CHAPTER 1

DEFINING A BUILD

> Philosophy: The build is a piece of software and should be treated as such. The build is among the most heavily used and complex pieces of software in the development group and should be treated as such.
> **—Danny Glasser, Microsoft developer in the Systems Group, March 9, 1991**

The first thing we should do is define what a build is. What Danny describes in the previous quotation is important. The purpose of a build is to transform code written in any computer language into an executable binary. The end result of a software build is a collection of files that produce a product in a distributable package. In this case, **package** can mean a standalone application, Web service, compact disc, hotfix, or bug fix.

If you do not think it is worthwhile to spend resources on a good build process, your product will not be successful. I have been on a couple of product teams at Microsoft that have failed, and I have seen many others fail because they were not able to consistently build and test all of the product's code. I also see this at customer sites when I am reviewing their build process. The companies that have clean, crisp, reliable build and release processes are more successful than the ones with ad hoc, insufficient processes.

The Two Types of Builds: Developers and Project

I like to say that there are really only two types of builds: ones that work and ones that don't. Seriously, though, when you're shipping a product, you should consider these two different types of builds:

- **Developers' (local machine builds**)—These types of builds often happen within an editor such as Visual Studio, Emaqs, Slick, or VI. Usually, this is a fast compile/link of code that the developer is currently working on.

- **Project (central build process)**—This type of build typically involves several components of an application, product, or a large project, such as Windows, or in some cases several projects included in a product, such as Microsoft Office.

The developer's build process should be optimized for speed, but the project build process should be optimized for debugging and releases. I am talking about optimizing the process, not compiler or linker optimization switches. Although speed and debugging are important to everyone who is writing code, you must design a project build process to track build breaks and the offender(s) as quickly as possible because numerous people are waiting for a build to be released. For a developer, what seems to be most important is clicking some type of Build and Run button to make sure the code compiles without errors and then checking it in. For the build team, building without errors and having the ability to track down the person who broke the build is the most important thing.

NOTE In some simple scenarios, these two build cases can use the same process. If this is the case, the team—what I refer to as the Central Build Team—should dictate the build process. This team—not the developers—should design the project build process. All too often, the developers design the project build process, which causes problems. Because developers usually build just the code modules that they work on and not the whole project on a regular basis, they look for shortcuts that are not necessarily in the best interest of building the entire project. For example, they might use file references instead of project references.

If a developer specifically references a file in Visual Studio and the sources of that file change, they are not automatically picked up because a specific version of the file was referenced instead of the project that builds the referenced file. In the interest of saving time, developers use file references. They are not interested in picking up the latest sources of the specified file, but it is not recommended to use file references in a project build.

The Central Build Team should never be at the mercy of mandatory build environment settings for building a specific component. If such a setting is necessary to build a component, it should be proposed to the Central Build Team for inclusion. Then the CBT can determine the impact of the addition or change to the entire project and approve or disapprove the proposal.

Building from the Inside Out

One of my favorite questions to ask a customer's development or build manager when I go onsite is how often they release a new build process. I usually get long pauses or funny looks and then finally get the answer "Every day." Of course, as you might suspect, I am not talking about releasing a daily build, but a new build process. The fact that so many companies do not release new build processes on a regular basis does not surprise me. This is because traditionally creating a build process is an afterthought when all of the specifications of a project have been written. Many project and program managers think that the actual building of a project is pretty trivial. Their attitude is that they can simply have the developer throw his code over the wall and hire someone to press a Build button, and everything will be fine. At Microsoft, we understand that whether you're building the smallest application or something huge and complicated like Windows, you should plan and think through the process thoroughly in advance.

Again, I recommend that you consider the build process a piece of software that you regularly revise and deploy throughout your product team. You should also add to your project schedule some "cushion time" to allow for unforeseen build breaks or delays, I would at least pad the milestone dates one week for build issues.

