A DYNAMIC SOFTWARE CONFIGURATION MANAGEMENT SYSTEM

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF MIDDLE EAST TECHNICAL UNIVERSITY
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

FATMA GÜLŞAH KANDEMİR

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

SEPTEMBER 2012

Approval of the thesis:

A DYNAMIC SOFTWARE CONFIGURATION MANAGEMENT SYSTEM

submitted by **FATMA GÜLŞAH KANDEMİR** in partial fulfillment of the requirements for the degree of
Master of Science in Computer Engineering Department, Middle East Technical University by,

Prof. Dr. Canan Özgen
Dean, Graduate School of **Natural and Applied Sciences** ________________

Prof. Dr. Adnan Yazıcı
Head of Department, **Computer Engineering** ________________

Assoc. Prof. Ali Hikmet Doğru
Supervisor, **Computer Engineering Dept., METU** ________________

Dr. Cengiz Erbaş
Co-supervisor, **ASELSAN** ________________

Examining Committee Members:

Assoc. Prof. Ahmet Coşar
Computer Engineering Dept., METU ________________

Assoc. Prof. Ali Hikmet Doğru
Computer Engineering Dept., METU ________________

Dr. Cengiz Erbaş
ASELSAN ________________

Assoc. Prof. Pınar Şenkul
Computer Engineering Dept., METU ________________

Assoc. Prof. Halit Oğuztüzün
Computer Engineering Dept., METU ________________

Date: ________________

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name:	FATMA GÜLŞAH KANDEMİR
Signature :	

ABSTRACT

A DYNAMIC SOFTWARE CONFIGURATION MANAGEMENT SYSTEM

Kandemir, Fatma Gülşah
M.S., Department of Computer Engineering
Supervisor : Assoc. Prof. Ali Hikmet Doğru
Co-Supervisor : Dr. Cengiz Erbaş

September 2012, 70 pages

Each software project requires a specialized management to handle software development activities throughout the project life cycle successfully and efficiently. Software governance structures provide easy and efficient ways to handle software development activities. Software configuration management is an important software development activity, and while selecting the right strategy in configuration management, its conformity to the software governance should be considered as well. Software configuration management patterns are aligned with the software governance structures to increase the success in development and management of the projects. Companies running large and inter-dependent projects, should adapt their governance structures to the changing characteristics and dependencies of projects. In this thesis, we propose a method to dynamically manage software configuration management, as a result of the changing specifications in the software governance representation.

Keywords: software configuration management, software governance structures, software configuration management patterns, dynamic software configriuation management

ÖZ

DİNAMİK BİR YAZILIM KONFİGÜRASYON YÖNETİMİ

Kandemir, Fatma Gülşah
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi : Doç. Dr. Ali Hikmet Doğru
Ortak Tez Yöneticisi : Dr. Cengiz Erbaş

Eylül 2012, 70 sayfa

Her yazılım projesi projenin yaşam döngüsü süresince yazılım geliştirme aktivitlerinin başarılı ve verimli bir şekilde idare edilmesi için özelleştirilmiş bir yönetime ihtiyaç duyar. Bu amaçla önerilen yazılım yönetimi yapıları yazılım geliştirme aktivitelerini kolaylaştıran etkili yollar sağlar. Yazılım konfigürasyon yönetimi ise önemli bir yazılım geliştirme aktivitesidir ve yazılım konfigürasyon yönetiminde doğru stratejiyi belirlerken yazılım yönetimi yapılarına uyumu da dikkate alınmalıdır. Yazılım projelerinin geliştirme ve yönetim sürecinde-ki başarıyı artırmak amacıyla yazılım konfigürasyon yönetimi örüntüleri yazılım yönetimi yapıları ile eşleştirilmiştir. Büyük ve birbirine bağlı projeler geliştiren şirketlerde, projelerin degişen karakteristiklerine ve bağımlılıklarına gore projelerin yazılım yönetimi yapılarının da zamanla değişmesi gerekir. Bu tez ile, yazılım konfigürasyon yönetimini yazılım yönetimi yapılarındaki değişimlere dinamik bir şekilde uyarlayan bir yöntem öneriyoruz.

Anahtar Kelimeler: yazılım konfigürasyon yönetimi, yazılım yönetimi yapıları, yazılım konfigürasyon yönetimi orüntüleri, dinamik yazılım konfigürasyon yönetimi

To my family

ACKNOWLEDGMENTS

This thesis would not be possible without the guidance and the help of several individuals who in one way or another contributed in the preparation and completion of this study.

