Configuration Management Patterns

Steve Berczuk
Optimax Systems Corporation
201 Broadway
Cambridge MA 02139
berczuk@optimax.com

Configuration management is an important aspect of an efficient development process. As with other aspects of software development, configuration management happens in the presence of social and organizational constraints. The various goals of configuration management (stability, developer independence and interaction, etc.) sometimes conflict with each other. This paper presents three configuration management patterns and shows how the forces between them balance out. It also relates them to organizational patterns

Introduction

Software configuration management (SCM) is an important aspect of development process. A good SCM process makes it possible for teams of developers to work together on a project effectively. While there are various tools that can make the process simpler, there are certain patterns for software configuration management[1] that exist unsuccessful development organizations. (A detailed discussion of Configuration Management can be found in [Humphrey90] among other sources.)

A successful configuration management process allows:

- Developers to work together on a project, sharing common code.
- Developers to share development effort on a body of code.
- Developers to have access to the current stable (tested) version of a system.
- The ability to back up to a previous stable version (one of a number of *Named Stable Bases* [Coplien95]), of a *system*
- The ability of a developer to checkpoint changes to a module and to back off to a previous version of that *module*.

[1]These patterns focus on source code configuration management. A complete SCM system should also take testing, tools, and other elements of the software process into account.

Attaining all of these goals involves compromises (many people working of one body of code makes maintaining consistency difficult, for example). This paper presents 3 patterns that illustrate how the problems presented by configuration management problems can conflict with each other, and how the various forces can be balanced. We also show how SCM interacts with some of the organizational process patterns described by Coplien [Coplien95].

The goals of this paper are to:

- Illustrate how patterns can be applied to the mechanics of software development.
- Show how configuration management patterns can interact with organizational patterns.
- Provide an illustration of how the forces in some patterns interact, and how one can try to balance conflicting forces.

Private Versioning[2]

A developer should have a way to checkpoint changes without making these changes available to the development team at large. We want to implement *Code Ownership*(17) [Coplien95] but subsystems never work entirely in isolation.

Problem:

Periodic integration of a developer's work with that of other members of the development team is important for ensuring stability. Checkpointing only after completing major changes can make it difficult to back off of one phase of a change. Using the revision control area for this can lead to changes being "published" before they are ready for integration. Also, publishing intermediate changes can lead to a deceptive number of revisions listed in the SCM system. It is necessary to be able to save intermediate steps in a change in case a coding step results in an error. This is particularly important when:

- The mechanism for specifying that a version is ready for integration is primitive, and another developer has access to a version as soon as it is checked it.
- There is a desire to keep the revision history database "uncluttered" with only significant changes logged.

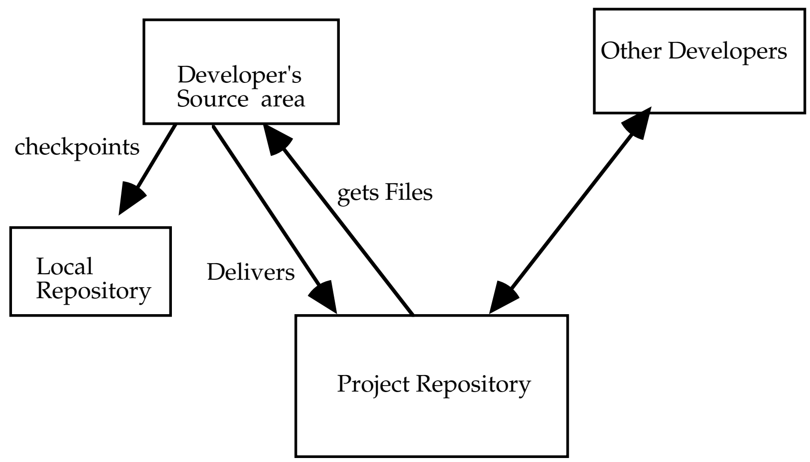
therefore:

Developers should be provided with a mechanism for check pointing changes at a granularity that they are comfortable with. This can be provided for by a

---

[2]This pattern was inspired by work done by Doug Alan at the MIT Center for Space Research.

local revision control area, Only stable code sets are checked into the team version control repository.



It is important to make sure that developers using *Private Versioning* remember to migrate changes to the shared version control system at reasonable intervals.

This pattern provides a way to automate the checkpointing of changes, which could alternatively be done by making copies of files before making changes, thus providing the local history mechanism. The revision control mechanism could also provide a means for restricting access to checked-in versions that are not yet ready for use by others, and could also provide a mechanism for filtering log messages to eliminate trivial changes.

