**American Journal of Engineering and Applied Sciences, 2012, 5 (3), 261-265**
ISSN: 1941-7020
© 2014 Kaur and Singh, This open access article is distributed under a Creative Commons Attribution (CC-BY) 3.0 license
doi:10.3844/ajeassp.2012.261.265 Published Online 5 (3) 2012 (http://www.thescipub.com/ajeas.toc)

# Configuration Management Issues in Software Process Management

**Parminder Kaur and Hardeep Singh**

Department of Computer Science and Engineering, Guru Nanak Dev University, Amritsar-143005, India

Received 2012-03-12, Revised 2012-09-02; Accepted 2012-09-14

## ABSTRACT

A software development process is concerned primarily with the production aspect especially the management of software development. The development of a software process passes through various phases and there is a need to manage all issues particularly configuration issues during the evolution of a software process. This study makes an attempt to deal with various configuration issues with the help of an opensource configuration management tool. The analysis of different software development paradigms is also presented in order to discuss the brief explanation with respect to software process management.

**Keywords:** Component-Based Development (CBD), Concurrent Version System (CVS), Open Software Description (OSD), ClearCase (CLE), SourceSafe (VSS)

## 1. INTRODUCTION

Software process research deals with the methods and technologies used to assess, support and improve software development activities. Component-Based Development (CBD) has emerged as a key element in the development of complex software systems within the domain of software processes. It follows the principle of "divide and conquer" for managing complexity i.e., breaking a large problem into smaller pieces and solves those smaller pieces, then build up more elaborate solutions from simpler foundations (Sametinger, 2001; Brown, 1998; Pressman and Pressman, 2004). Component technology offers the potential to assemble applications much more rapidly than ever before. A key to assembling applications quickly is the ability to reuse existing prefabricated components to meet the desired requirements of the application (Szyperski *et al.*, 2002; Brown, 2000; Heineman and Councill, 2001; Wallnau, 2002).

Traditional software process development follows two approaches: One, when the software is developed entirely from scratch and the other, where everything is outsourced. Each component is developed as a standardized product, with all associated advantages. The components are available at different prices and with different qualities like level of performance, resource efficiency, robustness and degree of certification. Some individual components can also be custom-made so that they could meet the specific requirements or to foster strategic advantages.

The major requirement of component-based systems is to manage the life-cycle evolution of software components. As change occurs, new revision/variant of the existing component takes place. The satisfactory result of that revision/variant becomes the basis of next version. Revisions can be performed in a serial as well as parallel fashion (i.e., by a single person or by a group of persons at a same time). So, the need exists to keep the track of multiple versions of constituent components. Configuration management is used for retrieving the information about the system with respect to various changes of available components.

To keep track of changes or to maintain the evolution history of the components, various open-source as well as commercial version control systems are available. Various version control tools, like SCCS (Rochkind, 1975), RCS (Tichy, 1985), Perforce (PER), BitKeeper (BIT), ClearCase (CLE), SourceSafe (VSS), Concurrent Version System (CVS) and Subversion (SUB), provide support for configuration identification and version control, allowing the software development to be integrated directly with configuration management processes.

**Corresponding Author:** Parminder Kaur, Department of Computer Science and Engineering, Guru Nanak Dev University, Amritsar-143005, India

### 1.1. Software Development Approaches

A software system can be understood from a life-cycle process of system development. There are several different approaches, which can be considered for the development life-cycle of software system. All these approaches are based on the same activities such as (Brown, 2000):

- Requirement analysis and system specification
- System and software design
- Implementation and unit testing
- Integration, system verification and validation
- Operation support and maintenance
- Disposal

Sequential Model, Evolutionary Development Models, Unified Process and Component-Based Development are some of the different software development approaches. A brief description of these models is as follows.

### 1.2. The Sequential Model

The sequential model e.g., a waterfall model follows a systematic, sequential approach that begins at system level and progresses successively at each stage. The output from one activity becomes the input for the next activity. The disadvantage of this approach is that it requires defining and describing all system as well as software requirements, beforehand, by the customer explicitly. It is also very difficult to add or modify any requirement during the development process. Another problem, with the sequential model is the late response to the customer and by that time, working version of the model may become ineffective.

