Ten Commandments of Successful Software Development

By: Harris Kern's Enterprise Computing Institute

While not inscribed in stone tablets, these rules provide detailed suggestions on how to streamline and improve your software development organization for greater success.

All great organizations have a vision, a mission, and elemental guidelines for proper behavior that are infused into their people. Perhaps the oldest code of conduct is the Ten Commandments. Since the day Moses stumbled off Mount Sinai, people have applied the idea of condensing their groups' rules into 10 easy-to-remember sentences. Following the lead of Moses, we've distilled the successful software development concept into 10 commandments. If you want to be consistently successful at software development, be sure you always embrace these 10 ideals:

I. **Thou shalt start development with software requirements.**
II. **Thou shalt honor thy users and communicate with them often.**
III. **Thou shalt not allow unwarranted requirements changes.**
IV. **Thou shalt invest up front in software architecture.**
V. **Thou shalt not confuse products with standards.**
VI. **Thou shalt recognize and retain thy top talent.**
VII. **Thou shalt understand object-oriented technology.**
VIII. **Thou shalt design web-centric applications and reusable components.**
IX. **Thou shalt plan for change.**
X. **Thou shalt implement and always adhere to a production acceptance process.**

I. Thou Shalt Start Development with Software Requirements

Every morning, some software developer wakes up with a great new idea for a software application, utility, or tool. Those who go off and start writing code without first considering the requirements for their program are likely doomed to failure. Invest up front in developing your software requirements, and you'll be starting down the path to successful software development. A software development organization without any requirements-management process in place will not achieve repeated development success. Here are some tips as to why and how you should develop and manage software requirements for any project, regardless of size.

For starters, if you can't define the requirements for your software system, you'll never be able to measure the success of your development effort. If you were to write a calculator program, would it be successful if it could add two numbers and produce the correct result? What about subtraction, multiplication, and division? Does the calculator need to handle floating-point numbers or just integers? How many digits of precision are needed in results? What should the calculator's behavior be if a "divide by zero" error is encountered? Should results be displayed in a textual or graphical format? How long does the result have to be saved after it's displayed? The list goes on. Even in this trivial example, requirements are extremely important to determining the success of the project.

It's very difficult to write good software requirements. Without good requirements that state precisely what a software program is suppose to accomplish, it's difficult to complete the project, much less judge the application's success. One of the main reasons that it's hard to write good software requirements has to do with the nature of human language. Spoken language is very imprecise and leaves much to be inferred by the listener or reader. Computers, by their digital nature, are very precise and not easily programmed to infer missing requirements. Therein lies a dichotomy. Think of a requirements statement as simple as "the program will add two numbers and display the results." You could raise all the same questions posed in the calculator example in the last paragraph. Even if these questions were answered, more requirements would probably be uncovered during development of the application.

If your software development team asks questions like those in the calculator example, it's probably a good sign. There is no surer road to failure of a software development project than incomplete requirements. Of course, the next steps after asking questions and developing requirements are to documenting, organizing, and tracking functions that help you do this.

Many development projects actually start with a good set of functionality requirements: input this, perform this processing, and output that. Often left out, however, are performance and other environmental requirements as summarized in Table 1. How quickly does the program have to complete the required processing?

Table 1: Commonly Overlooked Performance and Environmental Requirements

Requirements	**Example(s)**
Processing speed	CPU speed
Memory capacity	Cache size, RAM size
Network capacity	Network interface card speed, switch/router bandwidth
Persistent storage	Online disk capacity; tape backup capacity
Internationalization support	Application deployed in multiple countries or languages
Display	Minimum monitor size and resolution; number of colors supported
Financial issues	Budget and schedule
Environmental issues	Power requirements, air conditioning, special temperature or humidity requirements

How much RAM or disk space can it use? Does the software have to support internationalization for worldwide use? What's the minimum display size required?

Environment-related requirements are becoming increasingly important, especially Java. Java truly provides a write-once-run-anywhere environment, ranging from smartcards to workstations to mainframes. A Java applet that looks great on a workstation's 21-inch color monitor, however, may look much different on the 4-inch screen of a monochrome personal digital assistant (PDA). Finally, don't forget budget and schedule requirements.

In gathering these requirements, the reasons for the second commandments should become very clear.

II. Thou Shalt Honor Thy Users and Communicate with Them Often

Software developers don't generally use the applications they create; the intent is for the product to be used by customers, clients, and end users. This implies that someone in your development organization had better spend a lot of time communicating with users so that their requirements can be correctly understood. In the calculator example discussed earlier, a developer may be perfectly happy with the four basic arithmetic functions, while the user would like a square root function or a memory function. What the developer thinks is a complete set of requirements isn't necessarily so.