The concept of "building from the inside out" tends to confuse customers who are not familiar with a centralized build process. The idea is that the Central Build Team determines what the build process is for a product and then publishes the policies to an internal build site. All development teams in the project must comply with the Central Build Team process; otherwise, their code check-in is not accepted and built. Unfortunately, this concept is usually the complete opposite of how a build system for a project actually evolves over time. The Central Build Team for a project usually goes out of its way to accommodate the way developers build their code. "Building from the inside out" means that the Central Build Team figures out the best way to get daily builds released, and everyone uses that process independently or in parallel with the way his specific development team builds. This total change in development philosophy or religion can be a culture shock to some groups. I talk more about changing a company's culture or philosophy in Chapter 18, "Future Build Tools from Microsoft." For now, let's stay on the topic of builds.

What we did in the past in the Windows group—and what they still do today—is to deploy new releases of the *build process* at major milestones in the project life cycle. Sometimes the new releases involve tool changes such as compilers, linkers, and libraries. At other times, there are major changes such as a new source code control tool or a bug tracker.

Because a build lab tends to have some downtime while the build team waits for compiles, links, and tests to finish, it should take advantage of these slow times to work on improvements to the build process. After the lab tests the improvements and confirms they are ready for primetime, it rolls out the changes. One way to deploy a new build process after a shipping cycle is to send a memo to the whole team pointing to an internal Web site that has directions on the new process that the Central Build Team will be using in future product builds.

Microsoft Sidenote: Developers in a Build Lab

Today, the Windows build lab has its own development team working on writing and maintaining new and old project tools. The development team also works on deploying new build processes. Conversely, of the more than 200 customers I've spoken to, only one or two of them have developers working in a build team.

Remember Danny's quote at the beginning of this chapter and notice the date—1991. In 1991, Windows NT had only a few hundred thousand lines of code, unlike the more than 40 million lines of code that Windows XP has today. Even in the early stages of developing Windows NT, Microsoft recognized the importance of a good build process.

Chapter 3, "Daily, Not Nightly, Builds," covers in more detail the importance of the build team being the driving force to successfully ship a product.

More Important Build Definitions

I need to define some common build terms that are used throughout this book. It is also important for groups or teams to define these terms on a project-wide basis so that everyone is clear on what he is getting when a build is released.

- **Pre-build**—Steps taken or tools run on code before the build is run to ensure zero build errors. Also involved are necessary steps to prepare the build and release machines for the daily build, such as checking for appropriate disk space.
- **Post-build**—Includes scripts that are run to ensure that the proper build verification tests (BVTs) are run. This also includes security tests to make sure the correct code was built and nothing was fused into the build.
- **Clean build**—Deleting all obj files, resource files, precompiled headers, generated import libraries, or other byproducts of the build process. I like to call this cleaning up the "**build turds**." This is the first part of a clean build definition. Most of the time, build tools such as NMake.exe or DevEnv.exe handle this procedure automatically, but sometimes you have to specify the file extensions that need to be cleaned up. The second part of a clean build definition is rebuilding every component and every piece of code in a project. Basically the perfect clean build would be building on a build machine with the operating system and all build tools freshly installed.

Microsoft Sidenote: Clean Build Every Night

While working in the Windows NT build lab on NT 3.51, I remember reading in a trade magazine that the Windows NT group ran clean builds every night. The other builders and I laughed at this and wondered where this writer got his facts. We would take a certain number of check-ins (usually between 60 and 150 per day) and build only those files and projects that depended on those changes. Then one of us would come in over the weekend and do a clean build of the whole Windows NT tree, which took about 12 hours. We did the clean builds on the weekend because it took so long, and there were usually not as many check-ins or people waiting on the daily build to be released.

Today, with the virtual build lab model that I talk about in Chapter 2, "Source Tree Configuration for Multiple Sites and Parallel (Multi-Version) Development Work," the Windows NT team can perform clean builds every night in about 5 or 6 hours.

