

Patterns for Configuration Management

Jürgen Salecker
Siemens AG, CT SE 2
Otto-Hahn-Ring 6, 81730 München, Germany
juergen.salecker@siemens.com

This pattern language will collect a series of patterns in the area of configuration management. The audiences are experienced developers and software architects. This pattern language should provide them with material to ask focused questions when it comes to select and setup a configuration management infra-structure for a software development. The pattern language is configuration management system independent. However some of the patterns will add constraints to the configuration management system. This will be explicitly described and if possible workarounds suggested.

Introduction The patterns presented in this paper are:

Work-Area-Validation

A broken integration build is introduced by a developer who has forgotten to check-in a new artifact. The solution is to prohibit the check-in as long as there is at least one artifact within his work-area which is not under configuration management system control.

Proposal-For-Check-In

In the development of complex and safety critical system it is a requirement (CMMI level 3) and good working practice to execute an audit about a planned check-in operation. However due to the information flood of check-in operations and also having usually no easy way to undo a check-in operation (a potential re-merge might have been executed) the development organization will be challenged significantly. A solution is to separate the concerns of micro-backups from check-in operations which decreases the information flood. Furthermore applying a simple state management for all

check-in operations will provide an effective methodology for audit operations. Both solutions combined will reduce the challenges a development organization is faced and provides the foundation for more advanced patterns in this context later on.

Credits

My shepherd Lise Hvatum was a great help to get the amount of information better structured. She initiated the collection into a pattern language and made sure that the paper got more focused.

Work-Area-Validation

The *Work-Area-Validation* configuration management pattern offers a solution to avoid broken builds in the case the developer has forgotten to check-in essential artifacts.

Context The pattern applies to any developer who is using a configuration management system and this even for teams with just one member.

Example A developer requires some additional software to accomplish his task. He starts searching the Internet and is lucky and downloads the found Software into his work area. He integrates this new piece of code into his work and hurries up to check-in because the daily build will start soon. Before the check-in he did – of course – a compile and builds. Everything was fine; however he is in a hurry due to late night dinner with his new girl friend.

Problem Let's further expand the example described above. Due to the fact the developer is not 100% concentrated, he forget to put his new piece of Software under configuration management control. Therefore this new piece of code is **not** part of his check-in. Even though the build was working in his environment, it will break the build of the common environment, causing delays and unnecessary work for other developers and the integration manager.

Solution This pattern prohibits any check-in, or even a proposal for check-in as long as there exists at least one artifact which is not under configuration management control. However generated artifacts, like object files, libraries, etc. have to be defined by an exception list. Providing this list from a central place ensures that once a check-in has been made it cannot happen anymore that artifacts which are required for a build are forgotten to check-in.

The solution in a nutshell:

1. First define a list of files, or regular expressions for files which are not required to be under version control.
Those files are typically generated object files, like *.class*, *.o* and similar ones.
2. Furthermore override the default check-in command with a customized script which examines when called first the list of

private files, extracts from this list the defined exceptions and finally calls the real check-in command if the calculated list was empty.

Implementation

This pattern has been implemented as an ANT task for validating a ClearCase work-area. In order to avoid an unnecessary generation of pages the complete source code will not be published here in this pattern. However in case of interest the author provides any reader with the working source code for a ClearCase work area. The custom ANT task is licensed under the GPL.

The implementation in a nutshell:

ClearCase provides the following command for getting the list of files within a work-area which are not under version control:

```
cleartool lspriv
```

Example ANT task *cc-val* invocation:

```
<target name="validate">
<echo>Files not under CM control:</echo>
  <cc-val filter="[checkedout],.bak,.contrib,.keep"
    regexpfilter=".*tmplogDir.*.*install.*"
    failifexecutionfails="false" />
</target>
```

In the above example a set of files are defined as exceptions, i.e. are extracted from the list of files not under version control. These are all files with the word *checkedout* in the file name and all files which have the extension, *.bak*, *.contrib* and *.keep*. Furthermore all files within the directories *tmplogDir* and *install* are being extracted from the list of files not under version control.

Known Uses

Omniworks is a configuration management system used within Schlumberger where the application of this pattern is standard since several decades.

At Siemens CT SE 2 it is used on ClearCase UCM (UCM = Unified-Change-Management) where the appropriated ClearCase UCM commands and the exception list is managed by a custom ANT task on top of ClearCase. ClearCase itself does not provide this feature out-of-the-box.