First and foremost, I would like to express my greatest appreciations to my supervisor Assoc. Prof. Dr. Ali Hikmet Doğru and my co-supervisor Dr. Cengiz Erbaş, for their encouragement, guidance and valuable advices.

I would like to thank my colleague Nagehan Pala Er for her constant advice and feedback throughout my thesis work.

I am extremely grateful to my parents Bülent and Nurdan Kandemir for their continuous support and love throughout all those years.

I would like to thank my fiancée Cansın for being considerate, for his patience and extreme support throughout all those years. He was always there for me when I was in need.

I would like to thank my friends and colleagues for being wonderful supporters.

In conclusion, I recognize that this thesis would not have been possible without the support of my employer, ASELSAN.

TABLE OF CONTENTS

CHAPTERS

APPENDICES

LIST OF FIGURES

FIGURES

CHAPTER 1

INTRODUCTION

Management of all software projects in the same way is not possible due to the very nature of different characteristics of the projects. And, each project requires a specialized management to handle activities of development successfully and efficiently throughout the project life cycle. Requirements and characteristics of projects are analysed at the beginning of the projects and the analysis process depends on the amount of uncertainty involved in the project. Uncertainty depends on the maturity level in the business area and the technology used [24]. Companies running parallel and inter-dependent projects improve their maturity level in the areas they work and these companies develop their projects with the help of their experience in the field. Depending on both the amount of uncertainty and maturity in projects, software governance structures are proposed and are being used currently in the field of software engineering.

Software governance structures provide easy and efficient ways to handle software transactions in various kinds of development activities. Software configuration management (SCM) is an important software development activity, because it is central to, and provides essential services to, all the major processes of systems and software engineering such as requirements management, design, implementation, integration, verification, release, project management, operation and maintenance. At the beginning of the project, while selecting the right strategy in configuration management, its conformity to the software governance should be considered as well. SCM has a key role during the software development life cycle since implementing wrong or deficient SCM may reduce the efficiency that can be get from all other software development processes like coding, building and releasing the software.

There are 3 primary approaches in software governance structures which are:

- bottom-up
- top-down and
- software reuse

approaches. And there are 3 widely known SCM patterns which are:

- single-line
- main-line and
- producer-consumer

patterns. These software configuration management patterns are aligned with the software governance structures to increase the success in development and management of the projects [23]. The alignment between SCM patterns and governance structures are established to manage software configuration in harmony with the governance structure, to minimize the costs of SCM transactions and to eliminate the difficulties of managing large software.

Large companies usually develop large and inter-dependent projects in parallel. Dependencies between projects and software modules change in time. As a result, governance structures and accordingly implemented SCM patterns for projects should be adoptable to the changes in characteristics and dependencies between these projects. When dependencies between projects change, e.g. a software component is required to be developed by multiple projects, SCM structure of the project should also change to enable shared development of components.

In this work, we propose a method to dynamically manage software configuration management, as a result of the changing specifications in the software governance representation. We developed an SCM pattern reconfiguration tool to adjust the SCM structure of a project to the changing software dependencies. By the aid of this tool we showed that SCM structure of projects should not be just static but should be dynamically adjustable to support development of the software under changing circumstances. Although a meta model for configuration management is also developed, this work is compatible with some of the widely used tools and in particular, we have applied our research utilizing the IBM Rational ClearCase [15] tool.

The rest of this thesis is organized as follows. Chapter 2 gives the necessary background information about the topics discussed in this thesis. Chapter 3 explains how ClearCase tool should be used to implement the SCM patterns discussed. Chapter 4 explains our approach, method and the tool we developed in detail. Lastly, we evaluate and conclude our work in Chapter 5.

CHAPTER 2

BACKGROUND & RELATED WORK

Each software project requires a specialized management for handling software development. For this reason different approaches on software governance structures are proposed according to the different characteristics of projects and dependencies between them. Selecting right governance structure helps handling development activities successfully and efficiently. An important activity in software development is the software configuration management, and while selecting the right strategy in configuration management, its conformity to the software governance should be considered as well. Companies running large and inter-dependent projects, should adapt their governance structures to the changing characteristics and dependencies of projects. This thesis concerns with a method for dynamically managing software configuration management, as a result of the changing software governance structures. In this section, the basics of software development governance structures, software configuration management, software configuration management patterns and the alignment between software configuration management and governance structures are explained briefly to give a better understanding of our proposal.

