Teamwork and Configuration Management: Some Patterns[1]

Steve Berczuk
berczuk@acm.org

Coordination and communication are two important parts of a successful development process. Social and organizational constraints have an effect on the development process. By thinking of software configuration management(SCM) in the context of how the SCM process helps team members work together we can apply SCM more effectively. These issues are important regardless of language used, but these patterns come out of my experience is in doing C++ development. This paper presents three configuration management patterns and shows how they relate to organizational patterns.

Introduction

Software development involves social process as well as a technical issues. As teams get larger, communication becomes harder[2]. To address these issues developers use processes and tools to help the team share a common vision of the system it is developing. Designing with team communication issues in mind can make the system easier to build[2] [3]. Considering these interactions in the implementation of the development process can also be useful, particularly when using a language such as C++, where it is easy to have many compile time dependencies between modules different people are developing.

Much of my development experience has been working with teams of people who work in different locations. In these cases effective configuration management was essential for cooperative development. Even when your project involves people in the same location though, it is easy to have ineffective configuration management cause problems which generate extra work.

Following patterns such as these can simplify the development of a number of systems in C++ where the common code base was changing rapidly, and we needed to stay in synchronization, while at the same time protecting each developer from changes occuring out from under them. Projects I have worked on which had these issues include a satellite telemetry processing system, an document imaging system, and an scheduling system. At some level though, any project being done with a demanding set of customers and a tight schedule will have similar issues.

Successful development organizations exhibit certain patterns which address the coordination and communication issues. Coplien catalogs some of the steps in the

---

[1] This is a pre-copyedit version of the paper that appeared in the July/Aug 97 C++ Report (Vol 9 #7 pp28 ff).

[2] One rule of thumb is that "communication overhead becomes serious as there are too many people on the project to fit at one lunch table at the same time."[1]

process[4], Cunningham[5] describes patterns which comprise the Development Episodes which occur in a software development organization, and Cockburn[6] describes patterns around how social issues affect the process. Software configuration management (SCM) can be used to help realize these patterns.

Software Configuration Management

There are many aspects of configuration management; this paper discusses some basic SCM patterns and how they can be used to implement the broader organizational patterns relating to coordination among developers. Grinter[7] describes some of the coordination challenges of software development process in detail.

A standard definition of software configuration management includes the following aspects[3]:

- Configuration Identification, which includes determining which body of source code you are working with. This makes it possible to know, among other things, that you are fixing a bug in the source code which is in the correct release.
- Configuration Control, controlling the release of a product and changes to it throughout the lifecycle to ensure consistent(re) creation of a baseline software product. This can include not only changes to source files, but also which compiler and other tools were used so issues such as differences between compiler support for language features can be taken into account.
- Status Accounting Audit, recording and reporting the status of components and change requests, and gathering vital statistics about components in the product. One question we may want to answer is:, "How manyfiles were affected by fixing this one bug?"and
- Review. validating the completeness of a product and maintaining consistency among the components by ensuring that components are in an appropriate state throughout the entire projectlife cycle and that the product is a well-defined collection ofcomponents.

It can also be said to include[4]

- Build Management, ie, managing what processes and tools developers use to create a release, so it can be repeated.
- Process Management,ensuring that the organization's development processes are followed by those developing and releasing the software
- Team Work, controlling the interactions of all the developers working together on a product so that people's changes get inserted into the system in a timely fashion.

---

[3] Thanks to Brad Appleton for summarizing these points quite succinctly.

[4] Private Communication with Brad Appleton.

Of these elements, the latter 3 play the largest role in the day to day workings of the development process, since they affect what a developer does day to day.

Ideally a configuration management process should serve both broad organizational interests as well as making the work of a C++ developer easier. A good SCM process makes it possible *for developers to work together on a project effectively*, both as individuals and as members of a team. While there are various tools that can make the process simpler, tools alone are not enough; The re are also certain patterns for software configuration management[5] that exist in successful development organizations. [6]

With respect to team interactions, a successful configuration management process allows:

- Developers to work together on a project, sharing common code, for example a developer of a Derived class needs to stay in synch with whoever is developing a base class, and a client of a class needs to be able to work with the current version of that class.
- Developers to share development effort on a module, such a class or simply a single source file. This can be by design or to allow someone to fix a bug in "another person's" module if the other person in unavailable.
- Developers to have access to the current stable (tested) version of a system, so you can check if your code will work when someone else tries to integrate it into the current code set.
- The ability to back up to a previous stable version (one of a number of *Named Stable Bases* [4]), of a *system*. This is important to allow a developer to test their code against the prior consistent versions of the system to track down problems.
- The ability of a developer to checkpoint changes to a module and to back off to a previous version of that *module*. This facility makes it safer to experiment with a major change to a module that is basically working.

