Software Reconstruction: Patterns for Reproducing Software Builds

Ralph Cabrera *cabrerar@agcs.com* AG Communication Systems

Brad Appleton *bradapp@computer.org*

Stephen P. Berczuk *berczuk@acm.org* NetSuite Development

Abstract: Software systems, as abstract, non-tangible entities, should be able to be constructed faster and more efficiently compared to tangible objects. However, software systems are actually more difficult to assemble. Tangible objects have discrete components and are usually independent of their environment. Components contributing to the build of a software system are not limited to source code; moreover, they are not always obvious. Software that has been built often cannot be reconstructed later. The software configuration management patterns described here examine and resolve some of the forces regarding the reconstruction of software systems.

Keywords: *Version Control, Patterns, Software Configuration Management, Reproducibility, Repository, Build.*

Introduction

The implementation of Software Configuration Management (SCM) processes, practices, and tools significantly affects the quality and timeliness in which a software product is developed. A component of the quality and timeliness achieved is due to the affect that SCM has on the configuration and production of the software. SCM provides the capability to identify the de-composition (configuration) and re-composition (production) of the software system structure. This paper presents some re-composition patterns from a pattern language for SCM that we began developing at ChiliPLoP ’98.

The following excerpt is from *Streamed Lines*: Branching Patterns for Parallel Software Development [Appleton et al] and lays the foundation for pattern languages in the SCM domain.

Motivation for an SCM Pattern Language

There are many approaches to SCM, and the structures, policies, and processes work best when applied in the appropriate context. This context is determined by organizational and architectural decisions, as well as previously existing SCM policies. Our goal is to place SCM structures in the context of these other existing structures, making it easier to decide how to structure an SCM process which will be effective for your context.

These SCM structures may be described as "*patterns*": named nuggets of insight conveying battle-proven solutions to recurring problems, each of which balances a set of competing concerns (see [Appleton97]). SCM Patterns fit into a framework of *Organizational Patterns*, which can be grouped as follows:

Organizational Patterns:

> Patterns which define how the organization is structured. This includes patterns that describe the size of the team, management style, etc. (see [Beedle97] and [OrgPats]).

Architectural Patterns:

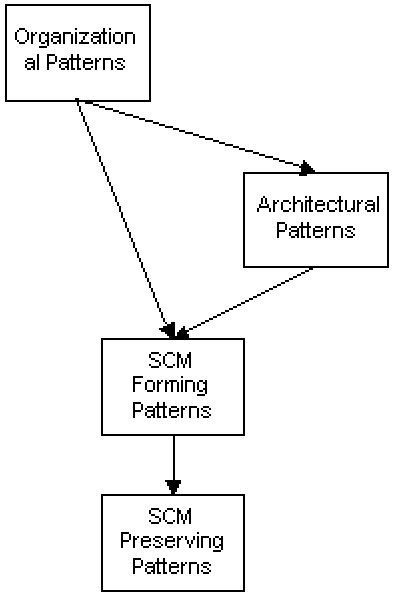
> Patterns which define how the software is structured at a high level. Some examples of these sorts of patterns have been published in prior PLoP Proceedings ([Berczuk95] and [Berczuk96]) and in books such as [POSA].

Process Defining (forming) Patterns:

> These SCM Patterns describe structures, such as the project directory hierarchy, which are set up near the beginning of a project.

Maintaining (preserving) Patterns:

> These are SCM patterns that affect the day to day workings of the organization.



These categories of patterns are shown in the figure at right.

The line between the Forming and Maintaining patterns may be blurry, but we feel the distinction is conceptually important to understand the architecture of the development process pattern language. Because of the strong relationship between the patterns in each category (how you set up the directory tree affects the process you follow for checking files in and out) we shouldn't spend too much time looking at where a pattern fits, but rather focus on which patterns it follows from.