- **Incremental build**—The secret to getting out a daily build to the test team, regardless of circumstances, is to perform incremental builds instead of daily clean builds. This is also the best way that you can maintain quality and a known state of a build. An **incremental build** includes only the code of the source tree that has changed since the previous build. As you can guess, the build time needed for an incremental build is just a fraction of what a clean build takes.
- **Continuous integration build**—This term is borrowed from the extreme programming (XP) practice. It means that software is built and tested several times per day as opposed to the more traditional daily builds. A typical setup is to perform a build every time a code check-in occurs.
- **Build break**—In the simplest definition, a **build break** is when a compiler, linker, or other software development tool (such as a help file generator) outputs an error caused by the source code it was run against.
- **Build defect**—This type of problem does not generate an error during the build process; however, something is checked into the source tree that breaks another component when the application is run. A build break is sometimes referred to or subclassed as a **build defect**.
- **Last known good (LKG) or internal developers workstation (IDW) builds**—These terms are used as markers to indicate that the build has reached a certain quality assurance criterion and that it contains new high-priority fixes that are critical to the next baseline of the shipping code. The term LKG originated in the Visual Studio team, and IDW came from the Windows NT organization. LKG seems to be the more popular term at Microsoft.

Microsoft Sidenote: Test Chart Example

The best way to show how Microsoft tracks the quality of the product is through an example of the way the Windows team would release its version of a high-quality build. Again, the Windows team uses the term internal developers workstation (IDW), and other teams use last known good (LKG).

In the early days of the Windows NT group, we had a chart similar to the one in Figure 1.1 on the home page of the build intranet site. Most people on the project kept our build page as their default home page so that whenever they opened Internet Explorer (IE), the first thing they would see was the status of the project; then they would check the Microsoft (MSFT) stock price.

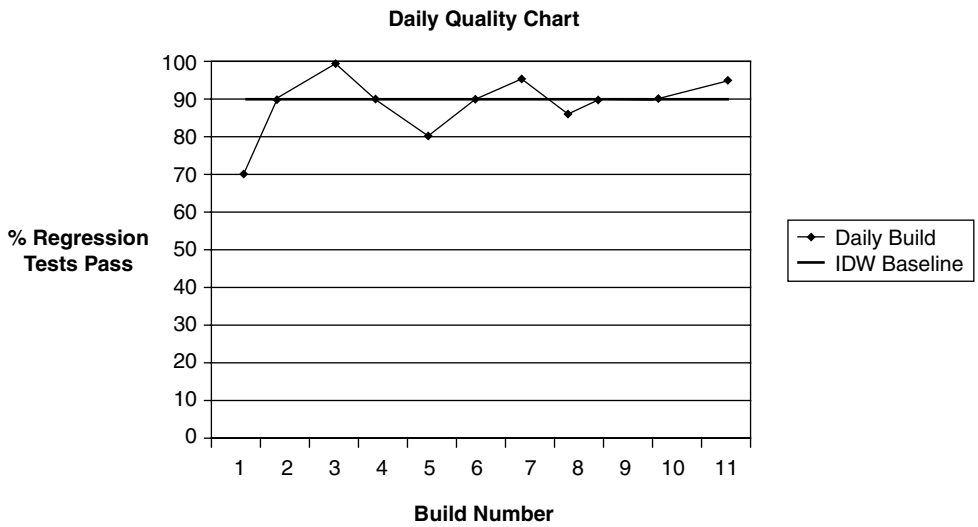

FIGURE 1.1 Sample quality chart.

The way to read Figure 1.1 is that any build we released that passed more than 90 percent of the basic product functionality tests—what we called regressions tests—and did not introduce new bugs was considered an IDW build. This quality bar was set high so that when someone retrieved a build that was stamped IDW, he knew he had a good, trustworthy build of the product. As you can imagine, when the shipping date got closer, every build was of IDW quality.

Furthermore, when a new IDW build was released to the Windows team, it was everyone's responsibility to load the IDW build on the machine in his office and run automated stress tests in the evening. Managers used to walk to their employees' offices and ask them to type **winver** to verify that they had the latest IDW build installed before they went home for the evening. Today, managers have automated ways to make sure that everyone is complying with the common test goal. This is also where the term "eating our own dog food" originated. Paul Maritz, general manager of the Windows team at that time, coined that phrase. It simply means that we test our software in-house on our primary servers and development machines before we ship it to our customers. Dogfooding is a cornerstone philosophy at Microsoft that will never go away.