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 pattern s that illustrate how configuration management can help us realize larger organizational patterns.

The patterns are illustrated with an example which has the following components:

- A number of *Developer Workspaces*, each representing the environment of a specific developer in the organization.
- The *Project Repository*, which is the common set of files which are used in a development project, for example, the Version Control system.

---

[5]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.

[6]A detailed discussion of Configuration Management can be found in [8] among other sources.

Configuration management is best implemented with tools that simplify and/or enforce policies, so reference is made to tools in the examples. The patterns are independent of the tools used.

An Overview of the Patterns

This section summarizes[7] the intents and solutions of the patterns describe here and those we reference.

Configuration Management Patterns Presented Here

- *Private Versioning*: Allow a developer to checkpoint changes for convenience and security without violating organizational standards regarding granularity of revisions.
- *Incremental Integration*: Provide a means for a developer to "pre-test" interfaces before an integration build.
- *Independent Workspace*: Make sure that new builds do not interfere with a developer's work in progress

Figure 1 shows how these Patterns relate to other organization patterns, with these patterns represented by shadowed boxes. These patterns are presented in Alexandrian [9] form which is a loosely structured form in which the following elements of a pattern[8] are presented:

- *The Context:* What are the issues with which we are working.
- *The Problem:* The specific issue we want to address, including the *forces* we may want to resolve.
- *The Solution:* How to resolve the problem in our specific situation.

.Associated Patterns

Since these patterns connect to other patterns, this section briefly describes the patterns. In addition to the printed sources cited, these patterns are also described at the Organization patterns web site (`http://www.bell-labs.com/cgi-user/OrgPatterns/OrgPatterns`).

Patterns from *A Generative Development Process Pattern Language* [4] (Shown in figure 1 with a solid line).

- Code Ownership: assign a single developer responsibility for a module.

---

[7] These patterns provide more information than can be summarized in one sentence, so these summaries leave out many details, and provided here are only for convenience. In particular the context in which the pattern applies is an essential part of the pattern. Please consult the original sources for this detail.

[8] For a general introduction to pattern concepts see http://www.enteract.com/~bradapp/docs/patterns-intro.html and [10].

- Named Stable Bases, provides guidance for how often to integrate, specifying that the integrated versions are available as working "releases" for other developers.

Patterns from *Episodes* [5] (Shown in figure 1 with rounded corner boxes).

- Programming Episode: divide a program into discrete episodes with deliverables.
- Work Integration: Assemble recent work products.
- Developmental Build: build the system, and do regression tests (to permit Named Stable Bases)

Patterns from *The Interaction of Social Issues and Software Architecture* [6] show in figure 1 with a bold solid line).

- Owner Per Deliverable: Make sure that each deliverable entity has a person responsible so that the deliverable does not get overlooked.

