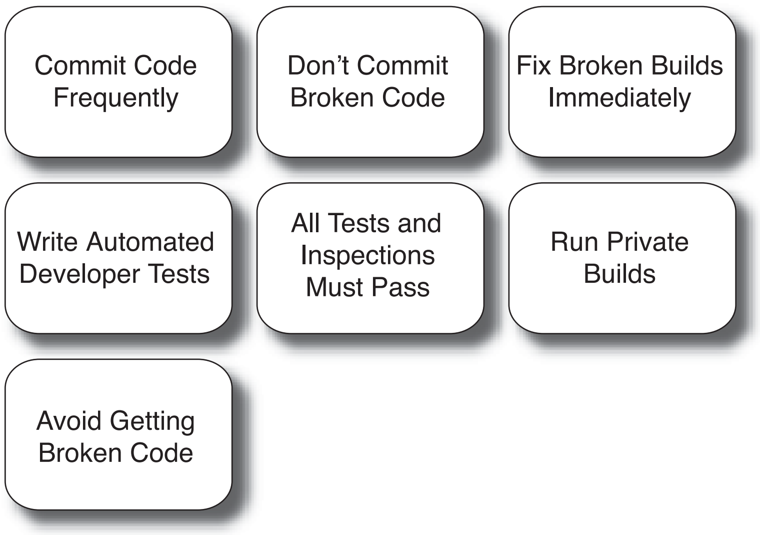
Chapter 2
Introducing Continuous Integration


Commit Code Frequently
Don't Commit Broken Code
Fix Broken Builds Immediately
Write Automated Developer Tests
All Tests and Inspections Must Pass
Run Private Builds
Avoid Getting Broken Code

Assumption is the mother of all screw-ups.
—WETHERN'S LAW OF SUSPENDED JUDGMENT

Early in my career, I learned that developing good software comes down to consistently carrying out fundamental practices *regardless of the particular technology.* In my experience, one of the most significant problems in software development is *assuming.* If you assume a method will be passed the right parameter value, the method will fail. Assume that developers are following coding and design standards and the software will be difficult to maintain. Assume configuration files haven't changed, and you'll spend precious development hours needlessly hunting down problems that don't exist. When we make assumptions in software development, we waste time and increase risks.

Reducing Assumptions

Continuous Integration can help reduce assumptions on a project by rebuilding software *whenever a change occurs* in a version control system.

We may think that the latest, greatest technology will be the "silver bullet" to solve all of our problems, but it will not. At one company, one of my initial responsibilities was to incorporate good software development practices into the company—by example. Over time, we were able to implement many widely accepted practices for developing good software into the projects. Having worked on many different projects that used different methodologies, I have found that, in general, iterative projects—using the Rational Unified Process (RUP) and eXtreme Programming (XP), in my case—work best, because risks are mitigated all along the way. Developing software requires planning for change, continuously observing the results, and incrementally course-correcting based on the results. This is how CI operates. CI is the embodiment of tactics that gives us, as software developers, the ability to make changes in our code, knowing that if we break software, we'll receive *immediate feedback.* This immediate feedback gives us time to course-correct and adjust to change more rapidly.

CI is about the fundamentals. It may not be the most glamorous activity in software development, but integrating software is vitally important in today's complex projects. Seldom do the *users* of the software say to me, "Wow, I really like the way you integrated the software in the last release." And since that doesn't happen, it may seem like it isn't worthwhile to make these efforts behind the scenes. However, anyone who has developed software using a practice such as CI is empowered by a consistent and repeatable build process kicked off when a change occurs to the version control repository.

CI as a Centerpiece for Quality

Some see CI as a process of simply putting software components together. We see CI as the centerpiece of software development, as it ensures the health of software through running a build with every change. Determining the quality of software can be as easy as checking the latest integration build.