2.1 Software Development Governance Structures

In business, a transaction is defined as an exchange of products or services between a supplier and a client. Similarly in the context of software, a transaction is an exchange of requirements and software between two stakeholders. Transactions can occur in various forms of governance structures, and the governance structure has a huge impact on transaction costs. Transaction Cost Economics (TCE) analyzes the relationship between transactions and the governance structures extensively [11].

There are three primary software development governance structures that can be proposed after considering the TCE approach [11], which are:

- bottom-up
- top-down and
- software reuse.

Bottom-up governance is adaptive to changes in development decisions and is used when it is not possible to determine or guess the components precisely in the design phase of the project. If the project team do not have enough experience on how to develop some kind of software and mostly the method of trial and error is going to be used during development process, the components and subcomponents determined in design phase may evolve and change throughout the project. In such cases, using the bottom-up governance will be more efficient since it allows changing design decisions[23].

Top-down governance is used when it is possible to decompose and plan the components of the software in the design phase. When the components are decomposed and interfaces between these components are defined properly, the development of these components can be done separately[23]. In other words, if the project team has the experience in developing such software, top-down decomposition will be the most efficient governance structure regarding the transaction costs.

On the other hand, software reuse governance is used when you have software components ready to be used from other projects provided that preconditions/postconditions are satisfied. It is a common practice to use previously developed components when working with multiple projects at a time or one after another. So instead of developing the component needed from scratch, the component is reused. As a result, the reuse governance reduces the transaction costs significantly in projects.

2.2 Software Configuration Management

Software Configuration Management(SCM) is a specialization of Configuration Management (discipline of controlling the evolution of complex systems) in software systems[27]. SCM is one of the most important processes in software engineering since it is central to, and provides essential services to, all the major processes of systems and software engineering such as requirements management, design, implementation, integration, verification, release, project management, operation and maintenance as seen on Fig. 2.1 [2].

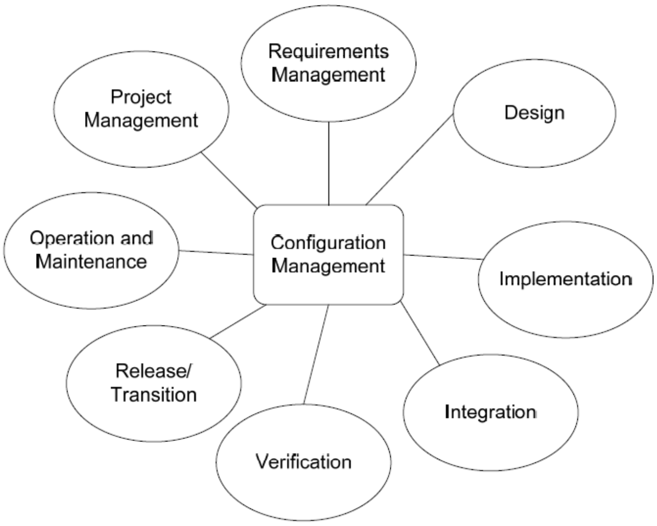


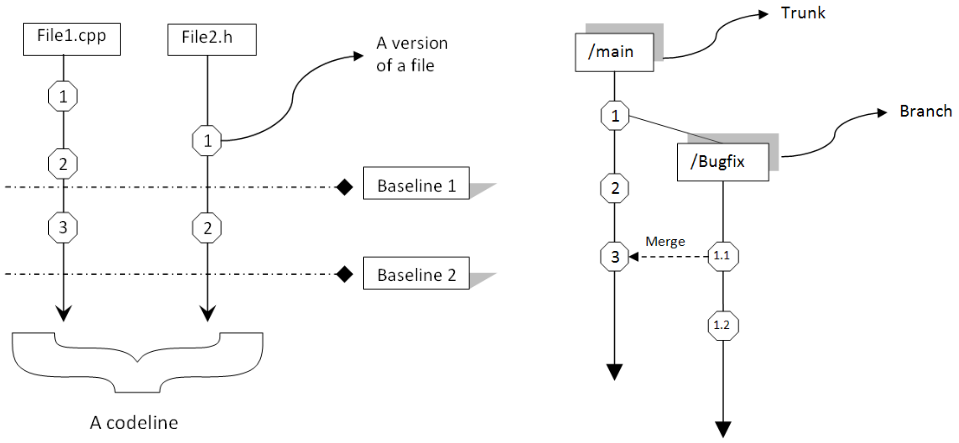
Figure 2.1: Example life cycle processes CM supports, reprinted, with permission, from [2]

The leading activities in SCM are planning of configuration management(CM), management of CM, configuration identification, configuration and change control, status accounting, auditing and release management [2, 9]. The implementation of these activities may change from project-to-project according to the needs of the projects since there is no strictly true implementation. In the beginning of the project how the CM will be implemented should be planned and then throughout the whole software processes, CM should be managed. This is not a trivial task regarding the pervasive nature of the CM in software engineering activities. This thesis concerns itself with the adjusting and managing the usage of CM according to the changing nature and needs of a project or the relationships between projects.

