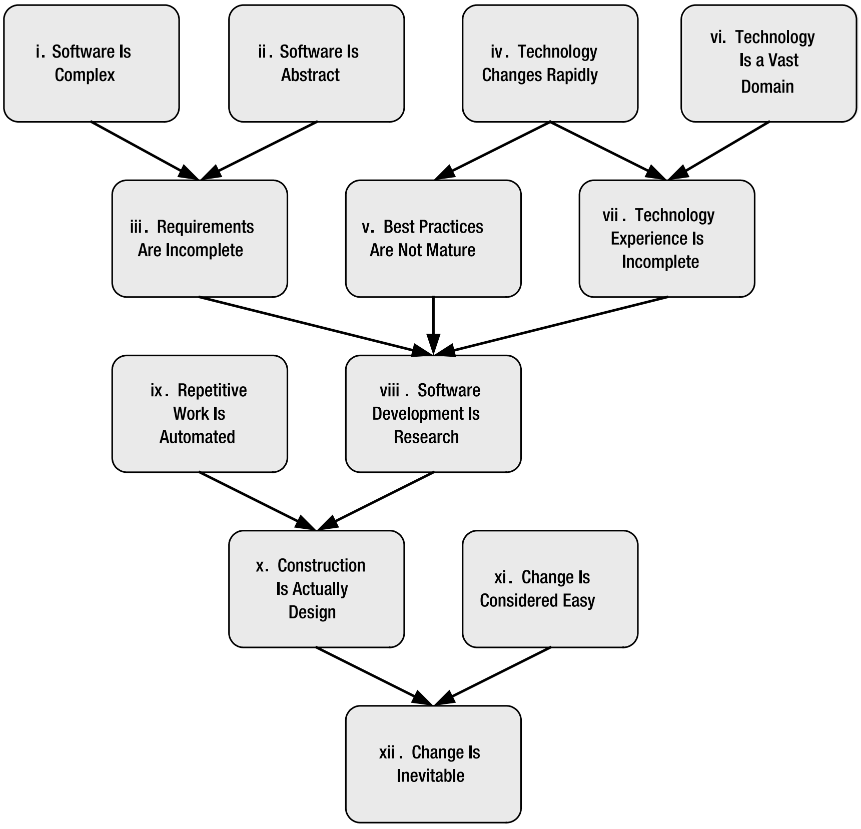
i. Software Is Complex
ii. Software Is Abstract
iv. Technology Changes Rapidly
vi. Technology Is a Vast Domain
iii. Requirements Are Incomplete
v. Best Practices Are Not Mature
vii. Technology Experience Is Incomplete
ix. Repetitive Work Is Automated
viii. Software Development Is Research
x. Construction Is Actually Design
xi. Change Is Considered Easy
xii. Change Is Inevitable

Figure 2-1. The 12 sections in this chapter each introduce one characteristic that is unique to software development, and explain its relationship to those already discussed. The concepts that appear further down on the diagram are derived from the more basic concepts shown at the top.

1. Software Is Complex

“The drive to reduce complexity is at the heart of software development” [McConnell 2004]. Even minor software can accumulate frightening complexity. A small program with one or two authors can easily run into tens of thousands of lines of code. Significant products, like the latest versions of Microsoft Windows, run into tens of millions. But numbers of lines of code may not mean much to you until you can relate that measurement to other types of complex systems.

When you look at software as it's being written, it appears as a sequence of *instructions*. Instructions usually appear as a single line in the text, but very complex instructions sometimes span two or more lines. An instruction may copy a piece of data, perform some arithmetic, manipulate text, or decide which parts of the program to execute (and in what order). There are also blank lines to separate groups of instructions, comments that explain to other programmers what these instructions are intended to accomplish, and elements that help define the structure of the program (*components*, *objects*, *methods*, etc.—please see the Glossary for more details).

But the most important part of the code is the instructions. You can think of an instruction as being equivalent to a moving part in a vehicle. An instruction, just like a moving part, takes a varying input and does something with it.

You might expect that a 100,000-line program would be ten times more complex than a 10,000-line program. However, a program's complexity depends not just on the instructions, but also on the interactions between the instructions. The 100,000-line program has ten times as many instructions interacting with ten times as many instructions, so we should actually expect it to be a hundred times as complex.

