

# Designing Your Build Process

By Alex S. Brown, PMP

The build process is like the fulcrum of a crowbar. It is the invisible, unmoving point that provides leverage. Faced with problems, most people will tell you, "Get a bigger crowbar!" or "Put your back into it!" Only someone with experience asks, "What do you have that crowbar wedged against?" In a similar way, many people look at the problems of software projects and say, "Work more hours, you will get through it," or "Try this whizbang.com development method," or "You need whizbang.net architecture." An experienced software veteran will ask, "How are you doing your builds?" The build is almost never the obvious problem in a project, but as the hub of the software testing and delivery process, it is often the underlying fix for many problems.

Every software project will have a compile and install ("build") process. It is up to the Project Manager to decide whether it is worth designing a process, or if it is better to allow the programmers on the team to cobble together something based upon their previous experience.

## Pitfalls and Problems

A poorly designed build process can cripple a project, particularly large projects involving many developers or many platforms. The problems associated with poor build control are very similar to those associated with poor configuration management:

- Tool incompatibilities discovered late in the development lifecycle
- Incompatible interfaces or implementations discovered shortly before desired ship date
- Developers each use "standard" but different compilers, only to find that their code cannot compile or run together cleanly
- Conflicts discovered in the way developers call shared components
- Late discovery of installation incompatibilities, such as different developer assumptions about target installation directories, web server directory structures, and file naming conventions
- Configuration and installation mistakes waste valuable tester work-hours

On one project, I saw two developers each working with a different version of Microsoft Visual Basic (VB), both working on the same software product. They were each comfortable with that fact, up until the point where they had to deliver an integrated product. They discovered critical system services did not operate as they expected, and testing was delayed for weeks while one of the developers converted their code to compile under the older version of VB.

Another project attempted to deliver production-quality code six different times within the space of two weeks. Each time the installation failed due to misspelled registry entries, missing files, or poor installation instructions. The registry entries were hand-installed by the developers for their unit testing, but were installed programmatically for production. The production installation procedure was never tested, until each new software version was delivered.

In mainframe environments, it is not uncommon to run a newer version of the operating system and compiler in the development environment than in the production environment. I have seen incompatibilities result in multiple, costly, fruitless installation attempts. Sometimes new code will install correctly, but calls to external drivers, particularly communication services, fail under obscure conditions. Debugging the problems is almost impossible, since the problems appear only in production, not in the development environment.

## The Range of Build Processes

There is a wide range of build processes to choose from. Little has been written about software builds, and what has been written focuses on the Microsoft daily build process. Project Managers have many decisions to make, but the two most critical are informal vs. formal and scheduled vs. on-request:

- **Informal vs. Formal:** Formal build processes require written documentation at the beginning and end of each build, with an independent organization performing all compiles and all installation procedures.

Informal builds usually have no documentation, except as the development team decides to write; developers perform all builds themselves.

- **Scheduled vs. On-Request:** Scheduled builds happen on a given time of day on a given day of the week. Developers are responsible for making sure their fixes are ready for the build, and the build will happen on schedule, even if nothing has changed in the code base. On-request builds happen only when a designated person asks for a build. Typically the development team lead or the Project Manager has authority to request builds.

The Project Manager can construct his or her own custom technique. Most real-life build processes combine formal, informal, scheduled, and on-request techniques. Examples include:

- **Allowing developers to check in new code for the build anytime:** A build group performs the build, documenting the results back to the developers. Any code the developer checks in is automatically included in the next build, with no documentation required by the developers. Use the "get latest" command on your code repository, compile and install the result. I very much recommend having a festive "build hat" with this method; any developer who breaks the build with a change has to wear an outrageous hat until the next build. The hat is very effective, making sure that developers do not break the build. To create a production-ready build, the team simply selects a stable, well-tested build, and releases it. I have personally had a lot of success with this method, which combines the central control of a formal build with the developer freedom of an informal build.
- **Allowing developers full control in building a development environment, but requiring written requests to promote code into a controlled testing environment:** A non-development group or a chosen developer handle the move into the testing environment, while all developers move code into the development environment. The controlled testing environment is well-documented for each change, while the development environment has only ad-hoc documentation. The biggest risk to this method is that the development environment can get out-of-sync with the controlled testing environment, and it becomes impossible to debug and test effectively in development. Periodically comparing the development and controlled code base can mitigate that risk. Having a development environment that contains the latest controlled build (separate from the uncontrolled development environment) can ensure that programmers can recreate and debug all problems. Small development teams (less than five programmers) developing enterprise-level products often use this technique; the small group of developers can track changes effectively through informal communication, and the controlled changes in testing keep the larger organization aware of changes through formal communication.
- **On-Request Development Builds and Scheduled Product Builds:** Often developers can create new builds whenever they want, either on their own PC, or in a personal library in a mainframe environment. In fact, having some method to rebuild and unit test at will is critically important to having productive programmers. Programmers need immediate or near-immediate feedback about whether their changes are successful. Many teams extend this technique to the group, providing a shared, team-wide, testing environment. Each developer compiles his or her own components at will and copies them to this environment. This team-testing environment helps programmers shake out problems when integrating components with the rest of the team. The team-testing environment is difficult to recreate, however, so it is often paired with a scheduled build, which may include all components from a variety of teams. The testing schedule and the roll out schedule will drive the schedule for the full-product builds. The biggest drawback is that the development environment is almost always out of sync with the full-product build. Small, well-integrated teams can be successful, though. The scheduled product builds help to provide visibility of progress, setting a pace for the project. Goals for specific functionality in each scheduled build can help the Project Manager ensure that dates and requirements are being met.

The options above are just a sampling of possibilities. It is up to the Project Manager to select the method that meets his or her needs the best, or to invent a new method. Having a list of possibilities and choices helps the Project Manager in that decision.

### Factors to Consider

The build process becomes a part of your team's culture. Jim McCarthy has said, "The daily build is the heartbeat of the project. It's how you know you're alive." (McCarthy, [Dynamics of Software Development](#)) Microsoft Project Managers chose the daily build to meet their unique needs for fast turnaround, frequent management updates, and high visibility of team progress. When selecting a process for your team, be sure to consider the following factors:

- Developer experience

- Existing organizational processes
- Environmental complexity
- Number of teams operating in the same computing environment
- Complexity of the compile and build process
- Time required for a single compile-and-build cycle
- Organizational support for computer languages used by the project and for computer platforms used by the project
- Tolerance for risk
- Reporting and communication requirements

Do not forget the importance of your team's experience when selecting a method. Using a brand new method has some risks if the team is tackling many new challenges. There is a benefit to keeping to the "way we have always done it." In contrast, a new build method can be a wonderful way to revitalize a existing team, and to open them up to other new ideas. Either way, it is critical to understand team experience before setting your policy. Organizational processes and procedures can often constrain your choices.

The number of teams in the environment, the complexity of the environment, and the time required for a build are all critical factors. Ideally the build will be performed in off-hours or on dedicated machines, so that the normal development and testing day is uninterrupted. If a full build requires more than eight hours to run, though, a daily build may be impractical; daily partial builds with a full weekly rebuild might be more appropriate. Further, if the organization can support it, a dedicated build team, experienced with the compile options and installation procedures for supported software, can enhance the productivity of expensive development resources.

Ironically, the lower the risk tolerance of the project, the more frequent builds should be. Dynamic organizations like Microsoft build frequently as a way to reduce risk; if they can build their system successfully every day, it is less likely that they will be surprised by last-minute problems. Common wisdom says that a project with lots of frenetic activity, which is building the software more quickly than it can thoroughly test it, is in the process of melting down. Frequent builds can actually result in a more stable system, because the testing team has the opportunity to catch developer mistakes more quickly. Since few code changes go into each build, it is easier to isolate the code change that is causing problems.

Use the build as a reporting tool. Microsoft shows the build number on many of its products, including the blue boot-up screen for Windows NT 4.0 (build 1381). Track defects opened and closed by build, and you have a useful reporting tool. Build failure and success are good tools to track early progress.

### Key Principles

**Match your process to your situation.** There is no one right way to compile code and to build environments. Be flexible, and change your methods if your current method fails.

**Have a consistent compile method for each language and platform.** Strange problems surface when code is compiled onto one test machine using two different versions of the same development tools or two versions of the same libraries. These problems are incredibly hard to diagnose. Control the compiles one of three ways:

- Set a clear, unambiguous standard for all developers,
- Choose one and only one person to perform all compiles, or
- Set up an independent group to perform all compiles across multiple projects.