Consequences

The pattern provides the following benefits:

- Even for developers in a hurry (which is anyway the normal case) it is impossible to forget relevant artifacts for a check-in.
- Very effective method for teaching newbie's a disciplined working style.
- Even oldie's and guru's are able to keep their work-area clean by the application of this pattern.
- Provides a way to make the crap in a work-area visible.

The pattern provides the following liabilities:

- Sometimes it might be required that a check-in is enforced, despite the fact that there are some artifacts in the work-area which are not under version control.

Proposal-For-Check-In

The *Proposal-For-Check-In* configuration management pattern offers a solution for auditing check-in operations and provides a very effective solution for micro-backups. Micro-backups are daily, hourly, or even minutely backups for a set of files a developer is currently working on.

Context This pattern applies to the development of safety related and complex systems. Furthermore it is very well suited for distributed development environments and IT infrastructures which have a boundary (firewall) between a LAB network and the “official” Intranet.

Example Consider a development team which needs to expand as fast as possible. Using classical project management measures exclusively to get new members up-to-speed is not very satisfactory, nor efficient, because it will get very time consuming to provide detailed feedback about their work. Instead of frequent meetings, or worst case telephone conferences, use the configuration management system to provide timely feedback for ongoing changes and therefore increase the learning speed of the newbie’s.

Problem The effective (low overhead) controlling of the software architecture under development is one of the biggest problems in typical development environments. Frameworks like CMMI suggest regularly audits before code check-in which is one requirement for achieving a level 3 grade. Even though the intention of those audits is a positive one, the real problems lies in its implementation. How is it possible to execute those audits within real live high speed developments? Without proper tool support those reviews will have the power of a paper tiger and will most likely create too much administrative overhead.

Furthermore the common habit of using check-in operations for micro-backups creates an inflation of versions for software artifacts and a flood of check-in operations. This again makes an audit of check-in operations a very cumbersome procedure.

Solution Split the repositories of a configuration management server into a temporary and a permanent one. Check-in operations are exclusively collected by the temporary repository. A check-in operation migrates to the permanent repository only if it gets *accepted* by a stakeholder. Furthermore any *save*

operation executed within a developer's work-area creates a micro-backup with a temporary version id in the temporary repository.

The temporary repository will manage the micro-backups and the state machine of the *changes* provided by developers. The micro-backups are managed by the state transitions *save* and *restore*, where as a check-in is managed by the state transitions *propose*, *accept* and potential *reject*.

The overall concept is outlined in the figure below:

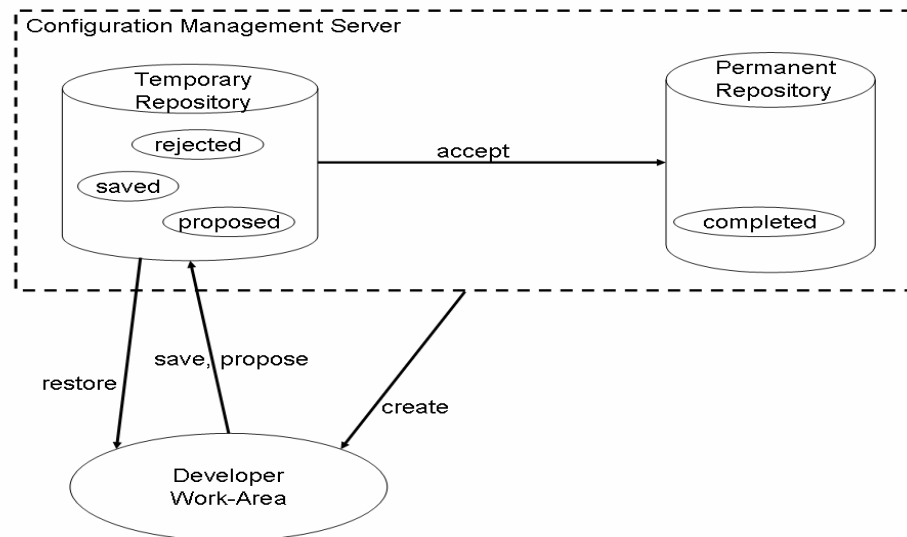


Figure 1, Temporary and Permanent Repositories

The above figure shows the different states a *change* can have together with its location within the configuration management server and developer work-area. *Changes* with the state *complete* can exclusively exist in the permanent repository. A *change* is considered here as a container which can contain one or more software artifact (files, for example) and created by the developers as necessary. Usually *changes* are provided as configuration management objects, typical names are: revisions (subversion), task (CM Synergy), activities (ClearCase UCM), or changesets (Bitkeeper, Perforce).