Skilled developers endeavor to reduce this overall level of complexity by isolating various parts of system from each other. These are sectioned off into small pieces (called objects) or somewhat larger pieces (called components), which are chunks of code that can be used without knowing exactly how they work. They hide the complexity of their internal mechanisms behind simple interfaces. This technique is known as *encapsulation*, and it is a key part of *object-oriented programming*.

Figure 2-2 shows how it works. This system is composed of 12 items (of some kind) that interact with each other. By dividing these items into four smaller assemblages, the total number of interactions has been reduced from 66 to 18, and the system is now much less complex.

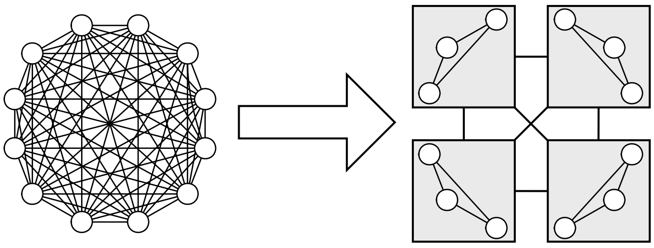

Figure 2-2. Simplifying a complex system by dividing it into smaller pieces

Despite such techniques, developers still find that the complexity of their software increases faster than its size.

Computer programs are the most intricate, delicately balanced and finely interwoven of all the products of human industry to date. They are machines with far more moving parts than any engine: the parts don't wear out, but they interact and rub up against one another in ways the programmers themselves cannot predict. [Gleik 1992]

NOTE Software is unique in that its most significant issue is its complexity.

2. Software Is Abstract

You can't physically touch software. You can hold a floppy disk or CD-ROM in your hand, but the software itself is a ghost that can be moved from one object to another with little difficulty. In contrast, a road is a solid object that has a definite size and shape. You can touch the material and walk the route. Even the foundations, which are hidden when the road is completed, can be viewed and touched as the road is being built.

Software is a codification of a huge set of behaviors: if this occurs, then that should happen, and so on. We can visualize individual behaviors, but we have great difficulty visualizing large numbers of sequential and alternative behaviors.

That's why playing chess well is so difficult. At its simplest level, chess is just a game where 32 pieces move from square to square across a board. We can think of each move as a piece's behavior, but any single move is meaningless in isolation. What gives it significance is its relationship to the moves that have gone before, and to the moves that are yet to come. These relationships are entirely abstract. It's a huge task to accurately assess them, and by doing so draw up a sound strategy. That's why there's such a gulf between novice and expert chess players.

The same things that make it hard to visualize software make it hard to draw blueprints of that software. A road plan can show the exact location, elevation, and dimensions of any part of the structure. The map corresponds to the structure, but it's not the same as the structure.

Software, on the other hand, is just a codification of the behaviors that the programmers and users want to take place. The map is the same as the structure. Once the system has been completely described, then the software has been created. Nothing else needs to be done. We have automatic tools that convert this representation into a program that the computer can execute.

This means that software can only ever be described accurately at the level of individual instructions. To summarize is to remove essential details, and such details can (as we've all experienced) cause the software to fail catastrophically or—worse—subtly and insidiously. But no one can hold 10,000 or 100,000 operations in mind at once.

Even encapsulation, which can reduce the overall complexity of a system, doesn't release us from the burden of having to individually define each and every one of these instructions (or behaviors); it just helps us to organize them better.

A map or a blueprint for a piece of software must greatly simplify the representation in order to be comprehensible. But by doing so, it becomes inaccurate and ultimately incorrect. This is an important realization: any architecture, design, or diagram we create for software is essentially inadequate. If we represent every detail, then we're merely duplicating the software in another form, and we're wasting our time and effort.

NOTE Software is the most abstract product that can be created in a project.

3. Requirements Are Incomplete

Software is normally commissioned for the needs of users and managers, not professional developers. These individuals are experts in their own roles, but they rarely have as much experience as professional developers in dealing with the abstraction and complexity of software. They understand the business processes much better than the developers, of course, but even when someone has a good grasp of the main flows of behavior that are required, it's still very difficult to take into account all of the alternative flows and error conditions, and how different sets of requirements relate to each other.