The sequential model provides a template onto which methods for analysis, design, implementation, integration, verification, validation and maintenance can be placed. Therefore it has remained the most influential software development process model.

### 1.3. Evolutionary Development Models

The basic principle of evolutionary development models is to develop a system in various stages and each stage helps in increasing the knowledge about system requirements and functionality. This model reduces the risk of detecting critical problems in later phases of the development. Iterative approach, Incremental model and Prototyping model are based on the principles of evolutionary development approach (Pfleeger and Kitchenham, 2001; Wallnau, 2002). Boehm has combined all these approaches in a model, in which, activities are performed several times in an iterative manner, beginning with a base functionality and addressing issues like objective setting, risk assessment and reduction, development, validation and planning for the next loop Boehm and Basili (2000). The iteration process can be concluded when a complete working software system has been developed.

The disadvantage of this approach is the increased difficulty of project coordination and evaluation. It is also difficult to determine the exact number of iterations, as new iterations may get added due to the occurrence of changes.

### 1.4. Unified Process

Unified Process, developed by Jacobson for Object-Oriented and Component-Based Systems, was an iterative incremental development process (Jacobson *et al.*, 1999). This process incorporates four phases named as Inception (the phase, in which the system is described in a formalized way), Elaboration (the phase, in which the system architecture is defined and created), Construction (the development of completely new products with reuse capabilities) and Transition (installation of the system and training of its users).

Several iterations of core activities like requirements, analysis, design, implementation and test occur in each of these four phases. The incremental part of unified process is based on the fact that successful iterations will result in the release of a system. Unified Process has the advantages of both incremental and iterative models. It also inherits the disadvantages of both incremental and iterative approaches.

### 1.5. Social Analysis of Software Development Approaches

In any social process, interactions occur among components playing particular roles. Many roles are generic as they appear in methodologies within all development paradigms. This section provides the social implications of development paradigms like Traditional Life Cycle, Iterative-Incremental and Component-based Development, using a multidimensional framework as shown in **Fig. 1**.

### 1.6. Traditional Life Cycle Paradigm

The traditional life-cycle paradigm follows a linear approach to systems development, processing through analysis, design, coding, testing and maintenance. Traditional approaches tend to be developer-centered. Although request for application to be developed is initiated by the user, yet, developers control the development process. Methodologies within this paradigm tend to be structured and formal for the judgment of acceptability at each phase. One methodology that exemplifies this approach is structured analysis and design (Yourdon, 1989).

### 1.7. Iterative-Incremental Paradigm

The iterative-incremental paradigm follows an iterative process, repeating various activities until design specifications are better understood and fully developed. Methodologies within this paradigm are neither developer centered nor user centered. The developer may direct some activities but results are obtained by their joint responsibility and effort. Many iterative-incremental methodologies fit within an Object-Oriented paradigm that focuses on software reuse. The most general methodology for this paradigm is prototyping.

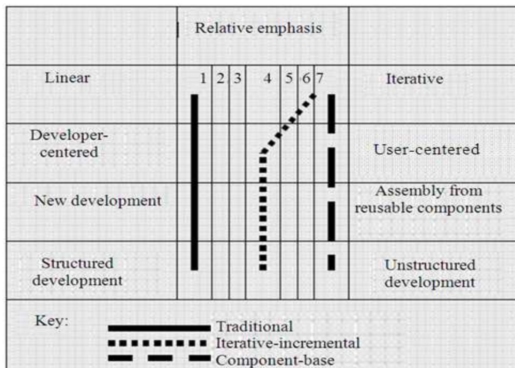


**Fig. 1.** Social Analysis of development paradigms (Robey *et al*., 2001)

## 1.8. Component-Based Development Paradigm

Component-Based Development (CBD) paradigm depends upon the availability of a wide variety of reliable utilities and business-application components so that they can be easily created and configured (Nierstrasz *et al*., 1992). Unlike objects, components are platform dependent and thus concrete enough to avoid the risks and problems of instantiating general objects on a particular machine and within a specific application at the language level. Most component-based development relies upon the use and reuse of components available from independent component suppliers.