**Developers need access to the official build.** Developers cannot recreate problems if they cannot debug the build; it is one of the best arguments for letting development keep control of the build. It is simple for a build-team to meet the developer's needs, though: the build-team always creates a machine for development's exclusive use. The extra cost of the time and machine is usually quickly repaid. Cheaper ways are to provide an official output of the build process in a shared machine's file system or to have simple build procedures. Developers can examine the build for consistency with their own environment either by comparing against the baseline or recreating the build themselves.

**Prepare for roll out.** Binary executables and installation methods are the fundamental output of any software development process. In the planning of your project, make sure that your build process supports the roll out plan for the software. Check version numbers of libraries, system software, and approved development tools in the target production environment. If other teams have successfully released software to your target environment, use their installation methods as a starting point for your own. Where possible, use production standards, and get official exemption to standards as early as possible if necessary. Also, the sooner you begin performing production-like builds, the sooner your testers will catch any conflicts or problems. Start early!

**Match the build process to your organization and roll-out plan.** Avoid informal build procedures if your target environment is a complex, bureaucratic, production environment. You risk catastrophe at the project end, when attempting to go from heavily informal processes to very formal processes. Likewise, avoid a complex, paper-intensive build process in a small development organization which will host the final application on a single PC maintained by the organization itself.

**Get buy-in from the team.** The build process is the critical interface between development, testing, and roll out. If the build fails, testing and roll out are impossible. The build can create discord and frustration among the development team if the team has trouble following its procedures or if it is unreliable. Getting the full team's buy-in early on is critical to project success. Include developers, testers, and the implementation team as early as possible. If these people understand and appreciate the build process, they will cooperate with its procedures more readily. When problems arise, teams are willing to fix and adapt the process, but only if they understand the processes, and only if they believe that it should work. If the build process is imposed from above, teams are more likely to complain about problems than to solve them.

**Be Prepared to Vary the Frequency of the Build.** Know your limits. How often does your team expect to build at a minimum? What is the most frequent build schedule envisioned? These questions usually change during the testing cycle. Typically builds are more frequent at the start of testing, when there are many critical problems to fix, and less frequent at the end of testing, when a stable environment is needed to test complex system functions.

### Evaluate Failure Modes

A robust build process will be simple and flexible. It is guaranteed that at some point the build will fail, and the team will need to recover. When evaluating a build process, ask the following questions:

- If the developer checks in some bad code, how will we know? Will we be able to code and test the next day? If work-time might be lost, have we budgeted for it? Risk Management can help to estimate and manage the contingency fund required for build failure. Better yet is to design a method using alternate machines, so that while one machine is being built, the old build is still up and running. When the new build is complete, everyone begins work on the new build, and the old machine is available for the next build. This solution is not always cost-effective, however.
- Can we roll back to previous code? Sometime, someone will check in code that should not be fixed; it should be rolled back. Roll back is a standard feature of every source-code repository. Make sure that everyone knows how to roll back their code, and make sure that right version of the code will appear in the next build.
- What if the build fails? How quickly can we get a new build? Will the old build still operate?
- What times of day or times of week cause problems for the build? Builds require lots of CPU time, and the compiling system should have little or no outside activity. Depending upon your physical computing resources, you might have to defer builds until off-hours, to avoid production or testing CPU loads, particularly with mainframe environments. Server or workstation environments can often have a dedicated build machine purchased, to eliminate this dependency.
- What else can go wrong? Make sure that every failure point in the process is monitored, with fast notification and resolution of problems. The biggest risk that a Project Manager faces with the build is that the testers will not have anything useful to test and that the developers cannot deliver their fixes. One hour of elapsed down time due to build failure can multiply as lost work-hours or even work-days for an entire software development team.

### The Build and The Project Lifecycle

If you start your build as early as possible in your development lifecycle, you will find the build frequency indicates your progress. Builds are frequent to start, sometimes multiple times per day. It is a struggle just to get the first successful, integrated build. There is little to test, and the team is focused on coding, so builds are frequent, with little pressure.

As the system grows, builds become more important. More and more of the team's time is spent on testing, and testing absolutely depends on the build to get new fixes and new functionality to test.