Moving from a trivial to a real-world example, an enterprise-wide IT application may have many types of users, each with his or her own business requirements. Take a payroll system. An employee is one type of user, whose requirements include being paid the correct amount in a timely fashion. A manager wants to be able to administer salary increases and track budgets. The HR administrator wants to compare salary ranges across an entire organization. Each type of user has unique requirements.

The second part of this commandment focuses on the word "often." Communication with users only *starts* at the requirements-definition phase. A developer may have to discuss a requirement with a user several times before the true definition of the requirement is captured. In addition, user requirements are likely to change often, and indeed several requirements may even conflict. Frequent communication with users gives the developer early notice of requirements changes that the user is considering.

A successful software development organization has established processes for frequent communication with users during all stages of the development process. The sooner an incorrect or missing requirement is discovered, the easier it is to fix the problem. Many successful development organizations have made customer advisory teams an integral part of the software development process. Such customer teams participate in all stages of development from initial requirements-gathering to production acceptance signoff. The Web-Centric Production Acceptance (WCPA) process presented in our book, *Software Development: Building Reliable Systems*, by Marc Hamilton is another vehicle for bringing users and developers together and promoting and instilling good communication practices.

III. Thou Shalt Not Allow Unwarranted Requirements Changes

While user requirements often change, it's the job of the development organization to manage these changes in a controlled fashion and assure that requirements don't change or grow unnecessarily. Pity the poor programmer who started off writing a basic arithmetic calculator, agreed to add square roots and a few other user-requested functions, then a few more, and soon had the task of developing a 65-function scientific calculator. Modifying or adding new requirements can happen for many reasons, not the least of which is the failure to honor commandments 1 and 2.

Another reason requirements grow beyond their original scope is simply that software is so easy to change, compared to hardware. No one would purchase a calculator at the local electronics store and then expect the manufacturer to add additional transistors to the calculator chip to implement a new function. But if the calculator is a software application, no one would think twice about the ability to add a new function.

Perhaps an even more common reason that requirements change is due to the programmers themselves. Many a programmer has accepted a new user requirement not based on valid business reasons, but simply to please the user. At other times, the software may be fully functional and pass all unit test requirements, but the programmer just wants to add one more feature to make the application "just a little better." Good developers always want their code to be perfect in their own eyes. The cost of making even simple changes, however, is minor compared to the cost of the retesting and re-qualifying that may result.

This doesn't mean that we oppose iterative development. Even in a spiral development model, however, new requirements are introduced at the start of new iteration, not continually during the development process. Requirements changes affect project budget and schedules. Allowing changes only at set points in the software lifecycle allows time to trade off the value and validity of each new proposed requirement against its cost. Meanwhile, developers can complete each stage with a frozen set of requirements, speeding the total development cycle.

Object-oriented and component-based software technologies help further isolate the impact of many requirements changes. Still, requirements changes are a constant problem for many software projects. Luckily, managing requirements is mainly a process issue rather than a technical one. Here are some ways to help prevent requirements from constantly expanding beyond their original scope:

- Document all user requirements, allowing users to review the completed requirements document and agree to its completeness.
- Get users to agree up front that future requirements changes will be accepted only after being evaluated for schedule and budget impact.
- Practice iterative development. Get users and developers to understand that the first version is not the final version. That "one last change" can always wait for the next version. Many a software system has suffered unexpected delays because a simple last-minute change rushed through just before release broke huge parts of the system.

IV. Thou Shalt Invest Up Front in Software Architecture

Every morning, some developer goes to work with software requirements for a new application in hand and starts writing code. For those who do so without first developing software architecture, their programs are likely doomed to failure. Invest up front in your software architecture, and you'll be starting down the path to successful software development.

It's essential as having a blueprint for construction of a large building. The architecture becomes the blueprint or model for the design of your software system. We build models of complex systems because we can't comprehend any such system in its entirety. As the complexity of systems increases, so does the importance of good modeling techniques. There are many additional factors to a project's success, but starting with a software architecture backed by rigorous models is one essential factor.

In the face of increasingly complex systems, visualization and modeling become essential tools in defining software architecture. If you invest the time and effort up front to correctly define and communicate software architecture, you'll reap many benefits including these:

- Accelerated development, by improved communication among various team members
- Improved quality, by mapping business processes to software architecture
- Increased visibility and predictability, by making critical design decisions explicit

Here are some tips on how and why to always start your software development project with software architecture.

