Getting Ready to Work: Patterns for a Developer's Workspace

Steve Berczuk
Verbind, Inc.
(http://www.verbind.com)
berczuk@acm.org
http://world.std.com/~berczuk

Brad Appleton
Motorola NSS
(bradapp@enteract.com)

Ralph Cabrera
AGCS
(cabrerar@agcs.com)

Abstract

The way software developers work day-to-day has a major impact on the speed and quality of product development. Workspace management and version control bridge the gap between architecture, process, and the code. But these issues are often ignored.

This paper presents a pattern language for constructing a productive developer workspace, tying the workspace structures together with architecture and configuration management. While organizational support for developer productivity is always helpful, you can apply these patterns, even if you are a developer working on your own.

Introduction

Much of the patterns literature is devoted to patterns about code. The patterns in *Design Patterns* [1] patterns address basic building blocks of Object-oriented systems; The patterns in Pattern-Oriented Software Architecture – A System of Patterns [2] address systems architecture issues. Many of the papers in the *Pattern Languages of Program Design* series[3-6]are about the design elements of applications. This is because the code embodies the ideas and functions of software systems, and is thus, more visible. However, we need to spend more time discussing the structures that enable us to build systems. In many ways this is the harder problem because it involves more than simply technical details.

Writing code as an individual does not present the coordination challenges that writing code as part of a team does. For an individual coder, the problems are all about objects, APIs, classes: technical details that developers are comfortable with. When you add other people to the project the task becomes complicated. In addition to solving a technical problem you must now also communicate with each other and coordinate your work.

The bigger picture is that development involves Product Architecture, Version Control & Branching, and Workspace Management.

A well thought out architecture can mitigate many of the tensions, such as coordination and dependency issues, caused by different people working on the same problem. The issue is compounded when you have different teams collaborating. Configuration Management and integration plans prevent people from interfering with each other's work by providing a mechanism and process for communication and coordination. Architecture, process, and other parts of the development environment all come together at the point where the developer does his coding.

Issues surrounding how you, as a developer set up your workspace are often orphans, in that t hey are not really about high-level management (there is an organization patterns language literature about that[1]) nor are they about low level coding. Version control issues are not often well understood by developers, but are rather treated as a procedure to follow. Version and configuration management are left for the "build" people to handle. But given the impact that they have on your day-to-day work environment, it is worthwhile for developers to have a better understanding of them.

As a developer you can independently exercise limited control over the development process and architecture, but a good personal process can go a long way in helping an individual be effective, even in the absence of an established group practice. This small pattern language presents an approach for building a realm in which a developer can work effectively as part of a team. Ideally, the whole organization should consider this approach, but, if it is used incrementally, one developer at a time, or one team at a time, improvements will be possible.

This paper describes workspaces, and is based on configuration management and build patterns that have appeared elsewhere [8-12].

Roadmap to the Paper

The following sections describe the concepts behind pattern languages, and how workspaces fit into the larger picture of constructing software systems. The Pattern Language starts with the *section A Pattern Language for Workspaces* on page 6. The introduction can be skipped if you are familiar with Pattern Languages and the general issues of workspace management. The section *Elements of a Workspace* on page 4 defines what we mean by *workspace*.

This paper briefly presents a scenario for building a workspace and points you to the patterns for completing the details. The patterns cover areas such as:

- Branching Policies and Techniques
- Version Control Policies and Version Management Techniques
- Build Management Procedures
- Integration Policies and Techniques.

As a developer, you may not feel that you have control over each part of the system. Even if this is so, you can still build part of the solution. The aim of the patterns is to focus on the result, not how you get there. While specific tools may help you (for example, version control systems), you can still approach the goal by other means.

Pattern Languages

Patterns are about structures that solve problems by balancing many, often conflicting, forces [13]. A Pattern language shows how structures relate, while showing you how to build those structures. To use the pattern language, look for the pattern that describes the structure you would like to set up, and then try to set up the ones that support it. This paper discusses how to use patterns to create a workspace that resolves issues in team development by building the supporting structures.

Why a pattern language, and not simply a list of guidelines? The details of what to do in a given environment often vary depending on the situation; guidelines often hide that detail.

The patterns cover many of the issues in detail, leaving you to explore whether the pattern is a structure that you need, and if it is, how you implement it.

[1] As an example: [7] James O. Coplien, "A Generative Development Process Pattern Language," in *Pattern Languages of Program Design*. Reading, MA: Addison-Wesley, 1995..

Workspace

This paper talks about the virtual place where you as a developer spend most of your day when programming: the development workspace.

Your work on a software project has many aspects, including:

- The architecture
- The organization's structure
- Version control and change management policies and tools
- Your working style.

Bass [14] for example, discusses how organization has an impact on the module structure. These aspects all have an effect on:

- Workspace Management: How you set up your local development environment and how your workspace related to others.
- Version Control and identification: how you use source control tools and other means to coordinate changes with others, publish your changes, and reproduce environments, such as when you need to fix a bug in an earlier release. This includes issues such as branching and labeling, which are often faced with much consternation.
- Coordination: How you work together with other teams and developers.
- Identification: How you know what you built.