The patterns presented by this paper should be applied with an understanding of the context in which the problem exists. The general context of the patterns is that of composition, or in some cases re-composition, of the software system from various components that contribute to the end product. These include, but are not limited to, source code components as well as environment components. A portion of these patterns belongs to the SCM Process Defining group because they establish how a project would behave (defining process/policy). A portion of these patterns belongs to the SCM Maintaining group because they describe how project would maintain the process/policy.

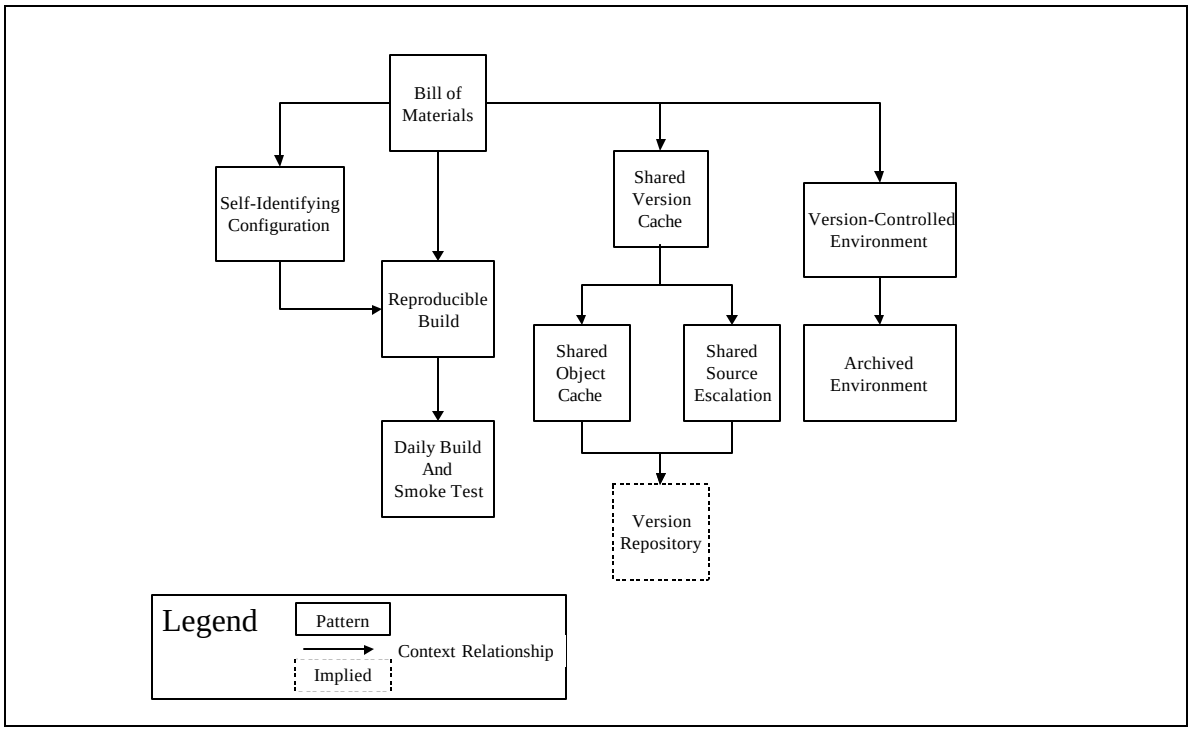


Figure 1 - Software Reconstruction Pattern Relationships

Forces

Common Forces among the patterns

- Faithful and exact reproducibility of software builds
- Isolation of desired changes
- Quality of software product
- Reduce or eliminate regression errors

Patterns

Patterns related to software reconstruction:

- *Bill of Materials*
- *Self-Identifying Configuration*
- *Reproducible Build*
- *Daily Build and Smoke Test*
- *Shared Version Cache*
- *Shared Object Cache*
- *Shared Source Escalation*
- *Version-Controlled Environment*
- *Archived Environment*

Bill of Materials

Context

You can successfully build the software system today, but you also need to build that version of the system in the future.

Problem

How can you reproduce the build if you know that more than your source is required to build the software system?