Start with a minimum yet sufficient software architecture to define the logical and physical structure of your system. Table 2 summarizes some sample activities to be performed.

Software architecture is the top-level blueprint for designing a software system. Developing good software architecture requires knowledge of the system's end users, the business domain, and the production environment, including hardware and the other software programs with which the system will interface. Knowledge of programming languages and operating systems is crucial to developing good software architecture. As software systems grow more complex, more knowledge is required of the software architect. Object-oriented and component-based technologies may simplify individual programs, but the complexity typically remains at the architectural level, as more objects or components and their interaction must be understood.

Table 2: Software Architecture Activities

Activity	Architecture Level	Examples
Gather user requirements	Logical architecture	Generate use-case examples Document sample user activities Create class, state, sequence, and collaboration diagrams

Start design and production acceptance	Physical architecture	Define packages and components Define deployment environment

There are no shortcuts to designing good software architecture. It all starts with a small number of software architects, perhaps one to three. If you have more than three architects working on a single program's software architecture, they probably are not working at the right level of detail. When software architecture delves too deeply into detailed design, it becomes impossible to see the whole architecture at a top level and properly design from it.

Most software applications are much more complex than the makers of GUI development tools would like you to believe. Every application should be built around a software architecture that defines the overall structure, main components, internal and external interfaces, and logic of the application. Applications that work together in a common subsystem should adhere to an architecture that defines the data flow and event triggers between applications. Finally, applications that run across your entire organization need to follow some set of minimal guidelines to assure interoperability and consistency between applications and maximize opportunities for reuse of components.

Software architecture should always be designed from the top down. If you're going to implement a multiple-tier software architecture across your IT organization, it's nice to do this before you have lots of components written that can only communicate with other applications on the same host. Start by developing your organization's overall application architecture. Next, define the system architecture for the major systems that will be deployed in your organization. Finally, each application in every system needs to have its own internal software architecture. All of these architectures should be developed before you develop any production code or invest in any purchased software packages. The notion of completing software architecture up front doesn't contradict the spiral model of software development that utilizes prototyping and iterative refinement of software. Rather, prototyping should be acknowledged as an essential step in defining many parts of your architecture.

Trying to design a global set of every reusable component you think you might ever need is a losing proposition. You only know which components will be useful with lots of real experience delivering systems. If you don't prototype, you don't know whether what you're building is feasible, useful, and tractable.

V. Thou Shalt Not Confuse Products with Standards

A common mistake made by IT organizations is to confuse products with standards. *Standards* are open specifications such as TCP/IP or HTML. Standards can either be de facto or official. *De facto standards*, while not necessarily endorsed by any standards body, are widely accepted throughout an industry. *Official standards* are controlled by standards bodies such as the IEEE or ISO. Products can implement specific standards or they may be based on proprietary protocols

or designs. Standards, because many vendors typically support them, tend to outlive specific products.

If your IT organization chooses to standardize on a product, say Cisco routers for network connectivity, you shouldn't do so until you first settle on a standard protocol for network connectivity, such as TCP/IP. Here are some common mistakes IT organizations make when defining their application, system, and software architectures:

- **The application architecture is defined at too high a level.** Some CIOs make the mistake of declaring Windows (or UNIX, or mainframes) their corporate application architecture for an IT organization. Even the various third-party programs designed for Windows (or any other operating system) don't define all the characteristics of how to run a business. This is not to say that a corporation might not standardize on Windows and use it wherever possible in its IT infrastructure only that application architecture requires a finer granularity of detail. In general, application architectures should not be so specific as to be tied to particular products.

- **The application architecture is defined at too low a level.** Oracle Version XX is not application architecture it's a specific version of a vendor's database product. Again, application architectures should not be product-specific. A better architecture phrase would be something like this: "Relational databases that implement the SQL standard." This may not preclude a company from deciding to purchase only SQL DBMS systems from Oracle, but specific product choices should be made only after the underlying standards decision has been made.

- **System architecture doesn't address how the system is going to be tested.** Many software projects have wonderfully elegant architectures (from a computer science perspective) that result in projects that fail miserably because no attention was ever paid to how the system was going to be tested. One of the most commonly overlooked test factors is performance testing. System architecture must take into account how a system is going to be fully exercised and tested. This is especially relevant when designing multiple-tier applications. For instance, in a three-tiered system, the architecture may allow for individual testing of components in each of the three tiers, but may not allow for end-to-end system testing to verify the correct interoperation between all three tiers. An equally bad architecture allows for end-to-end testing without allowing for testing of components in each individual tier. There are few worse plights than to know that your whole system isn't operating successfully and have no way to isolate which component is causing the problem.