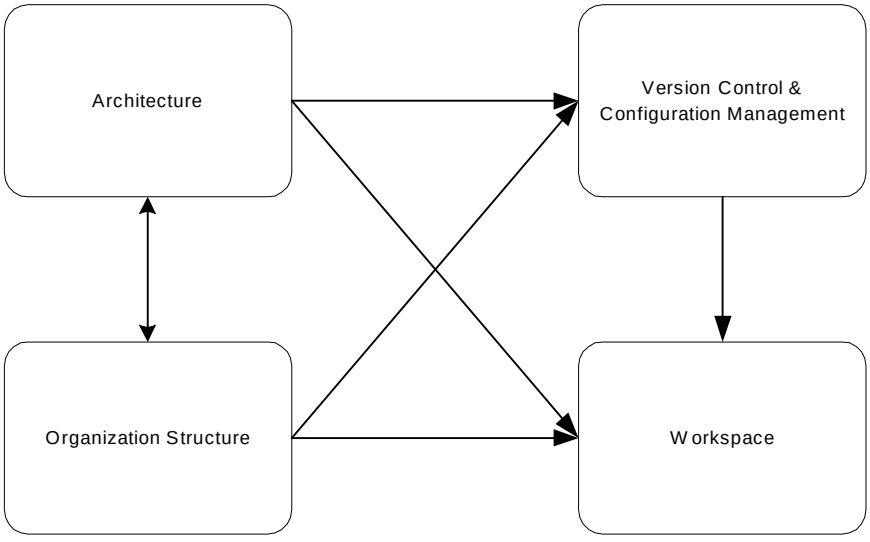


Figure 1: Influences between Structures

Why Care about Workspaces?

Even the most advanced, high-concept, distributed systems rely on low level processes working correctly. Effectively deploying a product plan or development process depends on the way you manage your workspace. The workspace is where software development happens. Consider the following scenarios.

A company has a complicated build scheme where the latest versions are always built nightly, and developers use the shared objects to do their own development. This lets them save disk space. You start working on a problem with the latest build. You spend the whole day setting up a reliable test scenario, but

then need to leave because of a personal engagement. The next morning, your test set up no longer compiles. Someone changed something in one of the user interface libraries (which do not impact the bug) and you implicitly got the latest code. You now need to spend the next day getting back to where you were last night.

In another scenario, you are making complicated changes to a body of code. It takes a couple of days, and you check in the code. When the nightly build runs your code appears to have broken the build. You track the problem down to changes by another person made before you checked in your change.

These scenarios are based on real problems in companies that thought they had a pretty good system going since they had automated builds. All could have been avoided by structuring developer's workspaces appropriately. Unless your organization puts significant roadblocks in your path, you can structure your workspace to avoid these problems almost entirely on your own.

Elements of a Workspace

A workspace is connected to the workspaces of other developers, as well as the surrounding infrastructure of the organization that the developer works in. As a developer, you can do a lot on your own to help a process go smoothly; ideally there are structures around the developer's "realm" that support make it easier. Figure 2 shows the relationship between these elements.

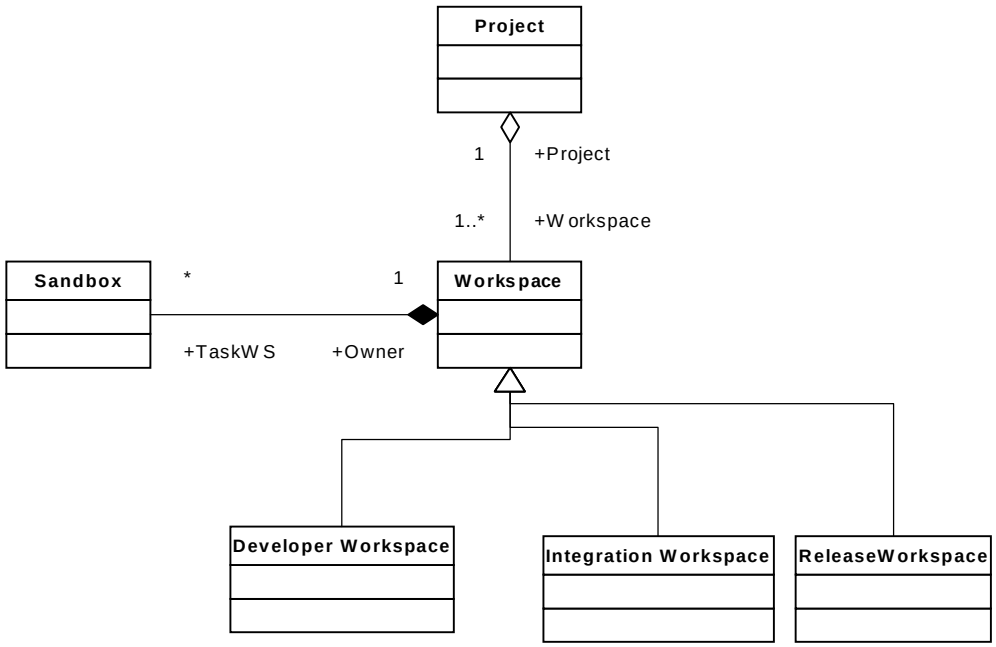


Figure 2 Elements of a Workspace

The workspace includes (but is not limited to):

- Any source files (or their equivalent).
- The procedures that you use to interact with other developer's workspaces.
- The tools used to manage development and change, including interfaces to the Version Control system.

