Introduction

An extremely strange, but common, feature of many software projects is that for long periods of time during the development process the application is not in a working state. In fact, most software developed by large teams spends a significant proportion of its development time in an unusable state. The reason for this is easy to understand: Nobody is interested in trying to run the whole application until it is finished. Developers check in changes and might even run automated unit tests, but nobody is trying to actually start the application and use it in a production-like environment.

This is doubly true in projects that use long-lived branches or defer acceptance testing until the end. Many such projects schedule lengthy integration phases at the end of development to allow the development team time to get the branches merged and the application working so it can be acceptance-tested. Even worse, some projects find that when they get to this phase, their software is not fit for purpose. These integration periods can take an extremely long time, and worst of all, nobody has any way to predict how long.

On the other hand, we have seen projects that spend at most a few minutes in a state where their application is not working with the latest changes. The difference is the use of continuous integration. Continuous integration requires that every time somebody commits any change, the entire application is built and a comprehensive set of automated tests is run against it. Crucially, if the build or test process fails, the development team stops whatever they are doing and fixes the problem immediately. The goal of continuous integration is that the software is in a working state all the time.

Continuous integration was first written about in Kent Beck's book Extreme Programming Explained (first published in 1999). As with other Extreme Programming practices, the idea behind continuous integration was that, if regular integration of your codebase is good, why not do it all the time? In the context of integration, "all the time" means every single time somebody commits any

change to the version control system. As one of our colleagues, Mike Roberts, says, "Continuously is more often than you think" [aEu8Nu].

Continuous integration represents a paradigm shift. Without continuous integration, your software is broken until somebody proves it works, usually during a testing or integration stage. With continuous integration, your software is proven to work (assuming a sufficiently comprehensive set of automated tests) with every new change—and you know the moment it breaks and can fix it immediately. The teams that use continuous integration effectively are able to deliver software much faster, and with fewer bugs, than teams that do not. Bugs are caught much earlier in the delivery process when they are cheaper to fix, providing

significant cost and time savings. Hence we consider it an essential practice for professional teams, perhaps as important as using version control.

The rest of this chapter describes how to implement continuous integration. We'll explain how to solve common problems that occur as your project becomes more complex, listing effective practices that support continuous integration and its effects on the design and development process. We'll also discuss more advanced topics, including how to do CI with distributed teams.

Continuous integration is dealt with at length in a companion volume to this one: Paul Duvall's book Continuous Integration (Addison-Wesley, 2006). If you want more detail than we provide in this chapter, that is the place to go.

This chapter is mainly aimed at developers. However, it also contains some information that we think will be useful for project managers who want to know more about the practice of continuous integration.