6

Private Workspace

A government clerk's room, showing a desk with books, telephone and directory, and a desk lamp on it. Washington, D.C., 1939.

Photo by David Meyers. Library of Congress, Prints & Photographs Division, FSA-OWI Collection, Reproduction Number: LC-USF33-015598-M2.

In *Active Development Line (5)*, you and other developers make frequent changes to the code base, both to the modules you are working on and to modules you depend on. You want to be sure you are working with the latest code, but because people don't deal well with uncontrolled change, you want to be in control when you start working with other developers' changes. This pattern describes how you can reconcile the tension between always developing with a current code base and the reality that people cannot work effectively when their environment is in constant flux.

➿ *How do you do keep current with a continuously changing codeline and also make progress without being distracted by your environment changing out from under you?*

Developers need a place where they can work on their code, isolated from outside changes, while they are finishing a task.

When a team develops software, people work in parallel, with the hope that the team gets work done more quickly than any individual. Each individual makes changes in parallel with the other team members. You now have the problem of managing and integrating these parallel streams of change. Writing and debugging code, on the other hand, is a fairly linear activity. Because in team development, concurrent changes are happening to the codeline while you are working on your specific changes, there is a tension between keeping up to date with the current state of the codeline and the human tendency to work best in an environment of minimal change. Changes that distract you from your primary purpose interrupt your flow.

DeMarco and Lister define "flow" as "a condition of deep, nearly meditative involvement"(DeMarco and Lister 1987). In *Peopleware*, the authors discuss flow as noise and task-related interruptions, but integrating a change that is not related to the task at hand can have a similar effect.

Developing software in a team environment involves the following steps:

- Writing and testing your code changes
- Integrating your code with the work that other people were doing

There are two extreme approaches to managing parallel change: literal continuous integration and delayed integration.

You can integrate every change team members make as soon as they make it. This is the clearest way to know whether your changes work with the current state of the codeline. The downside of this "continuous integration" into your workspace approach is that you may spend much of your time integrating, handling changes tangential to your task. Frequent integration helps you isolate when a flaw appeared. Integrating too many changes at once can make it harder to isolate where the flaw is because it can be in one of the many changes that have happened since you integrated. Figure 6–1 shows this concept.