The build team would get the data for the quality chart from the test teams and publish it as soon as it was available. This is how we controlled the flow of the product. In a "looser" use of the word *build*, the quality became part of the definition of a build number. For example, someone might say, "Build 2000 was an excellent build" or "Build 2000 was a crappy build," depending on the test results and personal experience using the build.

How Your Product Should Flow

Never mistake activity for achievement.
—Coach John Wooden, UCLA basketball legend

Recently, while I was at a popular application development site going through a build architect review, I noticed how extra busy everyone was. Everyone was running around like he was on the floor of the New York Stock Exchange trying to sell some worthless stock before the market closed. People barely had enough time to stop and talk to me about their top five build or SCM pain points. They didn't have time for chitchat because they were too preoccupied with putting out fires such as build breaks, administrating tools and permissions, and reacting to new bugs coming from their customers. Their explanation was that they did not have enough resources to do what the upper managers wanted them to do. This might have been partially true, but it was not the complete truth. They were equating this busy work as their job duties and why they got paid. This was later confirmed when I gave them my final trip report of how to improve their processes such that everything would be fixed and automated. The first question their build team asked was "If all of this is fixed and automated, then what will we do?" I was shocked. These guys were so used to being in reactive mode that they seemed to think that if they were not constantly putting out fires, their position was not needed.

The rest of this chapter outlines a smooth flow of how your product development should go. As Kent Beck, author of *Test Driven Development* and several *Extreme Programming* books, points out, flow is what the build team should encourage and try to achieve. **The build team drives the product forward.** I put together Figure 1.2 to show how this works at Microsoft because I don't think this concept is always clear. I don't think this concept is always clear, as this is the underlying philosophy of this book.

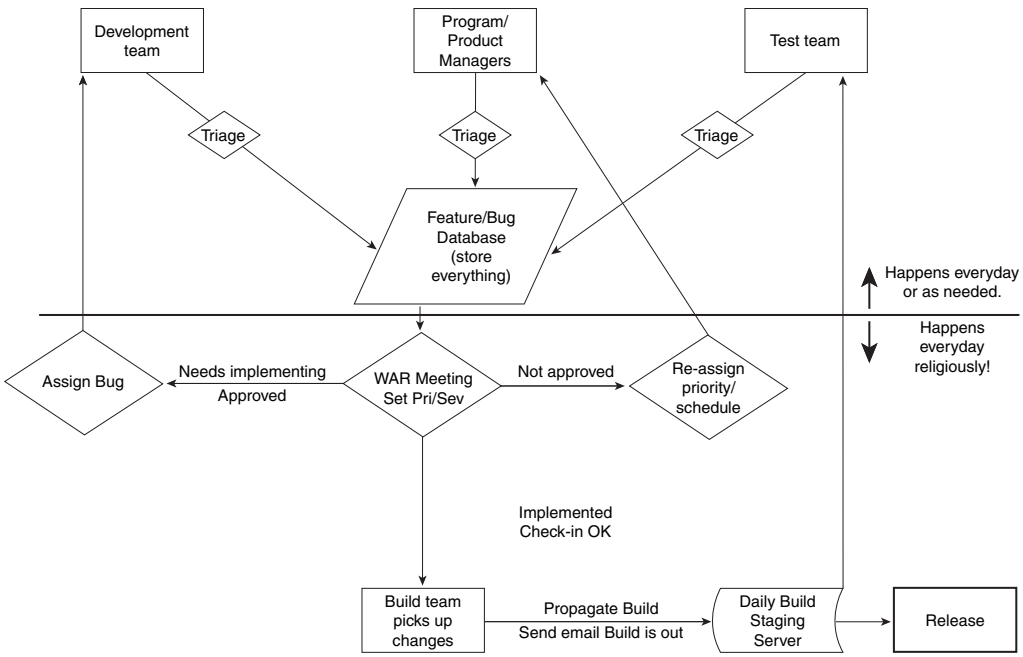

Figure 1.2 Software development flow.

Software Development Flow

The three boxes at the top of Figure 1.2 represent the respective teams listed. The members of each team meet to discuss the progress of its code development.