- **Software architecture doesn't consider production rollout of the application.** In addition to taking into account how an application will be tested, the process of rolling out an application into production needs to be considered in your software architecture. Many great systems have been designed that were never fielded because the infrastructure to support their widespread use was not available. The Web-Centric Production Acceptance process discussed earlier specifically addresses the production rollout process for web-centric applications.

VI. Thou Shalt Recognize and Retain Thy Top Talent

Too many software development topics concentrate on technology or management practices. At the end of the day, a lot of your success in software development will come down to who you have working for you and how happy they are in their work. You can have the best technology and organize your team in the best way possible, but if you have the wrong team it won't do you much good. Another surprise, given the effort so many organizations put into recruiting talent, is their failure to recognize and *retain* that talent after they join the organization.

Organizations that are successful at retaining their top talent start by recognizing their top talent. Plenty of metrics can be gathered around a developer's productivity and quality. Don't forget, however, the more subjective criteria summarized in Table 3. Who are the developers who always show up at other code reviews and make constructive comments? Who is known as the "go to" person when you have a tough bug to find? Which developers really understand your business? Who has the best contacts with your customers? Be sure not to concentrate 100% on hard metrics and overlook factors such as these. Once you know which developer(s) you want to keep around, go about thinking of ways to make that happen.

Table 3: Traits of Successful Developers

Skill Dimension	**Trait**	**Example(s)**
Technical	Operating system knowledge	Understands operating system (OS) principles and the impact of OS on code design choices
	Networking knowledge	Understands networking infrastructure and matches application network demands to available infrastructure
	Data management	Understands how and when to use databases, transaction monitors, and other middleware components
	Hardware	Knows the limits of what can be accomplished by the software application on various hardware configurations
Business	Understands the business	Differentiates between "nice to have" requirements and those that are essential to the function of the application
	Market knowledge	Keeps up to date on developer's tools, frameworks, and hardware
Professional	Written and verbal communication	Effective presenter at code reviews; documentation is easy to read and understand

	Teamwork	Participates actively in other developers' code reviews
	Flexibility	Can work well on a wide variety of coding assignments
	Reliability	Completes assignments on time; strong work ethic
	Problem-solving skills	Viewed as a "go to" person for help in solving difficult software bugs

Developer skill, more than any other factor, is the largest single determinant to the outcome of successful software projects. This is reflected in software costing and scheduling models, most of which place higher weighing factors on developer skill than all other factors, including project complexity. In other words, skilled developers working on a complex software development project are more likely to produce a successful application than less-skilled developers working on a simpler project. Studies have shown that top developers can be 2 - 3 times more productive than average developers and up to 100 times more productive than poor developers. This wide range of productivity and its standard deviation is higher than for any other profession. Good developers not only produce more lines of code their code has fewer errors, and is of higher general quality (performs better, is more readable, is more maintainable, and exceeds in other subjective and objective factors) than code produced by average or poor developers.

One false belief we've heard from many software development managers, especially those without a development background, is the notion that as development tools become more advanced, they "level the playing field" between average and great developers. Anyone who has ever attended a software development-related convention has seen slick demonstrations showing how "nonprogrammers" can use a development tool to quickly build an application from scratch. Modern development tools, especially the integrated development environment (IDE) that addresses everything from requirements definition to testing, certainly help developer productivity. This is especially true in the area of graphical user interface (GUI) code. Even with the best of IDEs, however, there remains a highly intellectual component to software development. The best software requirements, the best software architectures, and the most error-free code continue to come from the best software developers and software architects.

Rather than leveling the playing field, our experiences have shown that good IDEs, used as part of the development process, increase rather than decrease the difference between average and great developers. There is often a compounding effect as an unskilled developer misuses a built-in function of the IDE, introducing bugs into the program, while never gaining the experience of thinking out the complete solution. Of course, we don't oppose the use of good IDEs or the concept of code reuse. It's just that neither of these is a substitute for developer skill.

VII. Thou Shalt Understand Object-Oriented Technology

Every key software developer, architect, and manager should clearly understand object-oriented technology. We use the term *object-oriented technology* versus *object-oriented programming* because one doesn't necessarily imply the other. Many C++ and Java programmers develop in an object-

oriented programming language without any in-depth knowledge of object-oriented technology. Their code, apart from the syntax differences, probably looks very much like previous applications they've written in FORTRAN, C, or COBOL.