Moreover, as we saw in the previous section, it is impossible to accurately blueprint software, or draw up a complete set of requirements before the software has been completed in some form or another. This means that any specification of requirements for software is likely to be incomplete in important ways.

The users will gain new insights into their needs as the software starts to take shape. The software becomes less abstract to them once they can get hands-on experience and try out a variety of scenarios. This is where the "steady stream of 'essential' change requests" for the billing system comes from. It's not that users don't know what they want: it's that they're just not able to visualize a system of this complexity until it's at least partially complete.

To be successful, users and developers must work together to refine the requirements for the software. As the software grows in functionality, the users can revise the remaining features based on their testing of the system under construction. An expert can be brought in at any stage to perform usability testing, and to make recommendations regarding the user interface.

All of this suggests that the belief that you have, or ever can have, a comprehensive and finalized set of requirements is a self-deception. The most honest response that a user can give is "I'll know what I want when I see it." However, as we'll see later, this consideration is rarely taken into account.

Does this matter? The Standish Group [2001] identified problems with requirements and specifications as the top "Project Challenged Factors" for software projects. They were the most significant issues for 42 percent of projects.

In contrast, drawing up the specifications for a new road is a relatively straightforward process: only the route, number of lanes, intersections, surfacing, and so on need to be defined. Any driver can understand and recognize the value of these characteristics.

NOTE It is uniquely difficult to define a complete set of requirements for software before beginning development.

4. Technology Changes Rapidly

Twenty years ago we were struggling with the MS-DOS operating system and creating simple spreadsheets on our PCs. Today we edit video on our computers and connect to systems across the world. Computers double in speed about every two years, and this opens up more and more opportunities for software developers. Software changes quickly—we all know that—but we may not be aware of just how quickly it changes, and what impact this has on any new software we try to build.

Nowadays any significant new software is almost certain to be built with an *enterprise application framework* such as Sun Microsystems' Java 2 Enterprise Edition (J2EE) or Microsoft .NET. It's important to understand just what this phrase means, because these technologies largely define the software development landscape as it stands today:

- A framework is a toolkit, just like a Lego set, that you can use to build a variety of items. In the case of software, the building blocks are bits of software that do jobs that have been found useful in a wide range of situations. Examples include getting data from a database, drawing a window on the screen, or converting dates from one format to another.
- The word "application" includes more than you might imagine. There are the one-person desktop applications that we're all familiar with, such as Microsoft Word for word processing, but there are also multiuser applications that run on an office network, such as accounting and email. Beyond that are applications we use over the Internet, such as Amazon.com's online ordering and Google's search page, and applications that other applications use to exchange information, so that international phone calls can get connected, for example.

- "Enterprise" is the most difficult word to define. Perhaps the best way to think of it is "as big as you want." Desktop applications are limited to running on one computer, but that's OK because only one person is using them at a time. The popular Internet search engine Google provides information to more than 1,000 people every second: no single computer could handle that load. Enterprise technology allows many computers to work together for a single application, and also provides the connectivity to allow lots of people to access it at the same time. But "enterprise" also means "as small as you want": enterprise application frameworks are not just for major applications in big companies.

Given that some of the most important government and business software is now being built with these enterprise application frameworks, you might expect that they have a long and distinguished history, and that they'd be stable and mature products. That's not the case. Sun's J2EE, which was perhaps the first true enterprise application framework to be widely used, appeared in 1998, and has seen considerable change since then. Microsoft only released its competing technology (.NET) in 2002, and no one has more than a couple of years of experience with it yet.

In contrast, we've been building roads for thousands of years, ever since the time of the ancient Roman and Chinese civilizations. The problem is well understood, and the technologies change slowly. Hot-mix asphalt was patented in 1903, and that basic technology is still what we use today.

NOTE Software development technologies change faster than other construction technologies.

5. Best Practices Are Not Mature

Technologies can be used skillfully or unskillfully. For software, this distinction can often only be assessed some time after the software has been completed. The presence or lack of software quality shows up most clearly in its *extensibility*.