Spending *some* time on the nonglamorous fundamental activities in software development means there is *more* time to spend on the challenging, thought-provoking activities that make our jobs interesting and fun. If we don't focus on the fundamentals, such as defining the development environment and building the software, we'll be forced to perform low-level tasks later, usually at the most inconvenient times (immediately before software goes to production, for example). This is when mistakes happen as well. The discipline involved in keeping the build "in the green" frees you from worrying about whether everything is still working. It's like exercising—yes, it takes self-discipline; yes, it can be painful work—but it keeps you in shape to play in the big game, when it counts.

This chapter attempts to answer the questions that you may have when making the decision to implement the practices of CI on a project. It provides an overview of the advantages and disadvantages of CI, and covers how CI complements other software development practices. CI is not a practice that can be handed off to a project's "build master" and forgotten about. It affects every person on the software development team, so we discuss CI in terms of what all team members must practice to implement it.

What's a day of work like using CI? Let's examine Tim's experiences.

A Day in the Life of CI

As Tim opens the door to his company's suite, he views the widescreen monitor displaying real-time information for his project. The monitor shows him that the last integration build ran successfully a few minutes ago on the CI server. It shows a list of the latest quality metrics, including coding/design standard adherence, code duplication, and so on. Tim is one of 15 developers on a Java project creating management software for an online brewery. See Figure 2-1 for a visualization of some of the activities in Tim's day.

Starting his day, Tim refactors a subsystem that was reported to have too much duplicate code based on the latest reports from the CI server. Prior to committing his changes to Subversion, he runs a **private build,** which compiles and runs the unit tests against the newest source code. After running this build on his machine, he commits his changes to Subversion. All the while, the CruiseControl CI server is polling the Subversion repository. A few minutes later, the CI server discovers the changes that Tim committed and runs an integration build. This integration build runs automated inspection tools to verify that all code adheres to the coding standard. Tim receives an e-mail about a coding standard violation, quickly makes the changes, and checks the source code back into Subversion. The CI server runs another build and it is successful. By reviewing the Web reports generated by the CI server, Tim finds that his recent code refactoring successfully reduced the amount of duplicate code in his subsystem.

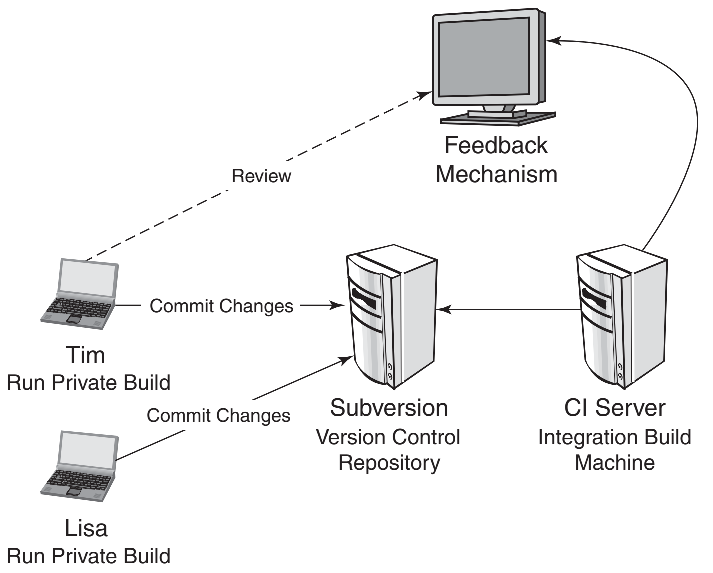


FIGURE 2-1 A day in the life

Later in the day, another developer on the project, Lisa, runs into Tim's office.

Lisa: I think the changes you made earlier today broke the last build!

Tim: Hmm…but, I ran the tests.

Lisa: Oh, I didn't have time to write tests.

Tim: Are you following the code coverage metric we have established for the project?

Because of this discussion, they decided to fail the integration build if their code coverage was below 85%. Furthermore, Lisa wrote a test for the defect and fixed the problem she discovered because of her conversation with Tim. The integration build continued to stay "in the green."