While object-oriented technology is not a panacea for software developers, it's an important enough technology that the key engineers in every development organization should understand it. Even if your organization doesn't currently have any ongoing object-oriented development projects, you should have people who understand this technology. For starters, without understanding the technology you'll never know whether it's appropriate to use on your next project. Secondly, due to the long learning curves associated with object-oriented technology, organizations need to invest in it long before they undertake their first major project. While object-oriented programming syntax can be learned in a few weeks, becoming skilled in architecting object-oriented solutions usually takes 6-18 months or more, depending on the initial skill set of the software engineer.

VIII. Thou Shalt Design Web-Centric Applications and Reusable Components

As in the case of object-oriented programming, not all software architectures are web-centric. With the explosion of the Internet, however, web-centric software is more universal. This changes not only the way you design software, but also some of the very basic infrastructure requirements. Here are some of the infrastructure components needed for a typical web-centric application:

- **Database server.** A web-centric application typically accesses one or more corporate databases. Unlike a two-tiered client/server application, however, a web-centric application is less likely to access the database directly. More commonly, a web-centric application accesses some sort of middle-tier application server containing the business rules of the application. The middle-tier application then communicates with the database server on behalf of the web-centric client. Such a multi-tiered approach offers many advantages, including greater application scalability, security, and flexibility.
- **Application servers.** In a web-centric architecture, application servers implement the business logic of the application. In many cases, this is programmed using the Java language. From a Java program, the Java Database Connectivity (JDBC) API is most often used to connect back to the central database. Specialized application servers may offer services such as DBMS data caching or transactions. A single business function is often broken down into components that execute across many applications servers.
- **Web servers.** Web servers are used to store and distribute both Java applications and web pages containing text and graphics. Many advanced applications generate web pages dynamically to provide a customized look and feel.
- **Caching proxy servers.** These servers, while not explicitly part of the application, are typically located strategically across the network to cut down on network bandwidth and provide faster access times to web-based data and applications.
- **Reverse proxy server.** A reverse proxy server is typically used to provide remote users secure access over the Internet back to their corporate intranet.
- **Web clients.** Until recently, a web client meant either Netscape's Communicator or Navigator browser or Microsoft's Internet Explorer browser. Today, a web client could be one of these browsers, or any of the following:

- HTML-rendering Java Bean component in your application
- Applet viewer built into a Java Development Kit (JDK)
- Java application
- Collection of functions built directly into the operating system

One of the main advantages of web-centric design is that it starts taking IT out of the business of supporting heavyweight clients. In fact, most new operating systems ship with one or more bundled web browsers, so no additional client installation is required for a web-centric application. Even if you're deploying to older desktops without a bundled web browser, the popular browsers are available for free and easily installed. If a web-centric application is designed correctly, the end user client really doesn't matter, as long as an HTML-rendering component and Java Virtual Machine (JVM) are present.

If there is any disservice that the web has brought to software development, it's that inexperienced managers may believe that the web has trivialized web-centric software development. True, almost any word processor today can spit out HTML code, and dozens of development tools promise point-and-click generation of Java code, while the web makes software distribution a non-issue. All of this has allowed web-savvy organizations to develop new applications on "Internet time," several times faster than using traditional client/server environments. But it hasn't trivialized software development. From requirements definition through production acceptance, the same disciplines that apply to client/server development hold true for web-centric development.

While embracing web-centric design doesn't necessarily require using reusable components, it certainly is a good place to start. More development organizations every day are investing in the design and development of reusable components. Several of the popular component frameworks have fostered the popularity of reusable components. Let's consider some of the advantages of building reusable components:

It can take longer and be more expensive to design and implement a given function as a reusable component than as a non-reusable one. The savings only come when the component is reused. Especially with web-centric design, however, developers increasingly reuse well-designed components. This reuse is facilitated by component standards such as Java Beans components integration. The cost tradeoff, therefore, is to compare the overhead of reusable design with the average number of times a component can be reused. A reusable component, on average, might cost from 10-25% more to develop. Few development managers today could justify a 25% cost and schedule overrun just to save the next project money. However, properly implemented, reusable components can begin saving project money today.

- If you invest in the design of reusable components and an accompanying framework, you will undoubtedly find components you can reuse from elsewhere in place of some of the code you would otherwise develop.
- It's likely that components developed for one project can be reused elsewhere in the organization on other projects.
- You can buy and sell components (either externally or by exchanging with other development groups inside your company).