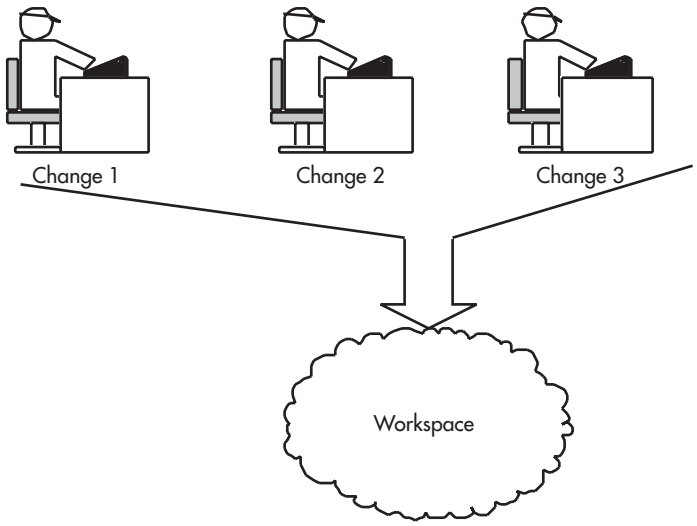

FIGURE 6–1. Combining changes at once

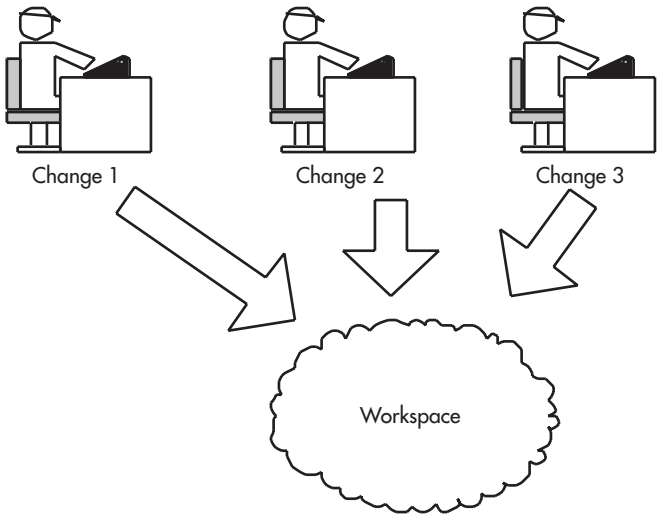

FIGURE 6–2. Integrating each change as it happens

Even when you do "continuous integration," as when you are doing Extreme Programming, you really integrate in discrete steps, as when a day's work is complete. Figure 6–2 shows this case.

You can integrate at the last possible moment. This makes it simplest for you, the developer, while you are working, but it means that you may have many outside integration issues to deal with, meaning that it will take longer to integrate at the end.

You can "help" developers keep up to date by having them work from a shared source/release area, keeping only local copies of the components they are modifying. Figure 6–3 illustrates this. But you don't want things to change unexpectedly. Also, a change in one of the other components can affect your work. If you are coding in a language such as C++, a change in a header can cause a compilation problem. A change in the source can cause a behavior problem. Even with a highly modular architecture, components interact, making it hard to get consistent results across a change.

Sometimes you are working on things other than the latest code base. You must interrupt your work on the current release to work on the code at an

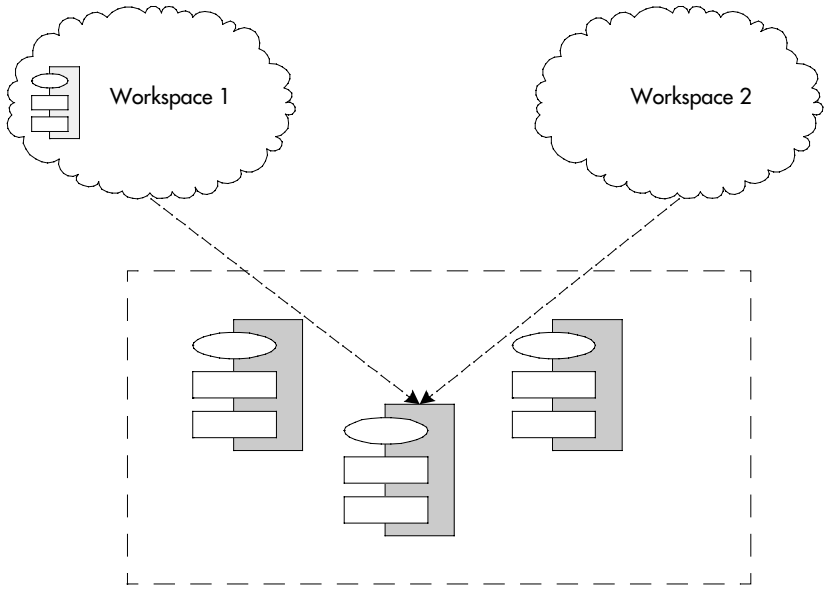

FIGURE 6–3. Sharing some components between workspaces

A SIMPLE PLAN

To some, this sounds like an easy-to-solve problem with an obvious solution. When I was interviewing for a job at a start-up company six years into my career, I discovered that some obvious solutions are easy to miss if you are not thinking about the context. The company had fully bought into the idea of nightly builds. The problem was that each developer worked from a shared product area, so after a night of working on a problem, you could come in the next day to find that your development environment had changed dramatically and then have to spend half the day simply getting to where you were the night before.

This illustrates one problem with blindly following a "good idea" without thinking through the reasons for using it.

earlier point in time. Or you may need to experiment with a new feature. Sometimes you can't be up to date and still do your work.

You can also avoid the problems of continuous updates by taking a snapshot of the entire system and performing all your coding tasks against the snapshot. This overly conservative approach can cause problems when you get behind the leading edge of changes. You may find yourself introducing problems into the global environment.

You need a way to control the rate of change in the code you are developing without falling too far out of step with the evolving codeline.