Terms of the Trade

automated—A "hands-off" process. Once a *fully automated* process begins, no user intervention is required. Systems administrators call this a "headless" process.

build—A set of activities performed to generate, test, inspect, and deploy software.

continuous—Technically, *continuous* means something that, once started, never stops. This would mean the build runs all the time; however, this isn't the case. Continuous, in the context of CI, is more like *continual,* and in the case of CI servers, a process continually runs, polling for changes to the version control repository. If the CI server discovers changes, it executes a build script.

Continuous Integration—"A software development practice where members of a team integrate their work frequently, usually each person integrates at least daily—leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly."[1]

1. From www.martinfowler.com/articles/continuousIntegration.html.

development environment—The environment in which software is written. This can include the IDE, build scripts, tools, third-party libraries, servers, and configuration files.

inspection—Analysis of source code/bytecode for the internal quality attributes. In the context of this book, we refer to the automated aspects (static and runtime analysis) as **software inspection.**

integration—The act of combining separate source code artifacts together to determine how they work as a whole.

integration build—An integration build is the act of combining software components (programs and files) into a software system. This build includes multiple components on bigger projects or only low-level compiled source files on smaller projects. In our everyday life, we tend to use the terms *build* and *integration build* interchangeably, but for the purposes of this book we make the distinction that an *integration build* is performed by a separate integration build machine.

private (system) build—Running a build locally on your workstation before committing your changes to the version control repository, to lessen the chances that your recent changes break the integration build.[2]

quality—The Free On-Line Dictionary of Computing[3] defines quality as "an essential and distinguishing attribute of something..." and "superior grade." The term *quality* is often overused, and some seem to think it is based on perception. In this book, we take the stance that quality is a measurable specification just like any other. This means you can identify specific metrics of quality, such as maintainability, extensibility, security, performance, and readability.

release build—Readies the software for release to users. It may occur at the end of an iteration or some other milestone, and it must include any acceptance tests and may include more extensive performance and load tests.

2. Based on *Software Configuration Management Patterns* by Stephen Berczuk and Brad Appleton.

3. At www.thefreedictionary.com.

risk—The potential for a problem to occur. A risk that has been realized is known as a **problem.** We focus on the higher-priority risks (damage to our interests and goals) that have the highest likelihood of occurring.

testing—The general process of verifying that software works as designed. Furthermore, we define developer tests into multiple categories, such as unit tests, component tests, and system tests, all of which verify that objects, packages, modules, and the software system work as designed. There are many other types of tests, such as functional and load tests, but from a CI perspective, all unit tests written by developers, at a minimum, are executed as a part of a build (although builds may be staged to run fast tests first followed by slower tests).

What Is the Value of CI?

At a high level, the value of CI is to:

- Reduce risks
- Reduce repetitive manual processes
- Generate deployable software at any time and at any place
- Enable better project visibility
- Establish greater confidence in the software product from the development team

Let's review what these principles mean and what value they offer.

Reduce Risks

By integrating many times a day, you can reduce risks on your project. Doing so facilitates the detection of defects, the measurement of software health, and a reduction of assumptions.

- **Defects are detected and fixed sooner**—Because CI integrates and runs tests and inspections several times a day, there is a greater chance that defects are discovered *when they are introduced* (i.e., when the code is checked into the version control repository) instead of during late-cycle testing.

- **Health of software is measurable**—By incorporating continuous testing and inspection into the automated integration process, the software product's health attributes, such as complexity, can be tracked over time.
- **Reduce assumptions**—By rebuilding and testing software in a clean environment using the same process and scripts on a continual basis, you can reduce assumptions (e.g., whether you are accounting for third-party libraries or environment variables).

CI provides a safety net to reduce the risk that defects will be introduced into the code base. The following are some of the risks that CI helps to mitigate. We discuss these and other risks in the next chapter.

- Lack of cohesive, deployable software
- Late defect discovery
- Low-quality software
- Lack of project visibility

Reduce Repetitive Processes

Reducing repetitive processes saves time, costs, and effort. This sounds straightforward, doesn't it? These repetitive processes can occur across all project activities, including code compilation, database integration, testing, inspection, deployment, and feedback. By automating CI, you have a greater ability to ensure all of the following.