The primary users of an SCM system are developers, since they use the system daily for the implementation of the software. From their perspective, the main concepts that they implicitly deal with are *workspaces, codelines and integration* [5].

A *workspace* can be defined as a working area on a disk where software developers edit the source code, build the software components, and test and debug what they have implemented and built. Changes made in a workspace, will also change the files in the file *repository* eventually since a workspace consists of versions of these files [5, 32]. The main aim in using workspaces is that they provide a private space for developers for their changes to the software and isolation between developers so that they do not interfere with each other [10]. This situation brings the concept of *transparent view* which is a viewing mechanism enabling only authorized access to a workspace from the main repository [10].

Codeline is a bunch of source code files and other software artifacts that when they evolve over time they form the software components. When you change a source code file or another software artifact, a *version* of them is created. And along one semantically defined process path, a codeline includes every versions of files and artifacts [5]. Plus, at some point in time, you may want to *tag* or *label* a snapshot of a codeline, and this is called a *baseline*. Some SCM tools keep these baselines for you to use or observe in the future. And also one or more codelines comprise a workspace, this may change according to the work you are conducting.



(a) A codeline and baselines

(b) A trunk and a branch evolved from the trunk

Figure 2.2: **(a)** A codeline **(b)** A trunk (the main codeline) and a branch evolved from it

While a software component is being developed, you may want a new feature to be developed independently from the main codeline of the component. In other words, you may want to have different derivatives of the component and this is possible with *branching*. Branches help parallel development. Branches may be obsolete in the future, or may be *merged* with the *trunk* (usually the main branch, the starting point of the branches are called trunk) or some other branches [28]. The Figure 2.2(b) illustrates a branch derived from the trunk to fix a bug. The bug fixing process and the development in the main trunk can be achieved in parallel. After deciding that the bug fixing is done in the code, the branch can be merged to the trunk with the help of various merge managers.

Usually in a software project, more than one developer work in parallel. At some points during development, their work should be *integrated* and the sooner integration the better, since frequent integrations prevent complex, hard to resolve merges. Depending on the workspace and branching strategy of a project, number and elapsed time of integrations change, so it is important to choose the right strategy and plan the SCM at the beginning of the project.

2.3 Software Configuration Management Patterns

The success in applying software development governance structures to the software projects resides in proper usage of each and every computer-aided software engineering (CASE) tools. As explained in section 2.2, SCM is central to all software engineering processes, and the two most important functions of SCM are: Integration of codes developed by different developers and keeping track of every change made to the developing or developed software [23]. Hence it is important to apply right SCM strategy to be more effective in development. There have been several research done on best practices and patterns of SCM [5, 23, 32].

The mostly discussed patterns in the past are:

- Single-Line Pattern
- Main-Line Pattern and
- Producer-Consumer Pattern

and will be discussed throughout this thesis because of the previously proposed alignment with the governance structures discussed in section 2.1.

2.3.1 Single-Line Pattern

Single-Line pattern suggests all developers have their own development branches, which are evolved from a single codeline of the project. This enables developers to work isolated from other developers as well as to integrate to the work in the codeline easily. In agile development, it is important to write your code independently without interfering others. Also it is important to sync with the other developers' work frequently. The sooner the integration, the less the conflicts. This is achieved with single-line pattern easily, because in their private branches the developers implement their code, and when required an integration, they deliver their work to the codeline. The process is composed of one step at a time rebase-implement-deliver-integrate iteration. Figure 2.3 illustrates the pattern by showing the */main* as the codeline and */Dev1* and */Dev2* as the branches of different developers.