Extensibility is the ability to add functionality or modify existing functionality without impacting existing system functionality. You cannot measure extensibility when the system is deployed, but it shows up the first time you must extend the functionality of the system.
[Cade and Roberts 2002]

Most programmers have had the painful experience of trying to modify a system that works well, but in which it's virtually impossible to make changes without breaking some of its functionality. Programmers call this *fragile code*.

Fragile code can have a big financial impact. Numerous studies have shown that at least 50 percent of software cost goes into extending and modifying the original system [Koskinen 2004], and modifications to fragile code can be twice as expensive as modifications to robust and flexible code. Clients need solutions that can change and grow along with them.

Code becomes fragile when it's put together in an ad hoc fashion, without sufficient attention being paid to its architecture. An architecture is the overall structure and design of a system, and can be seen as a codification of how to use the technologies that the system is built upon. New technologies need new architectures. For example, when Microsoft introduced "event-driven programming" to BASIC in 1991 via its new Visual Basic development environment, it provided powerful new capabilities, but also the potential for new problems.

One of these problems was a poor design practice, which became so prevalent that it ended up with a name of its own: the Magic Pushbutton. In event-driven programming, all a programmer does is write a few "event handlers," which are routines that respond to the user's actions. This technology means that instead of having to write the core functionality for each new program over and over again, a programmer can just add functionality to an application skeleton that's provided for them.

In the Magic Pushbutton, the only event handler that does any real work is the one that's called when the user clicks the OK button. If the programmer doesn't deliberately organize the program's code in a better way, then it all accumulates in this one routine, which ends up as a huge, unmanageable blob of code.

Over time, every field of human endeavor develops best practices to counter common mistakes like this. A best practice is a process or technique that has a proven record of success in providing significant improvement to the results of an activity. Experience allows users to define the best practices—in other words, the most consistent ways to use the technology well.

But how long should we wait for those best practices? Object-oriented programming has been in use since 1980, but it was only in 1995 that the "Gang of Four" (Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides) published their seminal book *Design Patterns*, which revolutionized object-oriented architectures and provided solutions to "anti-patterns" such as the Magic Pushbutton. Fifteen years is a long time in the IT industry: many technologies have come and gone in less time.

The enterprise application frameworks we've discussed have been around for only a fraction of that time. A few architecture books are available already, but none of them are as significant as *Design Patterns*. This suggests that the enterprise application frameworks' best practices aren't mature yet.

So is software based on new technologies necessarily poor? Fortunately, no. Later sections in this chapter will show how robust software can be created even in these circumstances, but it's not as simple as in road building, where the basic technologies have been around for much longer, and hence the best practices are long established and almost universally applied.

NOTE Most software development technologies are not mature enough to have a set of proven best practices.

6. Technology Is a Vast Domain

"No man is an island, entire of itself." Nor does any piece of software exist in isolation. We've seen how the building blocks for any new software would come from an enterprise application framework, and they would implement the most commonly used tasks that the software performs. These tasks are many and varied, and it comes as no surprise that the enterprise application frameworks themselves are huge and complex, containing tens of thousands of usable routines. You can think of each routine as a knob, button, or dial that allows the application—and the developer—to control the workings of the framework, although there are also some routines that behave more like standalone tools.

However challenging it is to develop software using an enterprise application framework, it's still much simpler and quicker than attempting to create your own versions of these tools from scratch. It's always cheaper to buy than to build, which is why a substantial marketplace has sprung up for software components that are more specialized than those available in the enterprise application frameworks.

Until recently, these software components were distributed on CD-ROMs, and copied into any application that required them. Since then, *web services* have become the preferred technology for accessing third-party software components. For all the hype surrounding web services, they are in principle very simple. Instead of selling CD-ROMs, a third party keeps their software components on their own servers and makes them available over the Internet for a fee. This makes it easy for the vendor to apply bug fixes and enhancements, and enables them to provide access to interesting data (stock quotes, news, retail price comparisons, etc.) as well as useful tools.