- The process runs the same way *every time.*
- An ordered process is followed. For example, you may run inspections (static analysis) before you run tests—in your build scripts.
- The processes will run every time a commit occurs in the version control repository.

This facilitates

- The reduction of labor on repetitive processes, freeing people to do more thought-provoking, higher-value work

- The capability to overcome resistance (from other team members) to implement improvements by using automated mechanisms for important processes such as testing and database integration

Generate Deployable Software

CI can enable you to release deployable software at *any point in time.* From an outside perspective, this is the most obvious benefit of CI. We could talk endlessly about improved software quality and reduced risks, but deployable software is the most tangible asset to "outsiders" such as clients or users. The importance of this point *cannot* be overstated. With CI, you make small changes to the source code and integrate these changes with the rest of the code base on a regular basis. If there are any problems, the project members are informed and the fixes are applied to the software *immediately.* Projects that do not embrace this practice may wait until immediately prior to delivery to integrate and test the software. This can delay a release, delay or prevent fixing certain defects, cause new defects as you rush to complete, and can ultimately spell the end of the project.

Enable Better Project Visibility

CI provides the ability to notice trends and make effective decisions, and it helps provide the courage to innovate new improvements. Projects suffer when there is no real or recent data to support decisions, so everyone offers their best guesses. Typically, project members collect this information manually, making the effort burdensome and untimely. The result is that often the information is never gathered. CI has the following positive effects.

- **Effective decisions**—A CI system can provide just-in-time information on the recent build status and quality metrics. Some CI systems can also show defect rates and feature completion statuses.
- **Noticing trends**—Since integrations occur frequently with a CI system, the ability to notice trends in build success or failure, overall quality, and other pertinent project information becomes possible.

Establish Greater Product Confidence

Overall, effective application of CI practices can provide greater confidence in producing a software product. With every build, your team knows that tests are run against the software to verify behavior, that project coding and design standards are met, and that the result is a functionally testable product.

Without frequent integrations, some teams may feel stifled because they don't know the impacts of their code changes. Since a CI system can inform you when something goes wrong, developers and other team members have more confidence in making changes. Because CI encourages a single-source point from which all software assets are built, there is greater confidence in its accuracy.

What Prevents Teams from Using CI?

If CI has so many benefits, then what would prevent a development team from continuously integrating software on its projects? Often, it is a combination of concerns.

- **Increased overhead in maintaining the CI system**—This is usually a misguided perception, because the need to integrate, test, inspect, and deploy exists regardless of whether you are using CI. Managing a robust CI system is better than managing manual processes. *Manage the CI system or be controlled by the manual processes.* Ironically, complicated multiplatform projects are the ones that need CI the most, yet these projects often resist the practice as being "too much extra work."
- **Too much change**—Some may feel there are too many processes that need to change to achieve CI for their legacy project. An incremental approach to CI is most effective; first add builds and tests with a lower occurrence (for example, a daily build), then increase the frequency as everyone gets comfortable with the results.
- **Too many failed builds**—Typically, this occurs when developers are not performing a private build prior to committing their code to the version control repository. It could be that a devel-

oper forgot to check in a file or had some failed tests. Rapid response is imperative when using CI because of the frequency of changes.

- **Additional hardware/software costs**—To effectively use CI, a separate integration machine should be acquired, which is a nominal expense when compared to the more expensive costs of finding problems later in the development lifecycle.
- **Developers should be performing these activities**—Sometimes management feels like CI is just duplicating the activities that developers should be performing anyway. Yes, developers should be performing some of these activities, but they need to perform them *more effectively and reliably in a separate environment.* Leveraging automated tools can improve the efficiency and frequency of these activities. Additionally, it ensures that these activities are performed in a clean environment, which will reduce assumptions and lead to better decision making.

How Do I Get to “Continuous” Integration?

It’s often surprising to learn the level of automation of most development organizations. Developers spend most of their time automating processes for their users, yet don’t always see ways to automate their own development processes. Sometimes teams believe their automation is sufficient because they’ve written a few scripts to eliminate some steps in the development process. The following is a typical scenario.