Forces

- Components are not co-located on same system.
- System is complex and/or large
- Software build processes are complex
- Previous builds of software system must be reproduced

Solution

Document all of the components that contributed to the build in a list, i.e., a bill of materials (BOM). The BOM may contain the names, versions, and directory paths of operating systems, libraries, compilers, linkers, make-files, build scripts, etc. The BOM may be manually created, but many configuration management tools generate it as a by-product of the build. Since the BOM is also a file, it should be placed under version control and associated with the revision of version-controlled components that it documents. This can be done applying the same label to the version of the BOM that was used to identify the version of source code.

Resulting Context

The bill of materials identifies what components you need, where they can be found, what versions they are, and how to assemble them to reproduce the software system. The BOM can define the order in which the software components (source and libraries) are to be assembled (compile and/or link order). Although the BOM identifies the source code as one of the components of the build, the important purpose of the BOM is to identify components that are not under version control directly (e.g., environment information, compilers, linkers, et al).

Known Uses

Bills of material are commonly used by the manufacturing and construction industries for product data management. In their domain, the BOM provides a way of faithfully replicating the end product. It also serves as a cost analysis tool because it breaks down the end product into components that can be individually priced.
In the software domain, Continuus' ObjectMake™ product creates a list of the following to provide controlled, reproducible builds [Continuus]:

- Build platform data (OS, architecture)
- Full list of dependencies (controlled and uncontrolled objects)

- Commands and options used to create controlled product
- Makefile contents
- Makefile command line options
- Variant options
- Build environment

ClearCase and its "clearmake" tool have a notion of a "configuration record" (some other build tools have this as well, though not all use the same name). When clearmake does a build, it audits all access to all files in the workspace and records dependency information. When it builds an executable, it creates a configuration record (config-rec) for the build. The config-rec shows each pathname and revision of every file and derived file used and generated as part of the build. For each generated (derived) file such as object files, libraries, or GUI-code generation, it also records things like the time, the exact command-line and options or flags used to generate that object, and any important "flags" from the environment that clearmake knows about or was told to care about. ClearCase uses labels to select versions in a workspace corresponding to a config-rec. Config-rec's may (and often are) checked-in to the VC repository.

Related Patterns

A supporting pattern of *Bill of Materials* which is not documented in this paper is *Version-Control the Generated Deliverables* which describes placing generated object and/or executable components under version control for use in reconstruction.

Another pattern that supports *Bill of Materials* is *Version-Control the Environment* which is documented in this paper.

Self-Identifying Configuration

Context

You're tracking all changes made to the files using a version repository.

Problem

In isolation from the version repository, how do you know what version an extracted file is and what configuration it belongs to?

Forces

- Extracted files lose association with version control.
- Developers may not know what version they're changing once the file is outside of the version repository.
- Developers may not know the context (configuration) in which the file is being changed
- Reviewers aren't assured what versions of files are being reviewed.
- External identification is chaotic and unwieldy

Solution

Embed an identifier in configured item that identifies the version and optionally it's state. In some version control tools such as SCCS, this can be accomplished using expansion keys. Developers place special character strings (keys) in their files when checking them back in. The keys are expanded by the version control system when the file is extracted, placing the version information where the key was. This doesn't work for binary files, however.

Resulting Context

Files that are extracted from the version repository contain configuration identifiers that provide information to the developer. This information provides a guarantee that an extracted version of a file corresponds to a version in the repository and is viable for reviews, audits, etc. Hopefully developers are not malicious, but it must be noted that the expanded information can be altered in text files. Identifiers embedded in object code for later identification by tools like 'what' are more secure.

Known Uses

A project at AGCS uses expansion keys in files for self-identification purposes. The developers have placed SCCS keywords in each file that, upon extraction from the repository, expand to file name, version, promotion date/time, and characters that support the Unix 'what' command. The 'what' command examines object and executable code for expanded keyword information. This is used by the make-file link step to verify that the correct objects are being put together.