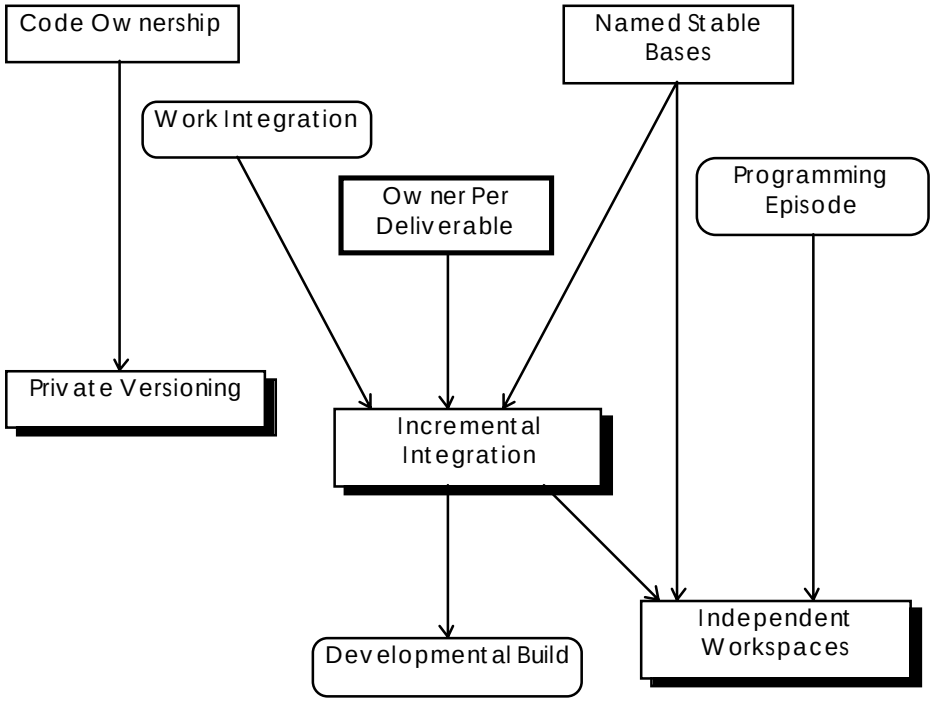

Figure 1: Relationships Between the Patterns

The Patterns

Private Versioning[9]

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) [4] but subsystems never work entirely in isolation.

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, and we want to insulate other developers from certain checkpoints. 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.

---

[9]This pattern was pointed out to me by Doug Alan at the MIT Center for Space Research.

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 local revision control area, Only stable code sets are checked into the Project Repository

Add a Private Repository to the Developer's Workspace so that a developer can save intermediate versions before checking them in to the Repository. The Private Repository can use the same mechanisms as the Project Repository (i.e., RCS) or can simply be a means of maintaining copies of intermediate files.

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

While one way to implement this is to provide a separate source control repository for each developer, in addition to the shared repository, this can also be implemented within the framework of the existing revision control system. If the revision control mechanism provides a means for restricting access to checked-in versions that are not yet ready for use by others, we can use the common version control system as a virtual Private Repository.

The important principle is to allow the developer to be allowed to use the Revision Control System to checkpoint changes in an granularity which meet their needs, without any risk of the changes (which may be inconsistent) being available to anyone else.

Example

The project repository using CVS[10] for version control. Add to the Developer workspace a shadow directory in which there is an RCS[11] directory for the developer's use only. The developer copies files to the "main" development area before checking them into the project repository.

Incremental Integration

Some organizations have infrequent integrations which reflect large changes. This can make it difficult for the integration release to work as expected, complicate the process of *Work Integration*[5] and make *Named Stable Bases*. [4] difficult to achieve when modules do not work together. Because we often develop with one *Owner Per Deliverable*[6] there will be occasional mismatches between units of work.

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

---

[10] CVS is a version control system based on RCS. It uses a merging model for changes, allowing multiple developers to work on a file at the same time.RCS is a file based revision control system which uses locking, allowing only 1 developer to work on a file at a time.

[11] For Information on RCS see: http://www.cs.purdue.edu/homes/hammer/rcs.html.

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.*

In addition to assigning 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?)), also encourage developers to build from files that are likely to be in the release (perhaps the newest code in the revision control system's trunk) to anticipate, and allow time to correct for, incompatibilities. The goal is to avoid a "Big-Bang" integration and allow the *Developmental Build*[5] to proceed smoothly.

This can be combined with *Independent Workspaces* to ensure that the changes integrate with a copy of the current development system. There are issues relating to the size of the software system (some systems take quite a while to build, making frequent integrations difficult). Balance this with *Private Versioning* to allow the developer some leeway on deciding when to integrate their new code into their environment, but do not put it off for too long.

Example

The Developer's Workspace could be updated (at the developer's request) to a named stable base from the Project Repository approximately weekly. Many tools use *labels* to indicate which files belong to s specific source code set. The developer will also retrieve the current files from the repository to anticipate how the current changes in the Workspace will work with files that may later be in the baseline.

Known Uses

Steve McConnell in *Rapid Development* [11]describes the importance of allowing developer to perform a "private build of the system on a personal machine, which the developer then tests individually," in the "Using the Daily Build and Smoke Test "best practice.