Joan (Developer): …I already automated that. I wrote some batch scripts that drop and recreate the database tables.

Sue (Technical Lead): That’s great. Did you apply it to the CVS repository?

Joan: No.

Sue: Did you make it a part of the build script?

Joan: No.

Sue: So, if it’s not a part of the CI system then it’s not really automated yet… right?

CI is not just the process of gathering a few scripts together and running them all the time. In the preceding scenario, it's great that Joan wrote those automation scripts, but in order for them to actually add value to the end product, they must be added to the version control repository and made a working part of the build process. Figure 2-2 illustrates the steps to making a process continuous.

These steps can be applied one by one to virtually every activity you conduct on a project.

- **Identify**—Identify a process that requires automation. The process may be in the areas of compilation, test, inspection, deployment, database integration, and so on.
- **Build**—Creating a build script makes the automation repeatable and consistent. Build scripts can be constructed in NAnt for the .NET platform, Ant for the Java platform, and Rake for Ruby, just to name a few.
- **Share**—By using a version control system such as Subversion, you make it possible for others to use these scripts/programs. Now the value is being spread consistently across the project.
- Make it **continuous**—Ensure that the automated process is run with every change applied, using a CI server. If your team has the discipline, you can also choose to manually run the build with every change applied to the version control system.

Here is an acrostic to help you remember and communicate this: "**I** **B**uild **S**o **C**onsistently"—for **I**dentify, **B**uild, **S**hare, and **C**ontinuous.

Aim for incremental growth in your CI system. This is simple to implement, the team gets more motivated as each new item is added, and you can better plan what you need next based on what's working

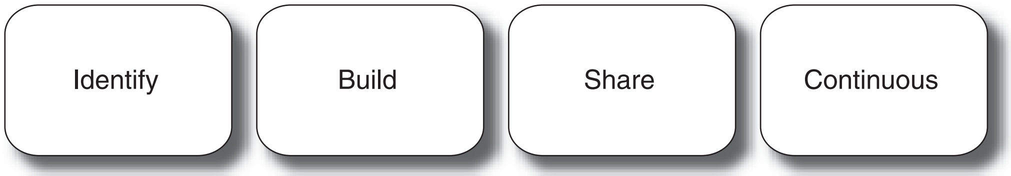


FIGURE 2-2 Getting to CI— "**I** **B**uild **S**o **C**onsistently"

Is It Continuous Compilation or Continuous Integration?

I've worked with a number of organizations on implementing CI, and on several occasions I've heard the reply, "Yes, we do CI." Of course, I think, "Great!" and then ask a few questions. How much code coverage do you have with your tests? How long does it take to run your builds? What is your average code complexity? How much code duplication do you have? Are you labeling your builds in your version control repository? Where do you store your deployed software?

I discover that what they've been doing all along is more like a "continuous compilation," in which they've set up a tool like Cruise-Control to poll their version control repository (e.g., CVS) for changes. When it detects changes, it retrieves the source code from CVS, compiles the code, and sends an e-mail if anything goes wrong. Automatically compiling the software system on a separate machine is better than nothing at all, but doing that isn't going to provide all of the benefits of a full-featured CI system.

so far. Often, attempting to throw everything into a CI system immediately can be a bad move, just like refactoring a lot of code at once isn't the best approach when writing software. Get it to work first, get developers using it, and then add other automated processes as needed based on the project risks.

When and How Should a Project Implement CI?

It is best to implement CI early in the project. Although possible, it is more difficult to implement CI late in a project, as people will be under pressure and more likely to resist change. If you do implement CI later in a project, it is especially important to start small and add more as time permits.

There are different approaches to setting up the CI system. Though you eventually want a build to run on every change to the system, you can start by running a build on a *daily* basis to get the practice going in your organization. Remember: *CI is not just a technical implementation; it is also an organizational and cultural implementation.* People often resist change, and the best approach for an organization may be to add these automated mechanisms to the process piece by piece.