Related Patterns

This pattern is similar to *Embedded BOM* (not described in this paper), which describes another method of placing information inside source code for use after compilation. You add a character string of the form:

```
Static const char* vssid[]= ‘$Revision$
```

Then you can do a Unix 'strings' command on the object files to see what you actually built. A bit overkill, but an easy way to get validation when your build process isn’t very repeatable.

This pattern is also related to *IDBase* (not described in this paper) contributed by Paul Sander (pauls@broadvision.com). This pattern employs another method to identify and verify components in a built product with their sources. An ID base is a list of tuples consisting of file paths, version identifiers, and checksums. As the last step of the build process, a tool processes every "interesting" file to capture the tuple for the file and build the ID base. A second tool used to verify a file's membership in a build by producing its checksum and looking it up in the ID base. The ID base can be used as a BOM when selecting sources and to assess the accuracy of a reproduced build. Paul's use of the ID base is in post-release to support diagnosis of version mismatch problems at a customer site. His company ships the second tool as part of their product and keeps the ID bases internally.

Reproducible Build

Context

You have successfully built the software system once in the past and as a result, created a build process. Perhaps you've implemented the *Bill of Materials* pattern.

Problem

How do you know if the build process and/or bill of materials can faithfully reproduce the software system?

Forces

- The build process may not capture all of the components (completeness).
- The build process may not identify the correct versions and locations of components (correctness).

Solution

Test both the build process and bill of materials by producing a build from them and checking for differences between the initial build and the process-generated build. The simplest method would be looking at final file size of the executable(s). Commands like 'cmp' (Unix) can do a binary difference between the executables to show that something's lacking in the build process. Another Unix tool is 'spiff', a tool that looks for embedded information (e.g., time stamps). If the build process uses compilers that can do some nifty optimizing parallel stuff, you may never be able to count on the exact same sequential output ordering twice. In this case, running a regression test suite can prove that the executables are equivalent.

Debug any differences found and reiterate until no differences are found. *Named Stable Bases* [Cope95] recommends establishing a frequency of integrating and building software that manages the stability of the base.

Resulting Context

After implementing this pattern, the developers will be confident that the software loads are stable. Integrators will be confident that they can recreate software builds, especially if they need to back up to a previous state of the base.

Known Uses

AGCS has a CM group that applies this pattern to validate that the product that is being shipped to the customer. The AGCS CM team applies the build process that the engineers use and verify that the product delivered to the CM team for release to the customer can be produced by the build process.

One of the CM team leaders worked with a project group that didn't document the build process until some time after the product was released. When the customer reported problems with the product, the project group couldn't reproduce that version of the product to reproduce the problem in their own labs, and thus couldn't provide a fix for the

customer. The customer had to wait for the next major release of the product instead of receiving a repaired version of what they were using.

Related Patterns

Named Stable Bases and *Bill of Materials* can be used in association with this pattern.

Daily Build and Smoke Test [IEEE1]

Context

Lots of changes are being made to the software daily and by many developers.

Problem

How do you keep the changes from getting out of hand and contain the potential for errors in the build?

Forces

- Lot of effort to back out or fix problems that break the build.
- If there are many changes, the error(s) can be many layers under the first error discovered.
- Trying to isolate the change(s) that broke the build is difficult.

Solution

Build the software product daily to see if it still builds and runs successfully or if it smokes when it runs.

Resulting Context

The number of changes introduced into the build is more manageable in a daily build. The risk of breaking the build is reduced and defect diagnosis is easier. You know that if the product worked on one day and didn't the next, something added to the product in that span of time caused the build and/or product to fail.

It also prevents integration problems from consuming the project. It is important that the build is daily because it establishes a rhythm for the project. No one has to remember what day the build takes place. Developers also know that if their changes didn't make it into this build, they won't have to wait until some time later in the week.

Developer morale is improved. With daily builds, a bit more of the product works every day, and that keeps morale high.

Known Uses

Microsoft uses this pattern for the development of Windows NT. They set a specific time of day as a deadline for submittal of changes and builds would be scheduled around the deadline [IEEE1].