Enterprise application frameworks contain a wealth of functionality, and beyond that there is even more functionality available from third-party software components. A single application would use only a few of these routines and components. Even when working on a variety of applications and systems, a developer will not be able to gain experience with more than a small proportion of the technologies that are available.

NOTE Software development has far more technologies, and its technologies have far more complexity than a single individual can hope to gain expertise with.

7. Technology Experience Is Incomplete

We've already seen that software technologies change rapidly. New technologies supplant older ones every few years, and even more frequently, new versions of existing technology appear that radically change the functionality and use of that technology. This change is necessary and inevitable.

For example, Microsoft's BizTalk Server 2004 is a very different product from BizTalk Server 2000. Some of the tools work like they did in the previous version, but the majority are entirely new or have changed beyond recognition. "The core purpose of BizTalk Server remains, but Microsoft has redesigned or enhanced almost everything around that core" [Yager 2004].

Moreover, developers work with an enormous range of specialized third-party software components. Experience with these components can rarely be carried over into future projects, because those future projects are unlikely to use the same third-party components. Experience with enterprise application frameworks is similar; these frameworks are so extensive that different projects that use the same framework may well use totally different parts of it.

Whatever a developer was working on even three to four years ago is unlikely to be of any direct use today. So what use is an experienced developer? Is it true that every significant new piece of software is written by developers who are essentially novices to the task?

It's true that long lists of desired technology skills, which are so prevalent in IT job advertisements, are virtually useless. The bulk of the technical knowledge required for a project will normally be learned on the job. However, the "softer" skills that make one a good developer, or even a good team leader or architect, do apply from one project to the next, and can accumulate over time. These skills include the software development best practices that are discussed later in this book. Sadly, these skills are rarely mentioned in job ads. They're harder to assess: you can't just boil them down to a list of buzzwords and acronyms.

NOTE Expertise with particular software development technologies is very quickly outdated, and therefore most specific skills are learned on the job.

8. Software Development Is Research

As noted in previous points, the requirements for a piece of software will invariably be incomplete. There will be conceptual gaps that must be filled, and there will be assumptions that aren't justified and aspects that just won't work. Because clients aren't software experts, they won't always be able to distinguish between what's possible and what's not, or know what trade-offs are available. They need to work with the developers to discover this.

This means that the development process is a process of discovery—progressively finding out the exact character of the software that will meet the customer's needs. Developers must combine analytical and creative skills to find out what their customer really wants (even if the customer's description is confused and incomplete) and invent ways to combine these requirements into a system that's logically consistent and easy to use.

The project will probably include software tools and components that are new or unfamiliar to the developers. New technology doesn't always match up to the marketing claims that are made for it. The rate of change in technology has been accelerating, and products are often released before they're mature, complete, or bug-free. The advantages of being first to the market can outweigh the drawbacks of proffering flawed software.

For these reasons, users need to be wary of "version 1.0" software: software that has just been released for the first time. Many companies "get it right" only on the second or third attempt. Just think about Windows 3.1 and Internet Explorer 3.0: who remembers any of their earlier versions?

These issues are particularly relevant for beta software. Traditionally, after in-house testing is completed, new software would be given to a small group of carefully chosen customers for *beta testing* in real-world conditions. This would uncover yet more bugs, which were fixed before the first general release. Recently, however, some software publishers have begun selling beta releases to their customers. This means that the first released version is known to contain bugs, and that customers are expected to pay for the privilege of testing it and finding the bugs. Other publishers do the same, but don't even mention that the software is beta.

We can't take it for granted that a given software tool or component will work as we expect it to, or do everything that we need when we use it to create our software. Even if the product chosen is mature and well regarded, and even if the developers have used it before, because of the complexity of software, you can rarely be sure that it can be used for the functions and circumstances that are unique to a particular project. You can't tell if it will do the job until you've actually made it do the job, and have seen that it works.

So software development is also a process of discovering whether and how the chosen technology can be made to perform the role that's required of it. Sometimes it will work as expected. Sometimes it won't, but there's a workaround that takes more effort than originally planned. And sometimes the technology just can't do what's needed. Software projects rarely run smoothly from beginning to end. They frequently encounter setbacks and dead ends, simply because of the scope of the unknown. In a very real sense software projects are simply the process of discovering the unknowns: once the unknowns are known, then the project is effectively at an end.