Incremental Integration

Some organizations have infrequent integrations which reflect large changes. This can make it difficult for the integration release to work as expected, and result in *Named Stable Bases.* [Coplien95]

Problem:

For iterative development to work well, it is necessary to make sure that components work together. Subsystems get developed at different rates. Developers work with *Independent Development*. We need to find a way to make it possibly to integrate without surprises.

therefore:

Provide a mechanism to allow developers to build all the current software periodically. Developers should be discouraged from maintaining long intervals between "check-ins." Developers should also be able to build *against any of the* Named Stable Bases, *or the newest checked in software, at will.*

Assign the task of building the entire software system periodically. *Named Stable Bases* suggests intervals no more frequent that a week. This periodic build should be checked for interface compatibility (does it compile?) and testing (Does it still work?) This can be combined with *Independent Development* when a developer ensures that the changes integrate with a copy of the current development system. There are issues relating to the size of the project, the size of the software system (some systems take quite a while to build, making frequent integrations difficult). Balance this with *Private Versions* to allow the developer some leeway on deciding when to integrate new code into their environment, but do not put it off for too long.

Independent Development

It has been decided to implement *Named Stable Bases* (31)[Coplien95] . However, we must balance the need to keep up to date by *Incremental Integration*, with the desire of developers to maintain a stable environment for feature development/bug fixing.

Problem:

How can we balance the need for developers to use current revisions, based on periodic baselines, with the desire to avoid undue grief by having development dependencies change from underneath them?

It is important for developers to work with current versions of software subsystems to keep up with the latest enhancements, avoid running into already fixed bugs fixed elsewhere, and to avoid getting out of synch with interface changes. A developer who keeps changes un-released (or not checked in to the version control system) can cause surprises to other team members. *Named Stable Bases* recommends integrations at an interval of no more than once a week. Introducing new software into an environment while debugging may cause grief by introducing new behavior, and providing distractions because of the time spent resolving integration issues — in some cases, code may no longer compile due to interface changes.

Some organizations, to facilitate *Incremental Integration*, will have a shared baseline of code, libraries, etc. Unfortunately changing a code base, even in a different subsystem, can cause problems when there are interface changes, for example. You want to avoid hearing stories about developers leaving a problem at night to view it in the morning with a clear head, only to find that one's test environment does not compile.

therefore:

Provide a mechanism where developers can maintain independent development environments, This allows them to avoid having an *integration step interrupt work in progress. The environment should represent a snapshot of all the software being developed in a system, not just the code the developer is modifying. Try to ensure that the private development area is not used as a crutch though.*

This pattern conflicts somewhat with *Incremental Integration* as in the case where a developer delays retrieving the current release for too long.

A consequence of this pattern is that, depending on how this is implemented, the disk space requirements of a project may grow quickly as N developers will have their own copies of the source code. But often the costs of personnel greatly exceed the cost of an extra disk. A modification to this approach is that stable, and distantly related subsystems can be used by reference, but one should be made aware of when changes are imminent. In this case the CM system should provide access to prior *Named Stable Bases* as well.

A variation on this pattern is to allow developers simply to defer advancing to a new *Named Stable Base* until the current problem is solved.

Conclusions

Patterns can be used to describe aspects of the development process. This paper shows a model for how configuration management patterns can interact with development process patterns. Even with good tool support, an organization should have a consistent model for software configuration management.

Acknowledgments

Software configuration management is a process that occurs at all development organizations, with varying degrees of success. Javier Barreiro and Neil Harrison for providing me with other perspectives on configuration management. Royce Buehler and Doug Alan worked with me on many of the issues for the configuration management system at the MIT Center for Space Research, on which some of these patterns are based. David Ting first brought SCM issues (particularly those involving remote teams) to my attention while I was at the Kodak Boston Technology Center. My PLoP shepherds, Russ Corfman and DeLoy Bitner have made many useful suggestions.

The development of the SCM system at MIT, which inspired these patterns, was supported in part by NASA/GSFC contract NAS5-30612.

References

[Coplien+95] Coplien, Jim and Schmidt, Doug editors. *Pattern Languages of Program Design*. Reading, MA: Addison Wesley, 1995.

[Coplien95] Coplien, Jim. *A Generative Development Process Pattern Language*. in [Coplien+95]

[Humphrey90] Humphrey, Watts. *Managing the Software Process*. Reading, MA: Addison-Wesley, 1990.