Today, there are three major forces in component software arena-Object Management Group, with its CORBA-based standards (COR), (Marvie and Merle, 2001), Microsoft, with its COM-based standards (COM), (MCDEC, 1995) and Sun Microsystems with its Java-based standards (JAVA), (Frederic *et al*., 2009). These component models focus on corporate enterprise, desktop and network solutions. The ready availability of commercial component-based infrastructures e.g., COM/COM+/DCOM/.NET, JAVA, CORBA and plug-ins for software such as Adobe Acrobat, Visual BASIC (Shapiro, 2002) and Netscape have made component-based development a reality. Companies such as ComponentSource.com, Flashline.com, ILOG and Rogue Wave Software sell thousands of ready-made components, mostly in the COM, Java, C, C++, Delphi and .NET categories and generate substantial revenues.

The major requirement of component-based systems is to manage the life-cycle evolution of software components. As change occurs, new revision/variant of the existing component takes place. The satisfactory result of that revision/variant becomes the basis of next version. Revisions can be performed in a serial as well as parallel fashion (i.e., by a single person or by a group of persons at a same time). So, there is a need to keep track of multiple versions of constituent components. To keep track of changes or to maintain the evolution history of the components, various open-source as well as commercial version control systems are available. Various version control tools, like SCCS (Rochkind, 1975), RCS (Tichy, 1985), Perforce (PER), BitKeeper (BIT), ClearCase (CLE), SourceSafe (VSS), Concurrent Version System (CVS) and Subversion (SUB), provide support for configuration identification and version control, allowing the software development to be integrated directly with configuration management processes.

## 1.9. Component Configuration Management

In component based systems, it is difficult to manage components during the lifetime of a system. A system of components is usually configured only during the buildtime when known and tested versions of components are used. When new versions of components are evolved, the system itself has no method to detect the recent installed components. There might be a check that the version of replaced component is at least the same as or newer than the original version, in order to ensure the 100% functionality of new component. Some sort of mechanism must be present in the system to check the version of replaced component. This mechanism prevents the system from using old components, but it does not guarantee system's functionality when new components are installed (Larsson and Crnkovic, 1999; Crnkovic and Larsson, 2002).

Configuration Management (CM) refers to a disciplined approach to manage the evolution process of software development and maintenance. It manages the artefacts produced in the development process, controls the changes to the software and its components. It helps in managing the systems built with components and checking dependencies between components during evolution process. Some level of configuration control can be achieved if it is possible to identify components with their version and dependencies to other components. CM is the art of keeping track of which items within a product have changed, how they have changed and how they are combined. It is who, what, when, why and how of every change, system build and integration.

To identify the change in the system, following points are to be considered:

- Identification of components including their versions
- Identification of direct and indirect independencies
- Get required information to confine the implicit dependencies

To get the full dependency graphs about all the components and the type of change occurred in a component, there is a need of meta-data. Meta-data provides the information like name, creation date, version, compatibility change, provided interfaces and required interfaces, which is helpful during the building of a system with consistent configuration management. Open Software Description (OSD) (W3C), an XML-based language, is

defined as a standard to describe components and their dependencies by World Web Consortium. Tools like Subversion (SUB), CVS (CVS, 2009) can also be used to describe the dependencies at build-time.

## 1.10. Experimental Work

The tool "*Dependency Walker*" (DPW) has been used to find dependencies by parsing the components. It is used for the evaluation of the presented configuration model. It parses through the system, finds all shared libraries and generates the dependency graph. Scanning all shared libraries and executables in a system creates a dependency graph. Various features of the tool then extend this graph. Processes can be supervised and when new components are dynamically loaded into the memory, the graph is extended with dynamic dependencies.

As the new version of the component is installed, it is the task of component configuration management to handle all the conflicts. Because in such a case, the new component may have some additional dependent files, so these are the issues to be handled by version management. The information with respect to various versions can be obtained using this tool.

Various versions of Adobe Acrobat Reader, Netscape Navigator, Internet Explorer, Windows Movie Maker [registered products of respective owners] are studied to check the dependencies between components and their shared libraries and it is found that as new version evolves, changes in dependencies occur.