After the teams discuss the issues, they mark their priority in a bug database, or **work item tracker**. Sometimes at Microsoft we call everything (features, requirements, bugs, tasks, risks, wish list) a bug, but work item is more accurate.

Teams must enter every type of code implementation or necessary fix on the project into the work item tracker and assign it a tracking number.

Some Work Item Field Definitions

With the internal Microsoft work item tracker more than 46 fields are available in each item, although not all are used all the time. For Microsoft confidentiality reasons, I cannot include a graphic of our tracking tool here. However, the following are some of the fields that are included in a work item.

Setting work item priority and severity:

- **Priority**—This field communicates overall importance and determines the order in which bugs should be attacked. A bug's priority takes severity and other project-related factors into account.
  - **Pri 0**—Fix before the build is released; drop everything you are doing and fix this immediately.
  - **Pri 1**—Fix by the next build.
  - **Pri 2**—Fix soon; specific timing should be based on the test/customer cost of the workaround.
  - **Pri 3**—Fix by the next project milestone.
  - **Pri 4**—Consider the fix by the upcoming release, but postponement is acceptable.
- **Severity**—This communicates how damaging a bug is if or when it is encountered.
  - **Sev 1**—This involves an application crash, product instability, a major test blockage, a broken build, or a failed BVT.
  - **Sev 2**—The feature is unusable, a bug exists in a major feature and has a complex workaround, or test blockage is moderate.
  - **Sev 3**—A minor feature problem exists, or the feature problem has a simple workaround but small test impact.
  - **Sev 4**—Very minor problems exist, such as misspelled words, incorrect tab order in the UI, broken obscure features, and so on. Sev 4 has little or no test impact.

Following are other work item or bug field definitions:

- **Status**—Active, Resolved, or Closed
- **Substatus**—Fix Available
- **Assigned To**—The most critical field, because this is the owner of the item
- **FixBy**—The project due date for the bug fix

Each work item has two build fields:

- **Build (1)**—The build number that the bug was found on
- **Build (2)**—The build number that the bug was resolved on

Microsoft Sidenote: How Visual Studio Resolves and Closes Bugs

Testers close bugs.
—Deep thought of the day.

I once was asked by a test manager to summarize everything I learned about builds in one sentence. I told him that "there are no free lunches, especially in the build lab, but there might be free beer." He told me that he was disappointed that I did not have anything deeper than that. He then said his motto was "Testers close bugs." I knew what he meant, so I said with tongue-in-cheek, "Wow, that's deep." I'm not sure if he took that as a compliment or just thought I was not very funny. Regardless, he did have a good point.

Let's break down the details of "a bug's life…"

When a developer fixes a bug on his machine, he marks the bug's substatus as Fix Available and keeps it assigned to himself. After he checks in the change to the team branch or tree, he resolves the bug (changing the status from Active to Resolved) and reassigns the bug to the original bug opener or a tester who owns that area of the product.

The original bug opener or tester then waits until an official build comes out that contains the bug fix. He then walks through the repro steps to ensure that the bug has truly been fixed. If it has, he closes the bug by changing the status from Resolved to Closed. If the issue still exists, the bug opener or tester reactivates the bug by resetting the status to Active and reassigning it to the developer. This continues until the bug is fixed or gets postponed for the next milestone or release.

WAR or Ship Meeting

Known as WAR, Central WAR, or Ship (the softer, more friendly Visual Studio Team System term), this meeting is focused on tracking and controlling the main product build. Its goal is to ship the product at a high quality according to its schedule by dealing with day-to-day project issues, test reports, and metric tracking.

Figure 1.3 WAR team.

The WAR team and—everyone attending the WAR meeting—must approve every work item before it can get built and shipped in the product. After the WAR team approves a work item, a field in the bug tracker gets set so that everyone on the build team knows that it's okay to accept this check-in into the main build lab.

If the WAR team does not approve the work item, the work item is reassigned to the person who opened it or to Active, which means that no specific person owns the bug, just a team. At this point, if the person who opened the bug thinks it should be fixed sooner than the people in the WAR meeting determine, it is his responsibility to push back with a solid business justification. If the person pushes back to the WAR team with a solid business justification and the WAR team *still* doesn't accept the change into the build, the work item is marked as Won't Fix or Postponed.