NOTE Software development isn't just a process of creating software; it's also a process of learning how to create the software that is best suited for its purpose.

9. Repetitive Work Is Automated

Automation is the automatic operation and control of a piece of equipment, or of an entire system. The use of automation began during the Industrial Revolution in the eighteenth century, and hasn't let up any time since then. We expect productivity gains in every industry, and software and road building are no exceptions.

In some industries, manufacturing is done in factories that are empty of workers except for a few maintenance staff, and production has been completely automated. But we're discussing projects here rather than production, and projects can never be wholly automated. The Project Management Institute [2000] defines a project as "a temporary endeavor undertaken to create a unique product or service." It's this uniqueness that makes complete automation impossible. No road is just like another, and even a piece of software that duplicates the behavior of another must be made in a unique way. This is called a "clean room" implementation.

No matter how much labor is saved through the use of machinery in road making, road workers must still do a substantial amount of repetitive work. Asphalt must be laid and rolled. Median barriers and lane markings must be installed.

All of the repetitive work can be automated in software development, and that's because software doesn't exist in the real world. It resides in the controlled environment of the computer. Every part of it can be created and controlled by means of a wide range of software tools.

Common tasks and services are included in enterprise application frameworks, and more specialized ones can be done by third-party software components, so programmers work more efficiently because much of their work has already been done for them. But even beyond that, tools are constantly being developed and refined to automate new chores and responsibilities.

One example is in web services. Web services use the industry-standard messaging language Extensible Markup Language (XML) to communicate messages over the Internet. To create a web service, you must define the message format for every possible type of message, to show how the data is to be converted into XML. This is a labor-intensive chore. But not long after the introduction of web services, Microsoft automated this step in its Visual Studio .NET development environment. Developers no longer have to put any effort into defining the XML message formats for their web services, because this work is now done for them by their development tools.

NOTE Software development has been automated to a greater degree than other project-based activities.

10. Construction Is Actually Design

Road building consists of a sequence of well-defined phases. The first step is to perform the planning and design, which results in a set of plans and blueprints that can be signed off. Once these tasks are completed, then construction can start. The construction phase largely consists of well-defined, repetitive tasks that can be accomplished by less highly skilled workers.

In contrast, software development is a process of research, so at no point can definitive plans be drawn up. The more definitive you try to make the plans, the more flawed they'll be, and the more labor there will be in revising them to meet changing circumstances. As the shape of the software becomes increasingly clear over the course of a project, the design of that software must be revised again and again. To completely redesign the solution each time around would be onerous and wasteful, so we should think rather of a process of ongoing design.

We've seen that the repetitive work in software development is rapidly automated. There aren't any repetitive tasks to define. But if tasks aren't repetitive, then defining them exhaustively becomes a time-consuming process. And there is little to be gained from defining tasks in this way. Software is abstract, so defining the construction tasks completely is equivalent to actually performing the work, because automated tools are available that can turn such designs into working software.

If tasks can't be well defined, then we can't cleanly separate the design and construction phases. Indeed, there's no construction as such; there's only design on smaller and smaller scales. This means that we can't easily categorize people into the roles of architect, designer, programmer, or analyst. The roles overlap. All developers have the same kinds of tasks to perform, even though some may have more responsibilities than others.

When a developer creates a new feature in a piece of software, their task is simply to answer the question "exactly how is this feature going to work?" They will add a set of instructions to the source code that defines in every detail how the feature will work. But each detail is a design choice. For example, a piece of text can be stored in an unchangeable object for greater efficiency, or in a changeable object for greater flexibility. Which option is chosen depends on how that piece of text is used. It's a significant decision.

Programming is more than just writing code. Each step requires the developer to analyze some portion of the problem and design some aspect of the solution.

NOTE Unlike other products, software is not constructed, but rather designed into existence.

11. Change Is Considered Easy

Last-minute changes to requirements are rare in road building because the consequences are so severe. If you discover during the course of a project that the foundations are in the wrong place, then it takes considerable effort to dig them out and rebuild them in another place. This is obvious to clients and contractors alike.