A workspace has the following properties:

- Some initial state. You populate the workspace from source files in version control, or from a system build.

- The current state of the source code. This includes any changes you have made.
- The current state of the built (executable) program that you are working on. This means any binaries that are affected by your source changes must be re-built. For example, when using C++, a change to a header file means (at least) a need to recompile any clients of that project. If you are developing interfaces, you may discover that certain clients will not compile unless their code is changed.

While it is possible to insulate yourself from changes by appropriate use of interfaces, developing an evolving system means cooperating with other people. You need to integrate their changes, and you need to publish your changes. With the appropriate structures in place you can do this painlessly.

A Pattern Language for Workspaces

This section details the patterns. The patterns come from many sources (as cited in the references section).

Once the architecture and basic structure of the codelines and responsibilities are established, you need to consider how to build and structure the workspaces. Figure 3 shows the structure of the language. An arrow from one pattern to another means that the first pattern is completed by the pattern that it points to.

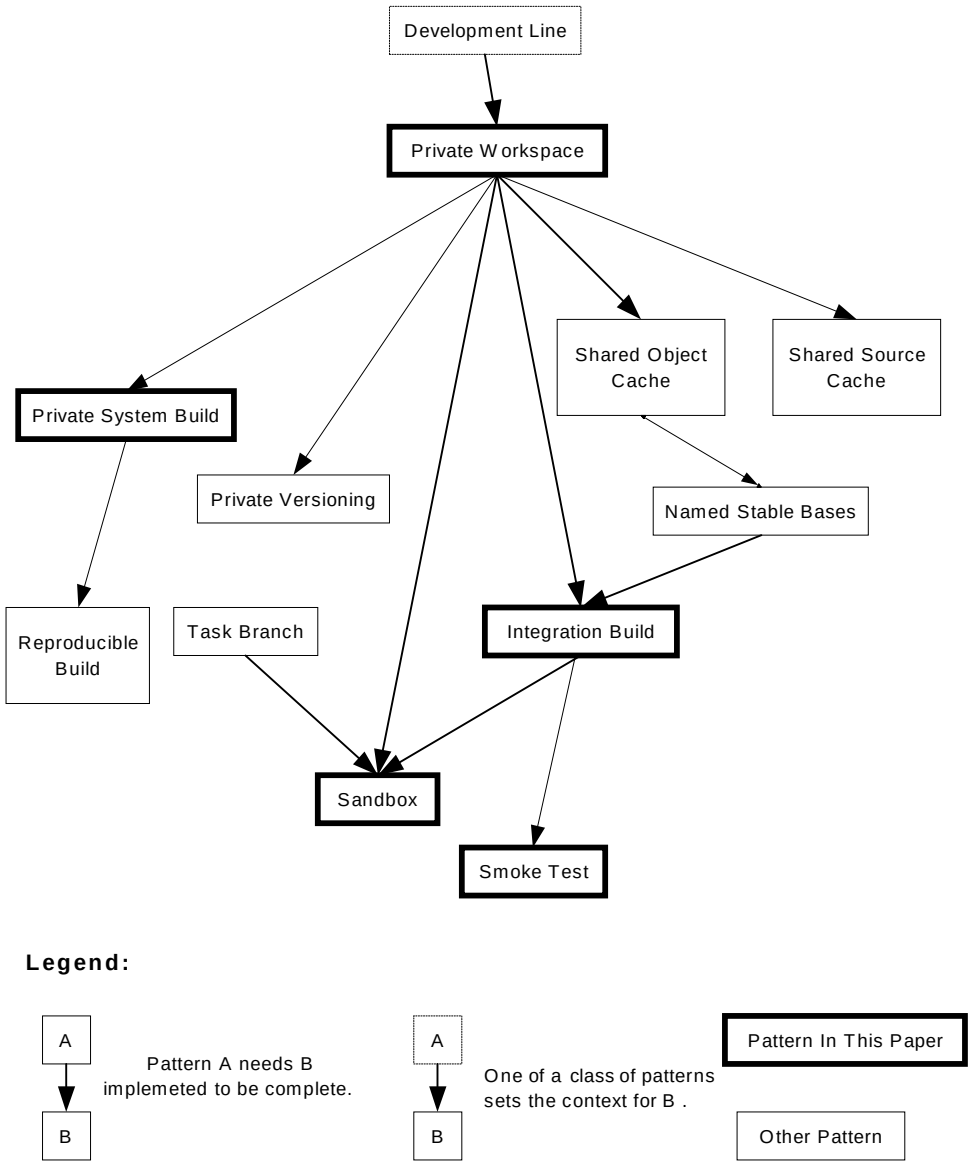


Figure 3: Overview of the Language

This looks more complicated than it is. The key to understanding the pattern language is to think in terms of structure instead of process. The language will show you how to build a workspace by adding on additional supporting, structures.