At first the build can just compile the source code and package the binaries without executing the automated regression tests. This can be effective, initially, if the developers are unfamiliar with an automated testing tool. Once this is in place and developers have learned the testing tool, you can move closer to the benefits of CI: running these tests (and inspections) with every change.

The Evolution of Integration

Is CI the newest, latest, "whiz-bang" approach to software development? Hardly. CI is simply an advance in the evolution of integrating software. When software programs consisted of a few small files, integrating them into a system was not much of a problem. The practice of performing nightly builds has been described as a best practice for years. Similar practices have been discussed in other books and articles. In the book *Microsoft Secrets*, Michael A. Cusumano and Richard W. Selby discuss the practice of daily builds at Microsoft. Steve McConnell, in *Software Project Survival Guide,* discusses the practice of the "Daily Build and Smoke Test" as part of a software development project.

In *Object Solutions: Managing the Object-Oriented Project,* Grady Booch writes, "The macro process of object-oriented development is one of 'continuous integration'… At regular intervals, the process of 'continuous integration' yields executable releases that grow in functionality at every release... It is through these milestones that management can measure progress and quality, and hence anticipate, identify, and then actively attack risks on an ongoing basis." With the advent of XP and other Agile methodologies, and with the recommended practice of CI, people began to take notice of the concept of not just daily, but "continuous," builds.

The practice of CI continues to evolve. You'll find the practice in almost every XP book. Often, when people discuss the practice of CI, they refer to Martin Fowler's seminal "Continuous Integration" article.[4]

As hardware and software resources continue to increase, you'll find that more processes will become a part of what is considered to be CI.

How Does CI Complement Other Development Practices?

The practice of CI complements other software development practices, such as developer testing, adherence to coding standards, refactoring, and small releases. It doesn't matter if you are using RUP, XP, RUP with XP, SCRUM, Crystal, or any other methodology. The following list identifies how the practice of CI works with and improves these practices.

- **Developer testing**—Developers who write tests most often use some xUnit-based framework such as JUnit or NUnit. These tests can be automatically executed from the build scripts. Since the practice of CI advocates that builds be run any time a change is made to the software, and that the automated tests are a part of these builds, CI enables automated regression tests to be run on the entire code base whenever a change is applied to the software.
- **Coding standard adherence**—A coding standard is the set of guidelines that developers must adhere to on a project. On many projects, ensuring adherence is largely a manual process that is performed by a code review. CI can run a build script to report on adherence to the coding standards by running a suite of automated static analysis tools that inspect the source code against the established standard whenever a change is applied.
- **Refactoring**—As Fowler states, refactoring is "the process of changing the software system in such a way that it does not alter

4. See www.martinfowler.com/articles/continuousIntegration.html.

the external behavior of the code yet improves its internal structure."[5] Among other benefits, this makes the code easier to maintain. CI can assist with refactoring by running inspection tools that identify potential problem areas at every build.

- **Small releases**—This practice allows testers and users to get working software to use and review as often as required. CI works very well with this practice, because software integration is occurring many times a day and a release is available at virtually *any time.* Once a CI system is in place, a release can be generated with minimal effort.
- **Collective ownership**—Any developer can work on any part of the software system. This prevents "knowledge silos," where there is only one person who has knowledge of a particular area of the system. The practice of CI can help with collective ownership by ensuring adherence to coding standards and the running of regression tests on a continual basis.

How Long Does CI Take to Set Up?

Implementing a *basic* CI system along with simple build scripts for a new project may take you a few hours to set up and configure (more if you don't have any existing build scripts). As you expand your knowledge of the CI system, it will grow with the addition of inspection tools, deployments that are more complex, more thorough testing, and many other processes. These additional features tend to be added a little at a time.