AGCS load integrators use this pattern to build group-wide loads for system testing. They build a load from developers' submissions on a daily basis. Developers can continue to submit changes to the build, but their changes will not be included unless exceptions are granted.

Related Patterns

This is an instantiation of *Named Stable Bases* where here the frequency is specifically one day.

Shared Source Cache

Context

You have a version repository in which you can identify viable sets of file versions. Developers are using private workspaces to do local edits or builds, but don't want to have the entire file set local to their workspace.

Problem

How do you support private workspaces without having the entire source code file set local to the developer?

Forces

- You don't have enough disk space on the developer's local platform to contain the entire source file set.
- There is a lot of overhead in managing a large private workspace; multiply this times the number of developers shows a huge productivity impact
- The local workspace is used only for edits or builds occur on another platform.

Solution

Create a common workspace that contains shared immutable objects. A project leader extracts viable versions and places them into the common workspace. This pattern serves as the foundation to *Derived Object Pool* where you can improve link performance by compiling the files in the common workspace.

Resulting Context

Developers' local file sets are constrained to files in which they are interested, making efficient use of their local disk space. Build scripts, development environment settings, etc., are set up to look locally first, followed by the location of the common workspace, creating local developer builds that are more consistent with each other.

Known Uses

An in-house CM tool at Honeywell CAS provided this feature. The tool provided the means to identify and configure source file versions. The configured source file versions were stored in a separate directory and could be referenced by users in their build processes. One example was providing a source of included files.

Related Patterns

Shared Object Cache.

Shared Object Cache

Context

Developers are performing local builds based on extracted files. Most of the file versions that they are extracting to their local workspace are in common with all of the other developers. Each developer separately compiles and links the same common file set.

Problem

How do you eliminate redundancy of effort in every developer compiling the same set of file versions?

Forces

- You don't have enough disk space on the developer's local platform to contain the object files produced by the entire source file set.
- There is a lot of overhead in managing a large private workspace; multiply this times the number of developers shows a huge productivity impact.
- Compiling the entire source file set takes a lot of time; multiply this times the number of developers shows a huge productivity impact.

Solution

Maintain a pool of derived (compiled) objects with associated information. This is done more efficiently when *Shared Version Cache* is implemented first. Developers' linkage paths point to their own local pool first followed by the common pool.

Resulting Context

Improved performance in total build time. Common file set is kept and managed in shared workspace, reducing the maintenance and build overhead for every developer.

Known Uses

A project at AGCS uses this pattern using a Unix technique called 'view-pathing.' The project administrator defines a common shared directory for the project developers. The administrator populates this directory with files from the version repository that meet specific criteria: the files compile and link cleanly, have passed review, and have met unit-testing requirements. Developers can set up a logical link to the common directory so their personal make-files will search their local directories first, then the common directory for source.

An in-house CM tool at Honeywell CAS provided this feature. The tool provided the means to identify and configure source file versions. The configured source file versions were compiled as they were inserted into the configuration, producing an associated object file. Users could include the path to this pool of associated object files in their link paths.

Related Patterns

Shared Version Cache provides the context for this pattern.

Shared-Source Escalation

Context

You have similar products/projects underway. You recognize that there are components of these projects that will be identical or have common components that have not diverged. Having two or more sets of identical components is a maintenance headache.

Problem

How can you support and maintain a common set of components used by more than one product/project?

Forces

- Portions of products/projects are similar.
- Product groups are providing a family of related products to provide flexible solutions to customers.
- Components or sub-products are discrete.
- There is overhead to supporting re-use.
- Product groups are challenged to bring products to market faster.

Solution

Make internal projects out of common code reused by multiple other projects. This requires 1) identifying the common components/products used by other projects, 2) establishing an independent project for these common components, and 3) establishing an internal release process of the common components to the individual projects.

Resulting Context

Shared components or products are maintained in one place rather than multiple places improving development efforts. A CCB (change-control board) may need to be put in place to coordinate and approve different groups' requirements for changes and enhancements of the common components. Over time different projects may find that some components need to be specialized. At that point they could take responsibility for their variant of the common components.