Upon the item's WAR team approval, the developer works with the build team to get his code changes into the next build. After the build team compiles and links all the source code, the code goes through the congeal process, which brings all the pieces of the project together. This includes files that don't need to be compiled, such as some HELP, DOC, HTML, and other files.

Then the post-build process starts (more on post-build in Chapter 14, "Ship It!"), which in some cases takes just as long or longer than the build process.

Microsoft Sidenote: How the Visual Studio Team Controls All Check-Ins and "Tell and Ask Mode"

The Visual Studio team controls check-ins in another way: the "tell and ask" process. Project managers use this process to slow the rate of code churn and force teams to deliberate about what work items or bugs are fixed or open. This is called **triage**.

Scott Guthrie is the product unit manager in Visual Studio. He explains triage in his blog:

> During tell mode, teams within our division are still given discretion to fix any bugs they want—they just need to be prepared to present and explain why they chose the ones they did to the central division ship room. This ends up ensuring a common bar across the division, slows the rate of fixes, and slowly brings up build quality. You might naturally wonder how not fixing bugs could possibly bring up build quality, since this obviously seems counterintuitive. Basically, the answer lies in the regression percentage I talked about earlier for check-ins. Even with a low regression number, you end up introducing new bugs in the product. (And when you have a division of over 1,000 developers, even a low percentage regression rate can mean lots of bugs introduced per week.) By slowing the rate of check-ins, you slow the number of regressions. And if you focus the attention on bad bugs and add [an] additional review process to make sure these fixes don't introduce regressions, the quality will go up significantly.
>
> During ask mode, teams within our division then need to ask permission of our central ship room committee before making a check-in—which adds additional brakes to slow the check-in rate. In addition, all bugs in ask mode must go through a full nightly automation run and buddy testing (which takes at least 12 hours) to further guard against introducing problems. Ask mode will also be the time when we'll drive our stress-passing numbers up to super-high levels, and we'll use the low rate of check-ins to find and fix pesky, hard-to-find stress failures.

You can read the entire entry at `http://weblogs.asp.net/scottgu`. I talk more about processes to control all check-ins into the source tree in Chapter 10, "Building Managed Code."

Release to Staging Servers

After the build is complete and has no errors, it is propagated to the daily build servers, where at least 15 to 20 builds are stored with all the sources and tools necessary to build. Milestone releases also are kept on the server. This is where the test team picks up the build. This is the "secret" to fast development and keeping your developers happy. I realize that most if not all SCC tools can retrieve sources of a certain build but sometimes those tools are clumsy or the labels on the trees are not accurate. So we came up with this staging server with massive amounts of diskspace available and stored our releases on it. It is a lot easier for the development and test teams to search that server than the SCC database.

From the staging servers, the build can go to production. This process is covered in Chapter 14.

Important Definitions

The following sections discuss terms that are specific to Visual Studio but that are used all over the Web and at various companies I have visited.

Solution Files

If you are new to Visual Studio .NET, you probably are not familiar with the term *solution*. A **solution** essentially represents everything you are currently working on. Visual Studio .NET uses solutions as containers for individual projects, which generate your system components (.NET assemblies). Solution files maintain project dependency information and are used primarily to control the build process.

Project

In the context of this book, projects are one of three types:

- **General development projects**—The term **project** in its loosest sense refers to your team's current development effort.
- **Visual Studio .NET projects**—Visual Studio .NET uses project files as containers for configuration settings that relate to the generation of individual assemblies.
- **Visual SourceSafe (VSS) projects**—A project in a VSS database is a collection of files that are usually related logically. A VSS project is similar to an operating system folder, with added version control support.

Microsoft Solution Framework

It would not be proper to print a Microsoft book on Software Configuration Management and not mention the Microsoft Solution Framework (MSF) that has been publicly available for years. The origin of this process came from the Microsoft Consulting Services (MCS) group and is based on the terms and the way that Microsoft organizes its software development groups. The funny thing is that many people on the Microsoft product teams have never heard of MSF. They use the processes or know the terms, but they do not realize that Microsoft has been teaching this to customers for years.