Once you've built a section of road, then it's built. A road may be extended or widened, but it is never moved. When a road is "realigned," what happens is that a completely new section of road is built alongside the old, which often remains as an alternative route. A freeway interchange may be reworked over several years to suit changing traffic patterns by the addition of permanent bridges, ramps, and lanes in a series of projects.

This isn't to say that changes never occur in civil engineering projects. When you start digging, you may find that subsurface conditions are different from the original analysis. Subcontractors might not be available when you want them, and schedules may be adjusted accordingly. However, such changes rarely have an impact on the nature of the product: we don't expect to end up with a different building or a different road.

Compare this to software. Software is soft, by definition. Any part of it can be changed at any time, just by rewriting that portion of the code. We expect that bugs can and will be fixed wherever they appear in the system, as indeed they are. Anyone who has written macros for Microsoft Office, or learned how to write small programs at school or at a university, knows how flexible software is and how quickly you can make substantial changes.

It's true that substantial changes can often be made quickly and easily, but to properly implement them you really have to revise the architecture of the software so that it gracefully supports the new functionality; otherwise you'll just create a mess and make the software more fragile.

The architecture must be flexible and designed to accommodate change. This is the main yardstick that we have for an architecture. If the software will never be changed, then why should we care if the software is fragile and badly designed as long as it still works? But most major systems are intended for long use—often decades—even if their underlying technologies rapidly become obsolete. Over such long periods of time, the business environments will almost always see significant change.

In addition to designing the architecture, we must also design our development process to support change. This subject will be covered in much greater detail in later chapters.

NOTE Software can be modified rapidly, and this pace is expected, but it's better to implement the changes properly.

12. Change Is Inevitable

Are there any situations where there will never be changes to the software? We've seen how software development is actually a process of design from beginning to end. It includes design work to accommodate requirements whose details become clearer as the project progresses, and design work to reflect what's learned about the tools and components used to develop the software. The process of software development is one of continuous design, and therefore of continuous change.

Moreover, clients see how easily changes can be made, and expect that they can change their minds at any point. Indeed, they often do, as they learn more about what their nascent software can achieve for them. In 8,000 large software projects analyzed in one study, some 40 percent of their requirements arrived after development had begun [Jones 1995].

Change is inevitable, and if a piece of software isn't built to support change then it will fall apart even as it is being built. The quality of software shows itself when the software is first extended or modified. If the process of development becomes one of extension and modification, then any software that resists change will have a difficult gestation. Poor changes are more likely to generate defects or bugs, and will make the code fragile and hard to debug.

It's easy to see the problems that change can bring to a project, and begin to see change as the "enemy," but is attempting to eliminate all change a viable option? Once we see change as inevitable, then the issue isn't one of avoiding change but of making change work to our advantage. This is a much more manageable problem, as we'll see later.

NOTE No software is perfect as first envisioned; it will always require changes to make it best suit its role.

Summary

In this chapter we've taken a whirlwind tour of the software development landscape to set the scene for the rest of the book. We've identified 12 key characteristics of software development that make it unique. The next chapter will use these concepts as a starting point. We'll be performing an in-depth analysis of project management to discover which kinds of activities it is suited to (and which it is not), and we'll be comparing this picture to the view of software development that we developed in this chapter:

1. Software is unique in that its most significant issue is its complexity.
2. Software is the most abstract product that can be created in a project.
3. It is uniquely difficult to define a complete set of requirements for software before beginning development.
4. Software development technologies change faster than other construction technologies.
5. Most software development technologies are not mature enough to have a set of proven best practices.
6. Software development has far more technologies, and its technologies have far more complexity than a single individual can hope to gain expertise with.
7. Expertise with particular software development technologies is very quickly outdated, and therefore most specific skills are learned on the job.
8. Software development isn't just a process of creating software; it's also a process of learning how to create the software that is best suited for its purpose.
9. Software development has been automated to a greater degree than other project-based activities.
10. Unlike other products, software is not constructed, but rather designed into existence.
11. Software can be modified rapidly, and this pace is expected, but it's better to implement the changes properly.
12. No software is perfect as first envisioned; it will always require changes to make it best suit its role.