At some point, the team may need to reduce the frequency of their regular builds. Large, complex systems often require long, complex testing scripts. Once the basic system is stable, the testers might request a build once a week instead of once a day, to run these scripts from end-to-end. Simpler projects do not experience this phase, but this phase is virtually mandatory for systems with long, overnight batch processes, or financial systems with month-, quarter-, or year-end processing.

As the system approaches completion, the team may accelerate the build process. For one project, we had builds three times per day. The system was stable, there were few problems left to fix, and there were few scripts that still encountered bugs. We could run all of the scripts with problems in a matter of hours, and the development team typically had a handful of fixes every few hours. I only recommend building multiple times per day as an end-of-project measure, to shut down a last few problems, but it is very effective. The Project Manager must be certain that the whole system is being regression-tested while the new fixes are applied, to make sure that the fixes do not cause problems in the working portion of the system. A total-system regression test can be performed over several builds, however. There is no need to insist on a full regression test with every build.

### Critical Supporting Processes

The build process never exists in a vacuum. It is the hub or funnel-point between development's coding efforts and testing and roll-out. There are a few tools which can support or hinder an effective build process:

- **Source Code Repository:** All modern development teams need a source-code management tool. There is no longer any reason to live with the risk of one developer overwriting the work of another developer accidentally. A central repository of source code is essential for a consistent, repeatable build process. The system should have version control, check-in/check-out support, and rollback of modules to previous versions. If you do not have such a system, I recommend purchasing and using one as soon as possible; it will improve team productivity more than any change to your build process.
- **Defect and Fix Tracking:** Having a consistent build process means that defects can be more easily recreated, and their fixes can be tracked. In a simple system, this can be a spreadsheet maintained by a single tester. In a complex system, a networked database can allow multiple people to track problems and resolutions across releases. By tying defects and fixes to the build numbers, it is possible to tie them to specific code fixes and code changes.
- **Automated Testing:** Microsoft treats automated testing as an essential part of the build process, but it can be addressed separately. If you develop automated testing scripts, it is possible to run these tests after every build. In an ideal situation, the build begins as developer's finish their day, it runs overnight, and automated scripts execute basic functions before the developers return to work the next morning. If there are problems, developers can be certain that code checked in yesterday caused the problem, and quickly fix it.

Ask yourself if there is there anything else you can automate. Automated compiles, with pager or e-mail notification of errors are very possible. So are automated installation scripts and tests. Time spent implementing simple, automated procedures can have huge payoff.

### Try Something Different

People like to stick with what they know. As a rule, things that worked in the past also work in the future, so it is comfortable to keep things the same. It is worth trying something very different now and again, though. Talk to your peers, read books, and try some new ideas.

In one of my jobs, a manager adopted a new technique from Rapid Development by Steve McConnell, called the "Daily Build and Smoke Test". The team began an automated compile at the close-of-business every working day, and spent two or three hours every morning running a "shakedown" script. The script exercised all of the fundamental features of the system and ensured that yesterdays' build could operate without burning out or "smoking". This process began very early on; starting when the basic services could compile and run.

The project was extremely complex, with many interrelated screens, and I expected a long, difficult debugging process. The daily build and smoke test unveiled architectural issues early on, before the code was complex and before the the programmers had written much code using the architecture. Microsoft builds and tests products frequently, which is why their operating system build numbers area typically over 1,000. Catching problems early saves time and money. By the time the system was feature complete and entering the testing and fix phase, the application was stable and functional. The team did not suffer any late-stage integration pain, because the system components were integrated and tested as soon as they could compile.

Taking a chance on a new process or procedure can lead to huge rewards. Although the team was not collecting any metrics to measure the performance gains, we knew the process was right; we knew it helped. In my experience, improvements with the build process are not incremental, they are dramatic, leading to other order-of-magnitude improvements in your testing, development, and delivery processes.

### **References**

Jim McCarthy, Dynamics of Software Development (Redmond, WA: Microsoft Press, 1995)

Steve McConell, Rapid Development (Redmond, WA: Microsoft Press, 1996)

Edward Yourdon, Death March: The Complete Software Developer's Guide to Surviving "Mission Impossible" Projects (Upper Saddle River, NJ: Prentice Hall PTR, 1997)

*About the Author: Alex S. Brown is a Project Manager at Chubb & Son. He is a member of PMI, IEEE Computer, and ACM. You can contact Alex at alexsbrown@alexbrown.com.*

© Alex S. Brown, 2001