This case study, **Table 1**, shows that different versions of a software product operated in same environment, have different number of dependencies.

For example version V5 and V6 of Adobe Acrobat Reader share the similar development patterns as well as dependencies where as version V7 onwards indicate a change in design. The reduced number of dependencies may indicate toward the simple architecture of the component integration. Same situation can be seen in case of Moviemaker software. Similarly, in case of Netscape Navigator and Internet Explorer change in design has taken place. This shows that there exists a relationship between dependencies and functionalities provided by the respective software. This indicates that with simple architecture, a system of components can be updated with enhanced features. Subversion, an open source project that attempts to remain as similar as possible to CVS while improving its capabilities with additional features, is used to keep check on configuration management activities. **Table 2** enlists various features of Subversion. Subversion offers directory versioning. Also, it is easy to handle file name changes in Subversion than in CVS, which requires a combined copy and deletion to rename a file.

**Table 1.** Number of major *dll* files available in different versions of Adobe Acrobat Reader, Netscape Navigator, Internet Explorer and Windows Movie Maker (---- shows that related version are not available to us)

| .dll files Version | Adobe acrobat reader | Netscape navigator | Internet explorer | Moviemaker |
|---|---|---|---|---|
| 5.1 | 18 | ---- | ---- | 23 |
| 6.0 | 21 | 8 | 5 | 3 |
| 6.2 | ---- | 9 | ---- | ---- |
| 7.0 | 6 | 9 | 14 | ---- |
| 8.0 | 5 | 10 | ---- | ---- |
| 9.0 | 6 | 20 | ---- | ---- |

**Table 2.** Comparison of version control systems (VCID, 2012)

| Feature | CVS | Subversion | Aegis | BitKeeper | SourceSafe | Perforce | ClearCase | Synergy |
|---|---|---|---|---|---|---|---|---|
| Platforms | MS Windows (clients), UNIX | MS Windows UNIX | UNIX | MS Windows, UNIX | MS Windows | MS Windows, UNIX | MS Windows UNIX | MS Windows UNIX |
| Atomic Commits | No | Yes | Yes | Yes | No | Yes | Yes | Yes |
| Tracking Uncommitted Changes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes |
| File/Directory moves, renames | No | Yes | Yes | Yes | Yes | Yes | Yes | Yes |
| Remote Repository replication | No | Yes, via tool | Yes | Yes | No | Yes, via tool | Yes, via tool | Yes |
| Propagating changes to Parent repository | No | Yes, via tool | Yes | Yes | No | No | Yes, via tool | Yes |
| Repository Permissions | Limited | Yes | Yes | Yes | Limited | Yes | Yes | No |
| Line-wise history tracking | Yes | Yes | Yes | Yes | Not directly | Yes | Yes | Via scripting |
| Ease of deployment | Good | Good | Medium | Good | Very Good | Very Good | Poor | Good |
| Networking support | Good | Very Good | Poor | Good | Good | Good | Poor | Good |
| Portability | Good | Excellent | Medium | Very good | Good | Excellent | Medium | Very good |
| License | Open source | Open source | Open Source | Proprietary | Proprietary | Proprietary | Proprietary | Proprietary |

Subversion lets us to create and store arbitrary properties, called *metadata* along with any file or directory; it creates versions of the file properties just as it does for the file contents. It also allows treating a collection of files or directory modifications as a single unit.

Subversion is a powerful tool that can help solve many problems arising in cooperative and distributed development. Subversion is targeted at text files as this allows subversion to merge documents that are edited at the same time by different people. Subversion can even cope with conflicting edits (e.g., two persons changing the same line). Unfortunately, this is not possible for binary files such as Word documents. For Word documents, one can use the internal tracking feature to tell other people about changes, but there is no possibility to merge two Word documents. As such, Word documents should never be edited by two persons at the same time.

## 2. CONCLUSION

Component-based systems are becoming increasingly important in software process management. The continuous change in component-based systems, demand for an efficient version control mechanism. Tools like Dependency Walker as well as Subversion prove helpful in keeping the track of changes during the evolution of software. Various configuration control issues can be solved with the use of these tools. Future work includes the validation of these tools on a large networked data and design of an automated tool which helps in detecting the configuration issues during the installation of a new component.