Changes are migrated from the temporary repository to the permanent repository only in the case the change has been *accepted* by the project leader or stakeholder responsible.

The detailed state diagram including the necessary transitions is shown in the figure below:

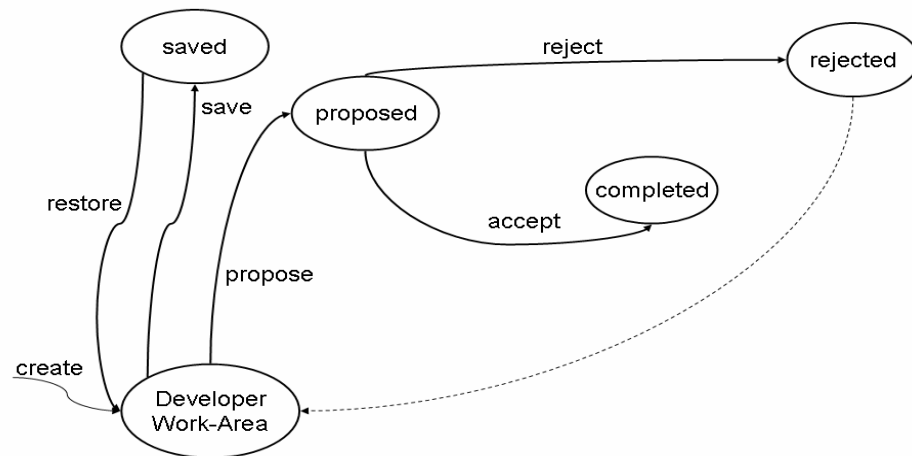


Figure 2, State Machine for Managing Changes

A *save* transaction might be executed as many times as the developer requires a micro backup. Every *save* transaction will copy the associated objects (source code files) into the temporary repository and create a new version of every artifact (source code file). The *restore* transaction is the opposite of the *save* transaction, it copies the selected version into the developer work area.

A *propose* transaction will be executed from the developer when he would like to **check-in** his change. This transaction will copy all associated objects into the temporary repository as the *save* transaction. However the state of the change will be changed to *proposed*.

Only if a change has the state *proposed* the project leader is able to either *accept*, or *reject* the change. The criteria to *accept* a change depends on the development organization, the phase the development and the type of system under development. For safety critical system development the acceptance criteria might be higher as for systems with fewer requirements in this area.

If the change is *accepted* the state will change to *completed* and the content of the change will be copied into the permanent repository. In the other case the change got *rejected*, it will stay in the temporary repository and the

developer will get a notification informing him about the new state together with feedback about the reason.

The following forces did influence the solution:

- The possibility to reject a check-in – due to missing defined quality attributes - before it actually migrates into the configuration system and creates trouble.
- Being able to provide feedback about a check-in as early as possible, because feedback not provided in a timely manner is more or less useless.
- Avoid an inflation of versions caused by micro-backups and not for having finished some work.
- Providing even developers a chance to reject their own work after having slept over the proposed change without having to go through cumbersome re-merge procedure.
- Being able to come easily across different network structures by using the configuration management server as a data exchange engine.

Implementation

To the knowledge of the author this pattern is not available yet in any open source or commercial configuration management system. However a possible workaround would be to add a simple versioning system (SCCS, PVCS, SourceSafe, CVS) in front of a configuration management system (ClearCase, Perforce, Bitkeeper, CM Synergy), where the simple system just manages the temporary repository. In case of a *accept* transaction the changes are checked-in into the configuration management system, executed by a script running on top of the simple versioning system

Known Uses

Omniworks a sophisticated configuration management system used within Schlumberger, not available for the public. This pattern is a standard procedure for software development at Schlumberger already over several decades.

Somehow a flavor from this pattern is used by the Open Source community, where developers have to apply for becoming the right to commit changes into a repository. They might also loose this right (somehow a global reject), depending on the community.

Consequences The pattern provides the following benefits:

- It increases the transparency within a development organization without the abuse of phone conferences, frequent meetings or cumbersome quality audit procedures without proper tool support.
- The overall workload for a project leader and software architect will be significant lower in the medium and long term.

The pattern provides the following liabilities:

- The work load for the project leader will be a higher in the short time (training, getting used to the concept).
- If the feedback is not provided in a human friendly way, there exists the danger that people will become very upset, therefore decreasing productivity.
- HR departments (or similar ones, like QA) might get the idea to use the number of rejections per developer and time as some kind of measure, for what ever reason. If this happens, immediately abandon the solution presented by this pattern and change business.