- Well-built components are much easier to swap out and upgrade.

IX. Thou Shalt Plan for Change

The best developers and architects plan for change during all phases of the software lifecycle. During the course of an average one-year development cycle, not only will the design be subject to change, but so will the user requirements, the development tools, the third-party software libraries, the DBMS system, the operating system, the hardware, the network, the programmers, and many other aspects of the application that can't possibly be foreseen or otherwise planned for. Some aspects of change, such as a new release of the operating system, can certainly be planned for by discussing schedules with the vendor and making a decision whether a new release should be installed or not. Sometime during the application's life, however, the underlying operating system will probably have to be upgraded, so it's really just a matter of when changes such as these are made. In either case, you still have to plan for the changes.

There are many ways to plan for change. For starters, allow extra budget and schedule in your project for unforeseen changes. At the start of the project, work to clearly identify all risk items that could lead to a possible change somewhere in the future. During the design, look for ways to mitigate the risk of a change further downstream. At the coding level, look for ways to quickly and easily set up code to be adapted to new situations and events within your business. For instance, use tabular definitions whenever possible versus hard-coding these parameters into your code.

X. Thou Shalt Implement and Always Adhere to a Production Acceptance Process

Our last commandment centers around the use of the Web-Centric Production Acceptance (WCPA) process mentioned in earlier commandments. The WCPA process is really a superset of the Production Acceptance process in the book titled *IT Production Services by Harris kern*. Most of the WCPA process is useful even if you're not yet designing web-centric applications. Production acceptance takes an application from the early stages of development and into a production status. However, planning for WCPA really begins at the first stages of the software development process. This is where we first start getting users involved and keeping them involved throughout the development process through the use of customer project teams. At the same time, the development team needs to start getting IT operations involved. It's never too early to start getting both users and operations involved.

One of the reasons we developed the WCPA process is to serve as a communications vehicle. All too often, development organizations are isolated from the business groups that will use their applications and the operations group that will run and maintain them. Proper use of the WCPA process helps promote and instill good communication practices. Just as iterative development is a key software development process, so is iterative and ongoing communications with operations and with users.

This commandment is important because, without a closely followed WCPA process, your business may lose valuable revenue or even customers because your web-centric application doesn't function as expected. Perhaps one of the earliest examples of a WCPA process can be traced to Netscape's web server when they first opened for business in mid-1994. When

designing their web site, Netscape engineers studied the web server load of their competitor at the time, the Mosaic web site at the National Center for Supercomputer Applications (NCSA). At the time, the NCSA site was receiving approximately 1.5 million hits a day. Netscape engineers thus sized their web site to initially handle 5 million hits a day during its first week of operation. Luckily, Netscape had planned their web site architecture to be scalable, and were able to add additional hardware capacity to handle the load.

The success of the WCPA process is also related to the robustness of your software system's architecture. An even greater percentage growth than Netscape's occurred at AT&T Worldnet. AT&T had expected to sign up 40,000 customers for their Worldnet Internet service during its first month of operation and had designed the site accordingly. During its first month of operation, however, Worldnet registered 400,000 new subscribers, 10 times the expected amount. Luckily for AT&T, they had architected the system for growth and put in place the equivalent of a WCPA process. As a result, all new subscribers were able to start receiving service, with few complaints of busy signals on dial-in attempts (in contrast to some other well-known Internet services).

A great example of what happens when you *don't* follow a complete WCPA process occurred at a major U.S. bank. The bank was planning an upgrade to its Internet home-banking service. Prior to the upgrade, the bank used a load-balancing scheme to distribute users to a number of front-end web servers, all of which connected to the bank's mainframe back-end systems. As part of the upgrade process, the bank was planning to let all users change their login ID and password, thus allowing more individual flexibility than the previous bank-generated login ID scheme. The first time a customer logged in after the upgrade, he would be required to select a new login ID and password or confirm keeping the old login ID. This process was delegated to a separate new web server to avoid interfering with any of the software on the existing load-sharing servers. The bank had some Production Acceptance processes in place, of course, and tested the entire web-centric system. On the first day of production, users started to complain of extremely long access times. Unfortunately, the bank had not taken into account the potential bottleneck of funneling all users through the single server while they were queried for potential login ID changes.

While no process can guarantee that a new production system will function 100% correctly, web-centric applications require new kinds of planning like that covered in the WCPA. Not only are user loads on the Internet much more unpredictable, web-centric applications typically involve the interaction of a much larger number of software and hardware systems.