Using the Language to Control Change

Software product development involves development on many time scales:

- Releases. (There may be more than one release available to customers.)
- Builds. You many have a process for preparing complete packages in a intermediate state. These packages may be for pre-release QA testing or simply for development testing,
- Change lists. Every time you makes a change,, the state of the system changes.

Set up your workspace to help the developer deal with the rate of change. Consider the following goals:

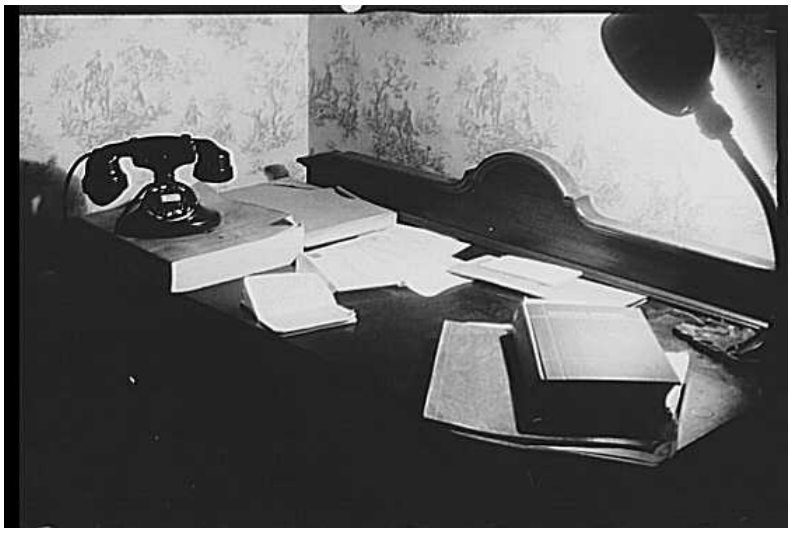
- Ensure that your changes will not break the rest of the system when they are checked in (published).
- Ensure that you know your current state of the system in your workspace. You do not want files changing out from under you.

You want an integration workspace where you can make changes, incorporate other changes, and publish the results at your own pace. An integration workspace is built out of the following other components:

- A Private Workspace. This is where the work gets done. The workspace by itself is a simple structure and it needs a number of supporting elements.
- A CodeLine (mainline) to get the source from which you will use to start work. If you are working on an older build you can get the code for that version. The CodeLine contains the SharedSourceCache.
- Since some components are quite stable, and perhaps expensive to build, you may want to start with a SharedObjectCache to populate your environment. This cache is populated by a system build. If the code if truly immutable, you can point to a shared resource, but you do not want these objects changing without your knowledge after a subsequent build.
- Any changes you make could affect many parts of the system. You want to be able to build the entire system using a procedure similar to the one that generates released objects. A PrivateSystemBuild enables you to do this.
- As you work on changes, you many want to make an experimental major change in the code, and want to checkpoint part of your changes. Checking in the code to a version control system often makes the code available to others. PrivateVersioning provides a way to handle this using Branching patterns or other mechanisms supported by your tools.

The Patterns detail the tradeoffs involved in setting up a workspace.

Private Workspace



When you are working on a problem on an active *Development Line* (for example, *Mainline*) people will be making frequent changes. But people don't work well with uncontrolled change. 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.

As a developer, you must have a way to maintain local stability in the midst of global change. This is the only way to get changes made efficiently and quickly.

When you develop software you need access to the code for the element that you are working on. The code base for this may be changing because other team members are working on related parts of the component. You also need access to the components that your element interfaces with so that you can do testing. The external components are also undergoing evolution.

One way to run a development project is to work off the latest system build all of the time, and receive automatic updates. This is the most economical in terms of space and time. You only need to change and build your components. The system gets built once, perhaps daily, and everyone references the new binaries and source. In a complex system (which most systems are) this means that many things are changing. Most of the time this won't be a problem, since the changes you are importing may not have a direct effect on what you are doing. But software systems are complex, and it is difficult to determine ahead of time whether a change will have consequences for your work.

At any point in time some of these changes may be buggy, or at least, inconsistent with the changes that you are working on. Trying to work with these latest components will slow you down.

This situation can cause problems if the shared repository changes while you are in the middle of a task. The benefits of saving disk space are evaporated by the cost of having the developer try to reproduce a test scenario, or try to get his components to build in the presence of interface changes in the newest code. A change that can have substantial effect on the part of the system you are working on can cause you hours of effort by forcing you to resolve new problems that are unrelated to your current work.