Known Uses

Ralph Cabrera (cabrerar@agcs.com**) contributes:** One of AGCS' product groups supports two parallel product efforts where one project manages an internal "sub-product" used by both projects. A file- sharing mechanism provided by an in-house CM tool allows the managing project to identify and make public those files that are part of the shared "sub-product". The borrowing project can select and deploy versions of the "sub-product" independent of the other project. Ownership is maintained by the managing project.

Related Patterns

None.

Version-Controlled Environment

Context

Customers, regulatory organizations, or the law may require that older versions of software systems be available for audit, debugging, patching, or enhancement. Only known and controlled changes (such as patches) may be introduced into the software system.

Problem

How do you faithfully and exactly reproduce builds of older versions of software systems?

Forces

- The level of software criticality requires exact reproduction of older versions of software.
- Certifying agencies like the FAA require exact reproduction of older versions of software.
- Operating systems change
- Tools (e.g., compilers) change
- Development environments change
- Level of impact that the environment has on the software system dictates scope of environment components.
- The version control system has sufficient storage capacity to contain versions of environment components.
- The version control system can handle versioning environment components (typically large binary files).

Solution

Identify the environment components that impact the software system and place these components under version control. Associate versions of the environment components with other components (e.g., source code) that contribute to a build, perhaps by using a common label.

Resulting Context

Software systems can be faithfully and exactly reproduced, depending on how close the re-created environment approximates the original. The combination of the bill of materials, environment repository, and/or vaulted components reduces variation in the reproduced build. Maintainers will find it easier to reproduce field-reported problems and not worry about introducing new problems due to changes in the environment.

If the environment components cannot be version-controlled or there isn't sufficient storage capacity to contain versions of these components, consider using the *Archived Environment* pattern.

Known Uses

Ralph Cabrera (cabrerar@agcs.com) contributes the following: Honeywell Commercial Avionics Systems (CAS) is required by the FAA to be able to reproduce software systems that have been released twenty-five years ago. Airplanes can be in service for any length of time; if problems are discovered in the version of software on an airplane, or an airplane crashes and software is one of the possible causes, CAS must first reproduce a build of the system to reproduce the problem or scenario.
Another reason for CAS to reproduce an exact build is mostly financial. Boeing may have enhancement requests of the older software and not be able to upgrade hardware or software. The enhanced system must not contain any changes other than the requested enhancements.

Andy Glew (glew@cs.wisc.edu) contributes the following: Some environments take advantage of Unix chroot 'boxes' to make it impossible to use other tools in your project by accident. (Editor's note: chroot defines what a user can see as the root path, aka '/'). Outside of such a strictly controlled environment, on machines where there is more than one compiler present, it is far too easy to forget about on tool's PATH variable or equivalent, and therefore end up with a build using tools that were otherwise not expected.
Since chroot 'boxes' correspond to directory trees, it would be straightforward to place all of the 'box' under version control (e.g., CVS), so that an update from the version control tool (CVS update) could update the tools in the 'box' (or not), as desired.

Ken MacLeod (ken@bitsko.slc.ut.us) contributes the following: Implementing SCM for the entire software environment on a host has been significantly easier over the past few years through pervasive use of software packaging and auto-installation. Ken has used Sun's JumpStart auto-installation system to maintain hosts to specification. The specification file is a post-install script that drives the installation of software and configuration files after the base OS has been installed. Any software that can be packaged can be added to the base-OS package installation and included in the base bill of materials. Configuration files are SCM'd through normal SCM tools. Short and mid-term updates are usually performed as package replacements and overlaying new configuration files. An audit facility for comparing a running system to 'what will be built next time' is composed primarily of doing a package cross-check, a known list of changed configuration files from SCM, and an audit of the remaining files that should not have changed since last install. We also wrote offline and online availability of regression tests for each application system to be used as part of installation testing, auditing, and disaster recovery.