For a project already in progress, it can take days, weeks, or even months to set up a CI system. It also depends upon whether people have been dedicated to work on the project. Usually you must complete many tasks when moving to a continuous, automated, and headless system such as when using a CI server. In some cases, you may be moving from batch or shell scripts to a build scripting tool such as Ant

5. Fowler, et al. *Refactoring: Improving the Design of Existing Code* (Reading, MA: Addison-Wesley, 1999).

or managing all of the project's binary dependencies. In other cases, you may have previously used your IDE for "integration" and deployment. Either way, the road map to full CI adoption could be quite a bit longer.

CI and You

In order for CI to work effectively on a project, developers must change their typical day-to-day software development habits. Developers must commit code more frequently, make it a priority to fix broken builds, write automated builds with tests that pass 100% of the time, and not get or commit broken code from/to the version control repository.

The practices we recommend take some discipline, yet provide the benefits stated throughout this chapter. The best situation is one where most project members agree that there is an exponential payback to the time and attention they pay to the practices of CI.

There are seven practices that we've found work well for individuals and teams running CI on a project.

- Commit code frequently
- Don't commit broken code
- Fix broken builds immediately
- Write automated developer tests
- All tests and inspections must pass
- Run private builds
- Avoid getting broken code

The following sections cover each practice in greater detail.

Commit Code Frequently

One of the central tenets of CI is integrating *early and often.* Developers must commit code frequently in order to realize the benefits of CI.

Waiting more than a day or so to commit code to the version control repository makes integration time-consuming and may prevent developers from being able to use the latest changes. Try one or both of these techniques to commit code more frequently.

- **Make small changes**—Try not to change many components all at once. Instead, choose a small task, write the tests and source code, run your tests, and then commit your code to the version control repository.
- **Commit after each task**—Assuming tasks/work items have been broken up so that they can be finished in a few hours, some development shops require developers to commit their code as they complete each task.

Try to avoid having everyone commit at the same time every day. You'll find that there are usually many more build errors to manage because of the collisions between changes. This is especially troublesome at the end of the day, when people are ready to leave. The longer you wait to integrate with others, the more difficult your integration will prove to be.

I Just Can't Commit

A friend runs a 25-developer project and he'd like to incorporate many CI practices, but he is experiencing challenges in getting the developers to commit code frequently. I've found that the main reason that changes are not committed frequently is because of the project culture. Sometimes developers do not want to commit their code until it is "perfect." This usually happens because their changes affect too many components. Committing code frequently to the version control repository is the only effective way to implement CI, and this means that all developers need to embrace this development practice by grabbing smaller chunks of code and breaking up their tasks into smaller work items.

Don’t Commit Broken Code

A dangerous assumption on a project is that everyone knows not to commit code that doesn’t work to the version control repository. The ultimate mitigation for this risk is having a well-factored build script that compiles and tests the code in a repeatable manner. Make it part of the team’s accepted development practice to always run a private build (which closely resembles the integration build process) before committing code to the version control repository. See the later section, Run Private Builds, for additional recommendations before committing your code.

Fix Broken Builds Immediately

A **broken build** is anything that prevents the build from reporting success. This may be a compilation error, a failed test or inspection, a problem with the database, or a failed deployment. When operating in a CI environment, these problems must be fixed immediately; fortunately, in a CI environment, each error is discovered incrementally and therefore is likely very small. Some projects have a penalty for breaking the build, such as throwing some money in a jar or placing the picture of the last developer to break the build on the company’s large-screen monitor (just kidding; hopefully no one is doing this). The project culture should convey that fixing a broken build is a top project priority. That way, not just some but every team member can then get back to what they were doing.

Write Automated Developer Tests

A build should be fully automated. In order to run tests for a CI system, the tests must be automated. Writing your tests in an xUnit framework such as NUnit or JUnit will provide the capability of running these tests in an automated fashion. Chapter 6 provides details on writing automated tests.

All Tests and Inspections Must Pass

In a CI environment, 100% of a project's automated tests must pass for your build to pass (this is a technical criterion, not an expectation that all workers or all work should be perfect). Automated tests are as important as the compilation. Everyone accepts that code that does not compile will not work; therefore, code that has test errors will not work either. Accepting code that does not pass the tests can lead to lower-quality software.