ISOLATE YOUR WORK TO CONTROL CHANGE

> *Do your work in a* private workspace, *where you control the versions of code and components you are working on. You will have total control over when and how your environment changes.*

Every team member should be able to set up a workspace where there is a consistent version of the software. A concise definition of a workspace is "a copy of all the 'right' versions of all the 'right' files in the 'right' directories" (White 2000). A workspace is also a place "where an item evolves through many temporary and inconsistent states until is checked into the library" (Whitgift 1991). You should have total control of when parts of the system change. You control when changes are integrated into your workspace. The most common situation is when you are working on the tip of the codeline along with other team members, but when you are working on a version that is not the latest, you can re-create any configuration necessary.

A *private workspace* comprises the following.

- Source code you are editing.
- Any locally built components.
- Third-party derived objects that you cannot or do not wish to build.
- Built objects for all the code in the system. You can build these yourself, have references to a shared repository (with the correct version), or have copies of built objects.

- Configuration and data that you need to run and test the system.
- Build scripts to build the system in your workspace.
- Information identifying the versions of all the components in the system.

A *private workspace* should not contain the following.

- Private versions of systemwide scripts that enforce policy. These should be in a shared binary directory so that all users get the latest functionality.
- Components that are in version control but that you copied from somewhere else. You should be able to reproduce the state of your workspace consistently when you are performing a task, by referencing a version identifier for every component in the workspace.
- Any tools (compilers, and so on) that must be the same across all versions of the product. If different versions of the product require different versions of tools, the build scripts can address this by selecting the appropriate tool versions for a configuration.

In addition, a *private workspace* can include tools that facilitate your work, as long as the tools are compatible with the work style of the team.

To do your coding for mainline development, follow a procedure similar to this.

1. Get up to date. Update the source tree from the codeline you are working on so that you are working with the current code and build, or repopulate the workspace from the latest system build. If you are working on a different branch or label, create a new *private workspace* from that branch.
2. Make your changes. Edit the components you need to change.
3. Do a *Private System Build (8)* to update any derived objects.
4. Test your change with a *Unit Test (14)*.
5. Update the workspace to the latest versions of all other components by getting the latest versions of all components you have not changed.
6. Rebuild. Run a *Smoke Test (13)* to make sure that you have not broken anything.

If your system is small enough, you can simply get the source and any binary objects for the correct configuration of all the product components and build the entire system. You might also consider getting the latest code from the *Mainline (4)* and building the entire system if it does not take too long. This will ensure that the system you are running matches the source code. With a good incremental build environment, doing this should work rather well, allowing for, perhaps, the one-time cost of the whole system build.

In more complex systems or where you are especially intolerant of problems, populate the environment by getting the source and object files from a known good build (*Named Stable Bases (20)*). You can also get all the source files from the *Mainline (4)* because this will probably simplify debugging. Get whatever external components you need from the *Third Party Codeline (10)*. All these components should be of the correct configuration (version, label, and so on) for the system you are working on. Get private versions of all the source components you will be changing.

If you are working on multiple tasks, you can have multiple workspaces, each with its own configuration. For example, you can have a release 1.1 workspace to fix problems in the old release while doing new development in a release 2 workspace. These can be separate and complete workspaces. It is not worth the effort, in most cases, to save space by factoring out common components. (For example, if component X has not changed between release 1.1 and release 2, it is worthwhile simply to have two copies of this component. If X changes in release 2 later on, it will be easy to update the release 2 workspace without affecting the release 1 workspace.

Be sure that any tests, scripts, tools, and so on use the correct execution paths so that they run with the correct workspace version and not with a component from another workspace or an installed version of the product. One way to do this is to deploy all local components in one binary directory and put the current directory in the path. Another way is to start tests in a script that sets the environment.

Some component environments, such as COM, define certain items machinewide, so be sure to have a mechanism to switch between workspaces by unregistering and registering the appropriate servers.

To be sure that you have built all dependencies, do a *Private System Build (8)*. Check that your changes integrate successfully with the work others have

done in the meantime by getting the latest code from the *Mainline (4)* (exclusive of changes you have made). If you are working on multiple tasks at one time, your workspace should have many workspaces.