Therefore,

To develop code, you need to have a Private Workspace where you control what you are working on. You have total control over when your environment changes.

Every team member should be able to set up an environment where they have total control of what they are working with. You control when systems are updated. Ideally, you want to be as up to date as possible, but

you only want to update your workspace between tasks, and you want to guarantee that you are able to obtain particular versions of other components so that you can be sure that you can build and test.

One risk with allowing this control is that developers will work with old "known" releases too long, and they will be working with outdated code. You can protect yourself from this by doing periodic *Private System Build*s and making sure that changes do not break the build or fail the *Smoke Test*.

You can populate the environment either by copying the files from a known good build (one of the *Named Stable Bases*) or from the *Mainline* or by referencing files from the *Shared Object Cache* for components that you will not change. If disk space is at a premium, use files from a *Shared Source Cache*(RC) if you will not be changing the files. You can also get the all of the source from the correct code line and build the system from scratch if that is not too time consuming. Figure 4 shows the structure between the workspace, source control, and the starting point for your workspace.

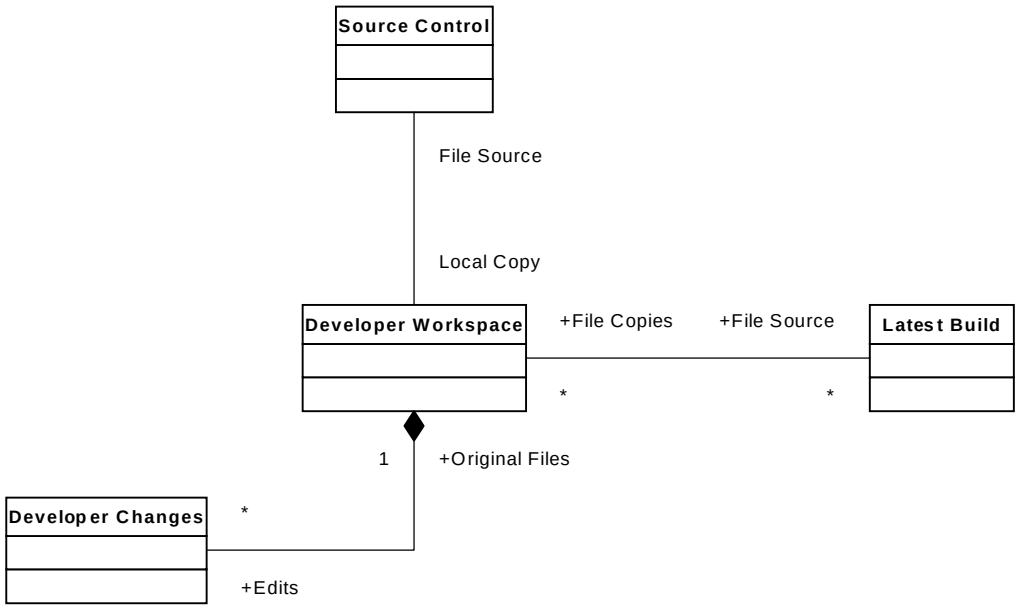


Figure 4 Structure of a Private Workspace

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 be sure that you have built all dependencies, do a *Private System Build*. Check that the changes integrate successfully with the work others have done in the meantime by getting the latest code from the *Mainline* (exclusive of changes you have made). If you are working on multiple tasks at one time, your workspace should have many *Sandboxes.*

Having a *Private Workspace* does take more space than working with shared source, but the simplicity that it adds to your work is worth it. An automated build process should also have its own workspace, but this workspace would always get all the updates, if you are doing a "latest" build.

Example

The Widget Editor team uses components from the "Shared Component Team." The Shared Component Team is in the midst of adding a complicated feature, and the last couple of build have flaws. You know that the interfaces that you are using are not changing. The Widget Editor team sets up their build environment so that they use a known good build of the Shared Components. As the quality of the tip of the source control line works, some of the Widget Editor Team members start building against the latestShared Components.

References

This is a structure that that some find so obvious, that some feel that it isn't worth documenting. Others never stumble upon it, but a *Private Workspace* is the first step in controlling your environment. Whitgift [15] briefly mentions the role of workspaces as a place "where an item evolves through many temporary and inconsistent states until is checked into the library."

Private System Build

A *Private Workspace* allows you, as a developer, to insulate yourself from external changes to your environment. But you are making changes to the global environment too. This pattern explains how to make a good effort at knowing that your code will still be consistent with the latest published code base when you submit your changes.

You need a way to ensure that the impact of your changes can be evaluated effectively before a system build.

In a development team with liberal code line policies changes happen very fast. The only true test of whether changes are truly compatible is the system build. Often organizations have very well established formal build procedures, but they don't scale down to the developers. To be able to do a reasonable test of the effect of the changes, you must be able to build all parts of the system that your code has an effect on.

Therefore,

Before making a submission to source control do a Private System Build.

Make sure that your code works in your current environment. Then update your *Sandbox* with the latest code from the Codeline that you are working on, and do a build. A complete (full) build is best, but if your dependencies are set up correctly, an incremental one is sufficient.