That is a good example of how a documented process came from an informal undocumented process. Now the documented process (MSF) is the leader, and many new terms in the product teams come out of MSF. MSF will be included in the upcoming Visual Studio Team System. It's a great high-level view of how Microsoft runs its product teams. Because a ton of information about MSF is available on the Microsoft Developers Network (MSDN `http://msdn.microsoft.com`), I will show just one chart that sums up the whole process (see Figure 1.4).

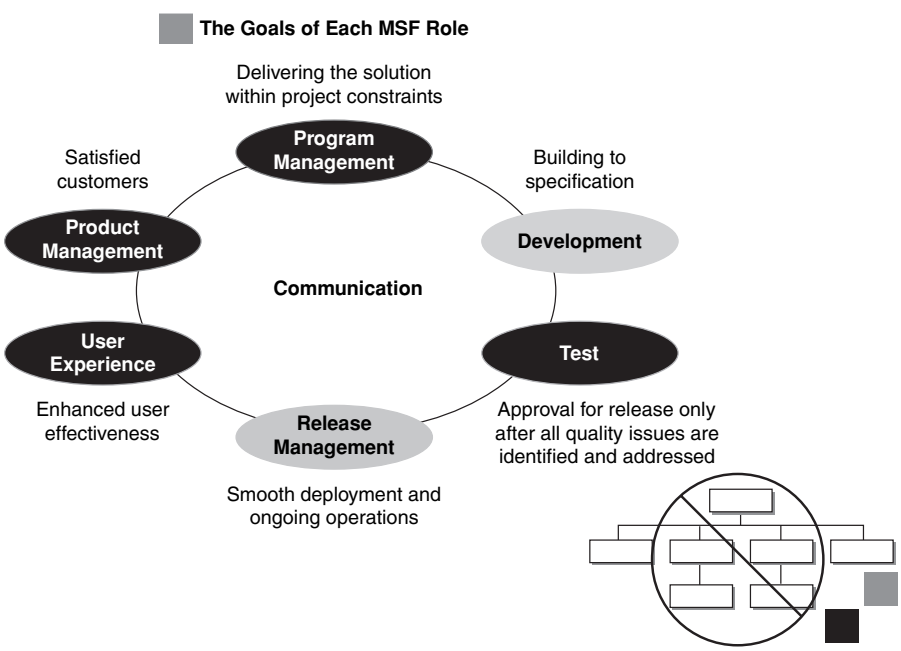

Figure 1.4 MSF roles.

Figure 1.4 is self-explanatory. The point of the graphic is to show that there is not a hierarchical approach to shipping software at Microsoft, but a "round table" one. Ideally the Build Master would be King Arthur.

Summary

Speaking the same language is important in any project or company. Making sure everyone is clear on the terms or lingo in your group is especially important. For example, if you are talking about a build process or bug to someone on your team and do not define the context, or if the terms are not explicitly defined somewhere, you'll miscommunicate your point or vice versa. This can lead to project setbacks.

In the following chapters, I will continue to define terms that we use at Microsoft and what seem to be industry standard terms. This is important because there can be variations of a definition, and I want to make sure we are all clear on the points being made. Also, it is the build team's responsibility to set these definitions for a group and publish them on an internal Web site so that no one's confused about what they mean and people who are unfamiliar with the terms can reference them easily.

Recommendations

- Define terms in your development process, and keep a glossary of them on an internal build Web page. If you like, standardize on the definitions in this chapter.
- Clean build your complete product at least once per week, or every day if possible.
- Use incremental builds on a daily basis if clean builds are not possible or practical.
- Start charting the quality of your product, and post it where everyone involved in the project can see it.
- Release LKG (or IDW) builds weekly; then switch to daily releases toward the end of the shipping cycle.
- Follow the Software Development Flow diagram.
- As noted earlier, I will also post the definitions in this book to `www.thebuildmaster.com` site so you can download them and publish them to your group or company.