## 3. REFERENCES


1. Boehm, B. and V.R. Basili, 2000. Gaining intellectual control of software development. Computer, 33: 27-33. DOI: 10.1109/MC.2000.841781
2. Shapiro, J.R., 2002. The Complete Reference: Visual Basic. NET. 1st Edn., Tata McGraw-Hill, New York, ISBN-10: 0070495114, pp: 865.
3. Brown, A.W., 1998. An overview of CBD.
4. Brown, A.W., 2000. Large-Scale, Component-Based Development. 1st Edn., Prentice Hall PTR, Upper Saddle River, ISBN-10: 013088720X, pp: 286.
5. Crnkovic, I. and M. Larsson, 2000. A case study: Demands on component-based development. Proceedings of the International Conference on Software Engineering, Jun. 04-11, IEEE Xplore Press, Limerick, pp: 23-31. DOI: 10.1109/ICSE.2000.870393
6. CVS, 2009. CVS-concurrent version control system.
7. Robey, D., R. Welke and D. Turk, 2001. Traditional, iterative and component-based development: A social analysis of software development paradigms. Inform. Technol. Manage., 2: 53-70. DOI: 10.1023/A:1009982704160
8. Frederic, P., M. Agnes, F. Vandome and J. McBrewster, 2009. Java (software platform).
9. Heineman, G.T. and W.T. Councill, 2001. Component-Based Software Engineering: Putting the Pieces Together. 1st Edn., Addison-Wesley, Boston, ISBN-10: 0201704854, pp: 818.
10. Jacobson, I., G. Booch and J. Rambaugh, 1999. The United Software Development Process. 1st Edn., Addison Wesley, ISBN-10: 0201571692, pp: 512.
11. Larsson, M. and I. Crnkovic, 1999. New challenges for configuration management. Proceedings of the 9th International Symposium on System Configuration Management, (SCM' 99), Springer-Verlag London, UK., pp: 232-243. http://dl.acm.org/citation.cfm?id=719006
12. Marvie, R. and P. Merle, 2001. CORBA component model: Discussion and use with open CCM. The Pennsylvania State University. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.2.2486
13. MCDEC, 1995. The component object model specification. Microsoft Corporation. http://www.daimi.au.dk/~datpete/COT/COM_SPEC/pdf/com_spec.pdf
14. Nierstrasz, O., S. Gibbs and D. Tsichritzis, 1992. Component-oriented software development. Commun. ACM, 35: 160-165. DOI: 10.1145/130994.131005
15. Pfleeger, S.L. and B.A. Kitchenham, 2001. Principles of survey research: part 1: turning lemons into lemonade. ACM SIGSOFT Software Eng. Notes, 26: 16-18. DOI: 10.1145/505532.505535
16. Pressman, R. and R. Pressman, 2004. Software Engineering: A Practitioner's Approach. 6th Edn., McGraw-Hill Science/Engineering/Math, ISBN-10: 007301933X, pp: 880.
17. Rochkind, M.J., 1975. The source code control system. IEEE Trans. Software Eng., SE-1: 364-370. http://www.basepath.com/aup/talks/SCCS-Slideshow.pdf
18. Sametinger, J., 2001. Software Engineering with Reusable Components. 1st Edn., Springer, Berlin, ISBN-10: 3540626956, pp: 272.
19. Szyperski, C., D. Gruntz and S. Murer, 2002. Component Software: Beyond Object-Oriented Programming. 2nd Edn., ACM Press, London, ISBN-10: 0201745720, pp: 589.
20. Tichy, W.F., 1985. Rcs-a system for version control. Software: Practice Exp., 15: 637-654. DOI: 10.1002/spe.4380150703
21. VCID, 2012. Better SCM initiative: Comparison.
22. Wallnau, K., 2000. Technical Concepts of Component-based Software Engineering. 1st Edn., Carnegie Mellon University, Pittsburgh, pp: 53.
23. Yourdon, E., 1989. Modern Structured Analysis. 1st Edn., Yourdon Press, Mexico, ISBN-10: 0135986249, pp: 672.