The private system build should have the following attributes:

- Be like the Integration and product builds as much as possible, though some details that are related to release and packaging can be omitted. It should at least use the same compiler, versions of external components, and directory structure.
- Include all dependencies.
- Include all of the components that are dependent on the change. (For example, various application executables.)

The build can differ from the product build in the following ways:

- It can be done in an IDE or other development environment, as long as you know that the compiler is compatible with the one used in the Product Build process.

- It can skip steps that insert identifying information into the final product, for example, updating version resources.

A *Private System Build* does take time, but this is time spent by only one person rather than each member of the team should there be a problem. If building the entire system is prohibitive, build the smallest number of components that your changes effect.

Example

You have just fixed a bug that took 3 days to fix. After verifying that the bug is fixed, you synchronize your workspace to the tip of the code line, and then do a Private System Build using a build procedure that builds all of the components of the system. This procedure can omit items such as incrementing release numbers, etc, that may be present in the product build. If the build works, and passes a minimal smoke test, you check your changes in.

References

Steve McConnell mentions the need to do a build before checking in code in Rapid Development [16].

Sandbox

Within a *Private Workspace*, you may need to work on different tasks at the same time. These projects may not even be compatible. While working on the tasks, you want to be able to have free reign without adversely impacting others. This pattern addresses the issues that arise when you need to work independently on a number of projects, in a number of environments.

Sometimes you need to work on more than one task simultaneously where you are free to experiment independently with the code for each one.

Software development is interrupt driven. Versions of the code are produced and some of these versions comprise the code base of the release. There are tasks for the release that is currently being developed and tasks to fix bugs in the latest release. Within each of these releases there are tasks that have different time scales. Each task may require a different version of the code base, with different supporting tools and libraries. The different versions may not work well together.

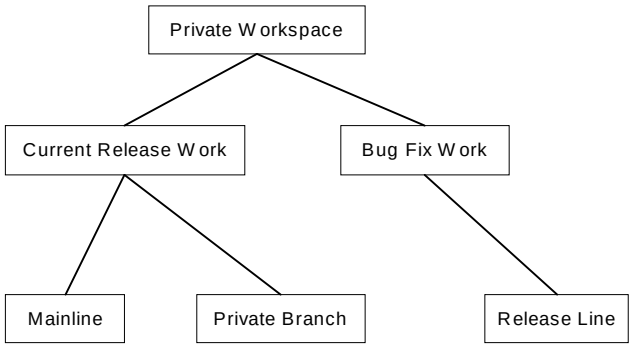


Figure 5: Tasks in a workspace

Therefore,

Allow for a number of sandboxes within each workspace. A Sandbox is an independent copy of all the code needed to perform a task. Create a sandbox for each task that works off of a different code base.

Populate the sandbox from the appropriate branch. Share or copy files and binaries as appropriate, being sure that you can re-build any objects that are dependent on objects that you change. The sandbox should also include the correct versions of third party code, as well as all local system components.

Some component environments, such as COM, define certain items on a machine wide basis, so be sure to have a process to switch between workspaces by un-registering and registering the appropriate servers.

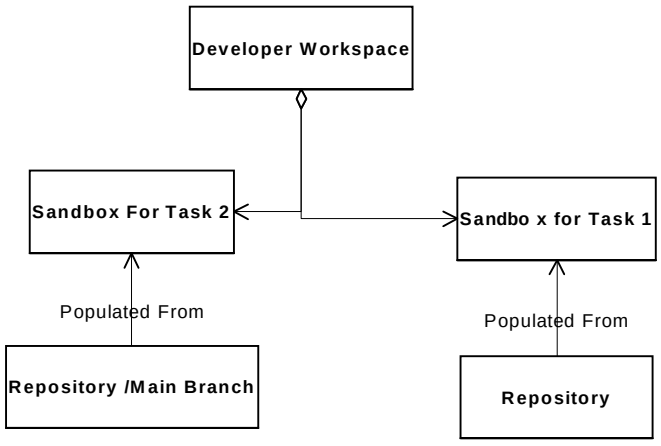


Figure 6: Relationship between Sandbox and Workspace

This structure can be used in combination with a *Task Branch* when you are performing a complicated Refactoring,[17]. Each *Task Branch* can then be associated with a sandbox.

Example

Suppose that you are working on release 3.0. This release is still on the main line. A bug for the 2.1 release, which is in the final stages of QA before release, is reported.