An unscrupulous developer may simply comment out the failing test. Of course, this defeats the purpose. Coverage tools assist in pinpointing source code that does not have a corresponding test. You can run a code coverage tool as part of an integration build.

The same goes for running automated software inspectors. Use a general rule set of coding and design standards that all code must pass. More advanced inspections may be added that don't fail the build, but identify areas of the code that should be investigated.

Run Private Builds

To prevent broken builds, developers should emulate an integration build on their local workstation IDE after completing their unit tests. This build allows you to integrate your new working software with the working software from all the other developers,[6] obtaining the changes from the version control repository and successfully building locally *with* the recent changes. Thus, the code each developer commits has contributed to the greater good, with code that is less likely to fail on the integration build server.

6. Some configuration management tools, such as ClearCase, have an option to automatically update your local environment with the changes from the version control repository (called "dynamic views" in ClearCase).

Keep Builds in the "Green"

I find that there are two measures of using CI effectively: number of commits and build status. Each developer (or pair) should have at least one commit to the repository per day, and the number of checkins usually demonstrates the size of the changes (more commits usually means smaller changes—and this is good). Your build status should be "green" (pass) a large percentage of the day; set this value for the team. We all get a "red" build status sometimes, but what's important is that it's changed back to green as soon as possible. Never let your team get used to waiting in the red status until this or that other project task is done. The willingness to leave the status at red for other criteria defeats much of the strength of CI.

Avoid Getting Broken Code

When the build is broken, don't check out the latest code from the version control repository. Otherwise, you must spend time developing a workaround to the error known to have failed the build, just so you can compile and test your code. Ultimately, it's the responsibility of the team, but the developers responsible for breaking the build *should* already be working on fixing their code and committing it back to the version control repository. Sometimes a developer may not have seen the e-mail on the broken build. This is when a passive feedback mechanism such as a light or sound can be useful for colocated developers. We consider it critical that all developers know the state of the code in the version control repository. For more information on continuous feedback mechanisms, see Chapter 9. An alternative, but not preferable, approach to avoiding a checkout is to use the version control system to roll back any changes since the most recent commit.

❑ ❑ ❑ ❑ ❑ ❑ ❑ ❑ ❑

Summary

Now you have the ammunition to go talk to others about CI. This chapter covered some of the basics of CI, discussed how to get to a continuous process, and pointed out all the other areas that get explored in detail in subsequent chapters. Table 2-1 summarizes seven practices to follow when using CI. The next chapter delves into the software risks that CI can help mitigate to improve quality.

TABLE 2-1 CI Practices Discussed in This Chapter

Practice	*Description*
Commit code frequently	Commit code to your version control repository at least once a day.
Don't commit broken code	Don't commit code that does not compile with other code or fails a test.
Fix broken builds immediately	Although it's the team's responsibility, the developer who recently committed code must be involved in fixing the failed build.
Write automated developer tests	Verify that your software works using automated developer tests. Run these tests with your automated build and run them often with CI.
All tests and inspections must pass	Not 90% or 95% of tests, but *all* tests must pass prior to committing code to the version control repository.
Run private builds	To prevent integration failures, get changes from other developers by getting the latest changes from the repository and run a full integration build locally, known as a private system build.
Avoid getting broken code	If the build has failed, you will lose time if you get code from the repository. Wait for the change or help the developer(s) fix the build failure and then get the latest code.

Questions

Practicing CI is more than installing and configuring some tools. How many of the following items are you consistently performing on your project? How many of the other CI practices can improve your development capabilities?

- On average, is everyone on your team committing code at least once a day? Are you employing techniques to make it easier to commit code often?
- What percentage of each day's integration builds is successful (that is, the most recent build run has passed)?
- Is everyone on your team running a private build before committing to the repository so that integration errors are reduced?
- Have you scripted your builds to fail if any of your tests or inspections fail?
- Is a broken integration build a priority to fix on your projects?
- Do you avoid getting the latest code from the version control system when there is a broken build?
- How often do you consider adding automated processes to your build and CI system—on a continuous or even periodic basis?