One risk with a *Private Workspace (6)* is that developers will work with old "known" code too long, and they will be working with outdated code. You can protect yourself from this by doing a periodic *Private System Build (8)* and making sure that changes do not break the build or fail the *Smoke Test (13)*. (The sidebar Update Your Workspace to Keep Current discusses the *workspace update* in more detail.)

The easiest way to avoid becoming out of date is to do fine-grained tasks, checking in your changes after each one and updating your workspace before starting a new task. Some people find it useful to establish a discipline of creating a brand-new workspace periodically to avoid problems that stray files might cause and preventing the "works for me" syndrome. This is not ideal but is an adaptation to the reality that some version control tools do an imperfect job of updating, particularly when you move files within the system.

Having a *Private Workspace (6)* does take more space than working with shared source, but the simplicity it adds to your work is worth it.

An automated build process should also have its own workspace. Setting up this workspace would always get all the updates, if you are doing a "latest" build.

Good tool support makes managing a combination of shared and private components easy, but you can get quite far by using basic version control tools and scripts. For example, if your system can be built quickly but uses some third-party components, your checkout process can populate your workspace from version control with all the source from your system and the built objects for the third-party components. After you build your product code, you will have a complete system.

A *Smoke Test (13)* allows you to check that your changes don't break the functionality of the system in a major way. A well-designed smoke test will help you minimize the amount of code you need to keep in your workspace and rebuild, because the Smoke Tests should test the features that clients of your module expect.

Some work touches large parts of the code base and takes a long time to finish. In these cases, a *Task Branch (19)* may be the more appropriate approach.

Depending on your specific goal, there are a number of variations to this pattern, including a developer workspace, an integration workspace, and a task workspace, in which case a developer has a number of workspaces in the area concurrently.

Variations of a workspace are used for specific purposes—for example, an integration workspace, which is where changes are combined with the current state of the system, built, and tested. This can also be called a build workspace and may exist on the integration or build machine.

UPDATE YOUR WORKSPACE TO KEEP CURRENT

After a workspace has been populated, the codeline may continue to evolve. If the work in your workspace is isolated for too long, the versions in the workspace can become outdated. A *workspace update* operation will "refresh" the outdated versions in your workspace, replacing them with the versions from the latest stable state of the codeline. If any of the files you changed are also among the set of "newer" files from the codeline, merge conflicts may occur and will need to be reconciled.

You should do a workspace update before you merge your changes back to the codeline during a *Task Level Commit (11)*. You will need to rebuild using a *Private System Build (8)*, or at least recompile immediately after the update, to find and fix quickly any inconsistencies introduced by the new changes. If desired, immediately before updating your workspace, checkpoint it using a label or *Private Versions (16)* to ensure that you can roll back to its previous state.

You may also update your workspace at known stable points, as well as right before you are about to check out a new set of files, to ensure that your workspace remains stable without growing "stale." This enables you to find out early on if any recently committed changes conflict with any changes in your workspace. You may then reconcile those changes in your *Private Workspace (6)* at incremental intervals, instead of waiting until the end to do all of them at once.

UNRESOLVED ISSUES

Once you have stability for yourself, you still need to prevent introducing errors into the system when you check in your changes. *Private System Build (8)* lets you check that your system does not break the build and also enables you to do an incremental build for the parts of your system when you do an incremental update from version control for other components.

You need to populate your workspace from a *Repository (7)* containing all the source and related components. Externally provided components need to come from a *Third Party Codeline (10)*.

Once you are done with your local work, it must be incorporated into the rest of the system in an *Integration Build (9)*.

FURTHER READING

- Brian White in *Software Configuration Management Strategies and Rational ClearCase: A Practical Introduction* (White 2000) has a good description of the various types of workspaces that ClearCase supports (ClearCase calls them "views"). He says that "one of the essential functions of an SCM tool is to establish and manage the developers' working environment, often referred to as a 'workspace' or a 'sandbox.'"
- Private workspaces are a common practice in successful development organizations, so common that they are often not described as such. Managing change, consistent build practices, and other essential components of private workspaces are all part of the practices that classic books such as *Code Complete* (McConnell 1993) and *Rapid Development* (McConnell 1996), among others, describe.