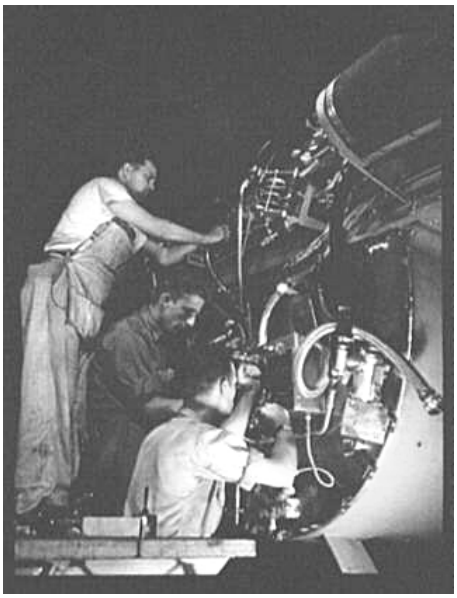
You create a new sandbox in your workspace and populate it with the binaries from the release that corresponds to the 2.1release. You then get the latest code from the 2.1 Branch, and extract the relevant source code in to your workspace. When you are confident that things work, check the changes back into the appropriate code line.

References

Many version control tools provide explicit support of separate sandboxes. Perforce[2] provides for "client specifications" that map different versions of files to different directories in the user's file system. The client specifications also have the change lists associated with them.

[2] www.perforce.com

Integration Build



When you change code, you should check to be sure that code compiles before it is checked in, but because of concurrent work done in separate *Private Workspace*s and *Sandboxes*, or simply error, the code may break the *System Build*. This pattern addresses mechanisms for helping to ensure that the code always builds.

How do you make sure that the code is always in a state such that it builds?

When you make code available for other developers (checked-in, published, etc) it is possible that, despite your best intentions, that you may introduce build errors. Your build environment may even be inconsistent with the "release" build environment at any point in time.

Doing a complete build does take time, but if the build is broken, the problem is at least localized.

Tracking down inconsistent change sets is frustrating work for other developers, so the smoother the build, the higher morale. You need a way to ensure that these inconsistencies are caught as quickly as possible.

Therefore,

Be sure that all changes (and their dependencies) are built using a central Integration Build process.

This build process should be:

- Reproducible
- As close as possible to the Product build. Minor items, such as how files are version labeled might vary, but it is best if the Integration Build is the same as the Product build. At the end of the Integration build, you should have a Release (Testing) Candidate.
- Automated, or requiring minimal intervention to work. The harder a build is to run, the more even the best-intentioned developers will skip the process occasionally. If your source control system supports triggers, you could have the build run on every check-in.
- A notification or logging mechanism to identify errors and inconsistencies. The sooner that build errors are identified, the sooner they can be fixed. Also, rapid notification makes it easier to track the change that broke the build.

Perform the build in a *Sandbox* that contains the components being integrated. Determine how often to run the integration build based on the following factors:

- How long it takes to build the system
- How quickly changes are happening

If the system takes a long time to build, or if the product is fairly static, consider at least a staged daily build, with an option to run additional builds as needed. Otherwise, consider running the build on every submission (check in) to source control. While this may seem resource intensive, it will make it very easy to determine the sequence of changes that broke the build.

Follow up the Integration Build with a *Smoke Test*.

Example

You check in a change to the repository. The source control system responds to the check in by extracting all of the files for the system, and it builds the resulting system. Errors in the build get reported to the build master as well as the person who submitted the change.

References

Rapid Development [16]describes a Daily Build and Smoke Test. The Daily Build and Smoke Test Pattern first appeared in Coplien's pattern language[7].

Smoke Test

Periodic *Integration Builds* are useful for verifying low-level integration issues. There are still runtime issues that can cause you grief later. This pattern addresses the decisions you need to make to validate a build.

How do we know that the system is still functional after the last changes?

You hope that you tested the code adequately before checking it in You hope that others have done so as well. Even if you and your colleagues make a good faith effort to test, you may still not have tested against all of the changes made by others. Also, some integration tests may need resources that are not on every development machine.

Therefore,

Subject each build to a smoke test that verifies that the application has not broken in an obvious way.

The scope of the test need not be exhaustive. It should test basic functions, and simple integration issues. Ideally it should be automated so that there is little cost to do it. The *Smoke Test* should not replace deeper integration testing. A suite of unit like tests can form the basis for the smoke test if nothing else is immediately available. Most importantly, these tests should be self scoring. They should return a test status and not require manual intervention to see if the test passed. (An error may well involve some effort to discover the source.)

Running a Smoke test with each build does not remove the responsibility for a developer to test his changes before submitting them to the repository. A smoke test is most useful for bug fixes, and for looking for inadvertent interactions between existing and new functionality. All code should be unit tested by the developer, and where reasonable, run through some scenarios in a system environment.

When adding new functionality to a system, extend the smoke test to test this functionality as well.

Example

The automated build runs a script that tests basic functionality.

References

"Daily Build" and "Smoke Test" often appear in the same sentence, so the references for Integration Build may be interesting. *Code Complete* [18] describes strategies for developing Unit tests. *The Art of Software Testing* [19]provides and excellent overview of basic testing strategy.

Related Patterns

The following table lists some patterns that were mentioned in the document that are described in detail elsewhere.