Related Patterns

Archived Environment.

Archived Environment

Context

Customers, regulatory organizations, or the law may require that older versions of software systems be available for audit, debugging, patching, or enhancement. Only known and controlled changes (such as patches) may be introduced into the software system. *Version-Controlled Environment* doesn't cover everything that contributed to the software system.

Problem

What do you do with environment components that cannot be put under version control?

Forces

- The level of software criticality requires exact reproduction of older versions of software.
- Certifying agencies like the FAA require exact reproduction of older versions of software.
- Operating systems change
- Tools (e.g., compilers) change
- Development environments change
- Hardware changes.
- Level of impact that the environment has on the software system dictates scope of environment components
- Limited amount of space prohibits version control of the environment

Solution

Put the environment components that cannot be version-controlled (like operating system CDs or tapes) into vaults with labels that associate them with releases of software. Identify these archived components in the bill of materials.

Resulting Context

Software systems can be faithfully and exactly reproduced, depending on how close the re-created environment approximates the original. The combination of the bill of materials, environment repository, and/or vaulted components reduces variation in the reproduced build. Maintainers will find it easier to reproduce field-reported problems and not worry about introducing new problems due to changes in the environment.

Known Uses

Ralph Cabrera (cabrerar@agcs.com**) contributes the following**: Honeywell Commercial Avionics Systems (CAS) is required by the FAA to be able to reproduce software systems that have been released twenty-five years ago. Honeywell places computing environment operating systems onto tapes that are sent to an off-site vault. The tapes are labeled to associate them with product releases. Honeywell also retains

necessary hardware platforms in storage until permitted to obsolete them as software products are retired.

Related Patterns

Version-Controlled Environment

Acknowledgements

The authors would like to give special thanks to the following people for their significant contributions:

- Andy Glew
- Keith McLeod
- Melvyn Jacobs
- Linda Rising
- David Kane
- Paul Sander

References

[Appleton97] Brad Appleton; *Patterns and Software: Essential Concepts and Terminology*; Object Magazine Online http://www.sigs.com/omo/, May 1997, Vol. 3 No. 5; http://www.enteract.com/~bradapp/docs/patterns-intro.html

[Beedle97] Michael A. Beedle; "*cOOherentBPR - A pattern language to build agile organizations*"; in **PLoP/Allerton Park 1997 Proceedings**; Washington University Technical Report #wucs-97-34

[OrgPats] *Organizational Patterns Wiki Web*; http://www.bell-labs.com/cgi-user/OrgPatterns/OrgPatterns

[Berczuk95] Stephen P. Berczuk; *Patterns for Separating Assembly and Processing*; in **Pattern Languages of Program Design**, James O. Coplien, Douglas C. Schmidt (Ed.), Addison-Wesley, 1995, pp. 521-528

[Berczuk96] Stephen P. Berczuk; *Organizational Multiplexing: Patterns for Processing Satellite Telemetry with Distributed Teams*; in **Pattern Languages of Program Design 2** J. Vlissides, J. Coplien and N. Kerth, editors; Addison-Wesley, 1996, pp. 193-206

[POSA] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerald, Michael Stal; **Pattern-Oriented Software Architecture: A System Of Patterns**; John Wiley & Sons, 1996

[Continuus] Continuus; Change Management for Software Development, *Chapter 6: Build Management*, Copyright 1998. (http://www.continuus.com)

[Cope95] James O. Coplien; *A Generative Development-Process Pattern Language*; in **Pattern Languages of Program Design** J. Coplien and D. Schmidt, editors; Addison-Wesley, 195, pp. 224-225

[IEEE1] Steve McConnell; *Best Practices*, IEEE Software, Vol. 13, No. 4, July 1996.

[Appleton et al] Brad Appleton, Stephen P. Berczuk, Ralph Cabrera, Robert Orenstein; *Streamed Lines*: Branching Patterns for Parallel Development; PloP 1998; http://www.enteract.com/~bradapp/acme/branching/.