Independent Workspaces

It has been decided to implement *Named Stable Bases* (31)[4] 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, enabling a *Programming Episode* [5]to proceed smoothly.

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 disrupt 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 Independent Workspaces *where developers can maintain control off their 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 means of avoiding integration issues.*

This pattern conflicts somewhat with *Incremental Integration* when a developer delays retrieving the current release for too long, so make sure that developers are encouraged to use integrate their code frequently, perhaps by providing a mechanism for easily backing of a difficult change.

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.

Example

A developer is working on a problem The Developer Workspace is self contained with all of the files needed to build the system. Developers retrieve new files from the repository only when they are ready and the current problem is solved.

Known Uses

Clearcase by Pure Atria[12] provides the concept of Views to give us this facility. The SCM tool CA/Endevor[13] has the concept of Private Stages which allow for this. Private Stages

[12] http://www.pureatria.com/

[13] Computer Associates Web Page is http://www.cai.com/.

are not available to anyone except the owner. When a private stage is "ready" the developer promotes the changes in that stage to a public stage.

The "Using the Daily Build and Smoke Test" best practice in *Rapid Development* says that "developers should maintain private versions of the source files they're working on[11]."

Conclusions

This paper did not present an exhaustive overview of either configuration management or social interactions, but rather a framework into which more patterns can be included. Isolated patterns can be quite useful, but connecting patterns so that they form pattern languages can better demonstrate the power of patterns.

These patterns describe processes which can be using almost any SCM tool, though some make it easier than others. *Independent Workspaces* is a key part of most all SCM tools, since you "check out" files into a local area for editing. *Incremental Integration* is simply a matter of and tool configuration. *Private Versioning* can be implemented without a tool at all by simply copying files, but some tools support the concept of checkpointing a development stage directly. Rather than being a luxury, configuration managment is an essential tool for developing software quickly, and thinking about how your team works when applying SCM techniques will make it even more effective.

For find out more about patterns in general, visit the patterns web site `: http://st-www.cs.uiuc.edu/users/patterns/patterns.html`. There are more patterns about organization and social process on the web `at http://www.bell-labs.com/cgi-user/OrgPatterns/OrgPatterns`. There is also a mailing list dedicated to organizational patterns. For information on joining this, and other pattern lists see: `http://st-www.cs.uiuc.edu/users/patterns/Lists.html`.

Acknowledgments

Software configuration management processes are slightly different at all development organizations. These patterns were improved by input from people with different experiences. Brad Appleton, Javier Barreiro and Neil Harrison for provided 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.

References

[1] A. Koenig and B. Moo, *Ruminations on C++*. Reading MA: Addison-Wesley, 1997.

[2] S. P. Berczuk, “Organizational Multiplexing: Patterns for Processing Satellite Telemetry with Distributed Teams,” in *Pattern Languages of Program Design*, vol. 2, J. Vlissides, J. Coplien, and N. Kerth, Eds. Reading, MA: Addison-Wesley, 1996.

[3] S. P. Berczuk, “A Pattern for Separating Assembly and Processing,” in *Pattern Languages of Program Design*, vol. 1, J. Coplien and D. Schmidt, Eds. Reading, MA: Addison-Wesley, 1995.

[4] J. O. Coplien, “A Generative Development Process Pattern Language,” in *Pattern Languages of Program Design*. Reading, MA: Addison-Wesley, 1995.

[5] W. Cunningham, “Episodes: A Pattern Language of Competitive Development,” in *Pattern Languages of Program Design 2*. Reading, MA: Addison-Wesley, 1996.

[6] A. Cockburn, “The Interaction of Social Issues and Software Architecture,” *Communications of the ACM*, vol. 39, pp. 40-46, 1996.

[7] R. E. Grinter, “Understanding Dependencies: A Study of the Coordination Challenges in Software Development.,” . Irvine, CA: University of California, 1996.

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

[9] C. Alexander, S. Ishikawa, and M. Silverstein, *A Pattern Language*: Oxford University Press, 1977 .

[10] S. Berczuk, “Finding Solutions Through Pattern Languages,” *IEEE Computer*, vol. 27, pp. 75-76, 1994.

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