Pattern Name	Description	Referenced in
Task Branch	A short-lived branch to perform a specific task. This allows you to checkpoint changes before they are ready to be shared	http://www.enteract.com/~bradapp/acme/branching/ *Streamed Lines: Branching Patterns for Parallel Software Development*
Mainline	Structure your version control system so that current work is done on a Mainline, with releases and other work branching off of it.	http://www.enteract.com/~bradapp/acme/branching/ *Streamed Lines: Branching Patterns for Parallel Software Development*
Shared Object Cache	Create a place where developers can copy or reference the results of a good build.	http://www.enteract.com/~bradapp/acme/plop99/ *Software Reconstruction: Patterns for Reproducing Software Builds*
Shared Source Cache	Create a place where developers can reference the source for components that they will not change.	http://www.enteract.com/~bradapp/acme/plop99/ *Software Reconstruction: Patterns for Reproducing Software Builds*
Named Stable Bases	Identify points in time for the software source tree for which the software works to an extent adequate for integration	

Acknowledgements

Thanks to Linda Rising, our PLoP 2K shepherd for this paper for her valuable attention to detail, and in particular for her suggestion of using a second person point of view throughout the paper. Also, discussions with Jim Coplien about the geometry of patterns got me thinking about the structure of this paper, a pursuit that helped solidify my thinking.

My PLoP 2K workshop group, especially Paul Asman, Dick Gabriel Andreas Rueping, and Phil Eskelin, made many insightful comments. One cannot overstate the role of the workshop process in improving the patterns literature.

References

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[2] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal, *Pattern-Oriented Software Architecture: A System of Patterns*. Chichester, England: John Wiley & Sons, 1996.

[3] James O. Coplien and Douglas Schmidt, "Pattern Languages of Program Design,". Reading, MA: Addison Wesley, 1995.

[4] John Vlissides, James Coplien, and Norm Kerth, "Pattern Languages of Program Design 2,". Reading, MA: Addison-Wesley, 1996.

[5] Neil Harrison, Brian Foote, and Hans Rohnert, "Pattern Languages of Program Design 4," in *Software Patterns Series*, John M. Vlissides, Ed. Reading, MA: Addison Wesley Longman, 2000.

[6] Robert C. Martin, Dirk Riehle, and Frank Buschmann, *Pattern Languages of Program Design 3*. Reading, Mass.: Addison-Wesley, 1998.

[7] James O. Coplien, "A Generative Development Process Pattern Language," in *Pattern Languages of Program Design*. Reading, MA: Addison-Wesley, 1995.

[8] Stephen P Berczuk, "Organizational Multiplexing: Patterns for Processing Satellite Telemetry with Distributed Teams," in *Pattern Languages of Program Design*, vol. 2, John Vlissides, James Coplien, and Norm Kerth, Eds. Reading, MA: Addison-Wesley, 1996.

[9] Steve Berczuk, "Configuration Management Patterns," presented at Third Annual Conference on Pattern Languages of Programs, Monticello, IL, 1996.

[10] Brad Appleton, Steve Berczuk, Ralph Cabrera, and Robert Orenstein, "Streamed Lines: Branching Patterns for Parallel Software Development," presented at Fifth Annual Conference on Pattern Languages of Programs, Monticello, IL, 1998.

[11] Ralph Cabrera, Brad Appleton, and Steve Berczuk, "Software Reconstruction: Patterns for Reproducing the Build," in *Proccedings of the Sixth Annual Conference on Pattern Languages of Program Design*. Monticello, IL, 1999.

[12] Stephen P. Berczuk, "Teamwork and Configuration Management," *C++ Report*, vol. 9, pp. 28 ff, 1997.

[13] C. Alexander, *A Timeless Way of Building*: Oxford University Press, 1979.

[14] Len Bass, Paul Clements, and Rick Kazman, *Software Architecture in Practice*. Reading, Mass.: Addison-Wesley, 1998.

[15] David Whitgift, *Methods and Tools for Software Configuration Management*. Chicester, England: Wiley, 1991.

[16] Steve McConnell, *Rapid Development, Taming Wild Software Schedules*. Redmond, WA: Microsoft Press, 1996.

[17] Martin Fowler, *Refactoring: Improving the Design of Existing Code*. Reading, MA: Addison-Wesley, 1999.

[18] Steve McConnell, *Code Complete : A Practical Handbook of Software Construction*. Redmond, Wash.: Microsoft Press, 1993.

[19] Glenford J. Myers, *The Art of Software Testing*. New York: Wiley, 1979.

Photo credits

Private Workspace: Library of Congress, Prints & Photographs Division, FSA-OWI Collection, LC-USF33-015598-M2 DLC

Integration Build: Library of Congress, Prints & Photographs Division, FSA-OWI Collection, LC-USW361-138 DLC 6

Sandbox: Library of Congress, Prints & Photographs Division, FSA-OWI Collection,LC-USF34-024142-D DLC

Smoke Test: Library of Congress, Prints & Photographs Division, FSA-OWI Collection LC-USF33-011632-M1 DLC

Private System Build: Library of Congress, Prints & Photographs Division, FSA-OWI Collection, LC-USF33-006349-M3 DLC