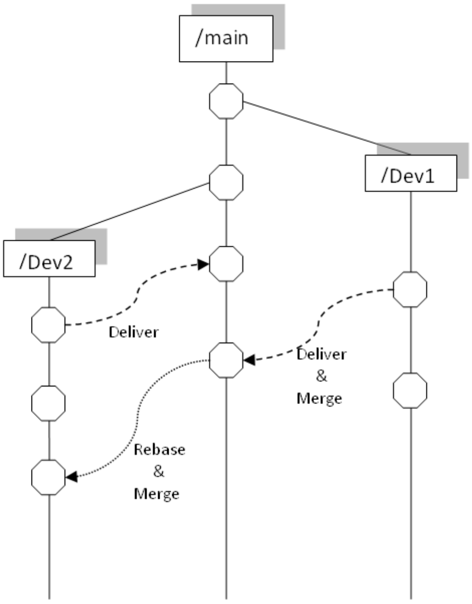


Figure 2.3: Codeline of a Single-Line Pattern

2.3.2 Main-Line Pattern

Mainline pattern is useful when you develop shared components between projects in parallel, in other words when you have multiple codelines developing one or more component and you want to handle the deliver-merge-rebase operations effectively. Parallel development between multiple projects can be handled with multiple branches and hence with merging, but this will have a cost which can easily be minimized when a simplified branching model is used carefully and mainline pattern helps simplifying your branching model [5].

Mainline stands for a central codeline which is the basis for sub-branches and their resultant merges, says Vance, and the mainline is the one with the longest lifespan of all other codelines [29]. This is because the mainline branch continues to exist while a shared component is being developed by at least one project and until the last project is done with development.

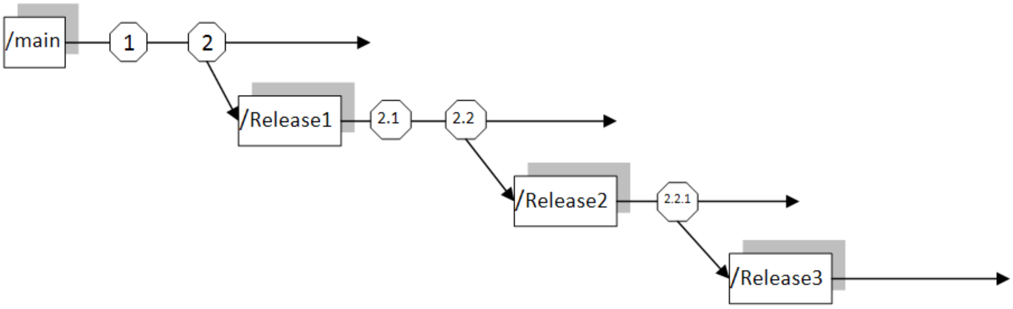


Figure 2.4: Cascading Projects(Releases)

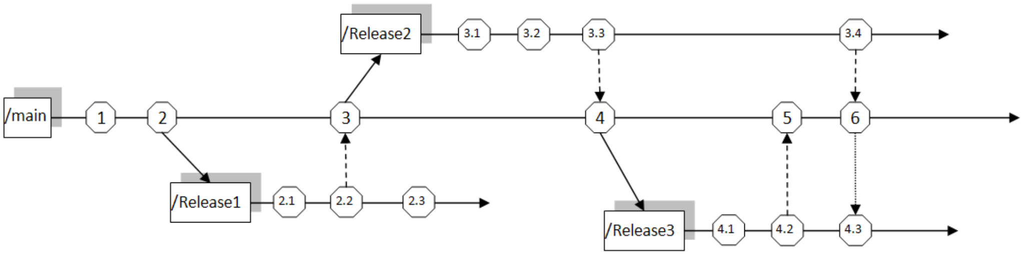


Figure 2.5: Main-Line Pattern

Figure 2.4 shows a branching model when a project starts development based on a release of the previous project and they continue development in parallel. For each release a branch is evolved from the previous release, this is called cascade branching. But think of an additional feature which is wanted in all of the releases; in this case each project will implement that additional feature or, it will be implemented in one of the release branches and then will be merged to all of the other release branches. The first workaround has the problem of rework of the implementation and the second brings the cost of complex merge. It would be a better choice to use a mainline project at the beginning. Figure 2.5 shows a mainline project denoted by /Main and 3 different projects denoted by /Release1, /Release2 and /Release3. All of the projects merge their changes with the mainline project and whenever a new project is about to start, instead of branching from a previously released software, they branch from the mainline, which includes the latest and most stable version of the software. The mainline eliminates the problem of costly complex merges, as merging in this case requires only one step of code delivery.

2.3.3 Producer-Consumer Pattern

When having shared software components between projects, and only one project is responsible for the development of that component, then producer-consumer pattern should be applied. As name suggests, there is a producer project, developing (creating versions) the component, and consumer projects just add the component baselined by the producer project to their configuration as read-only.

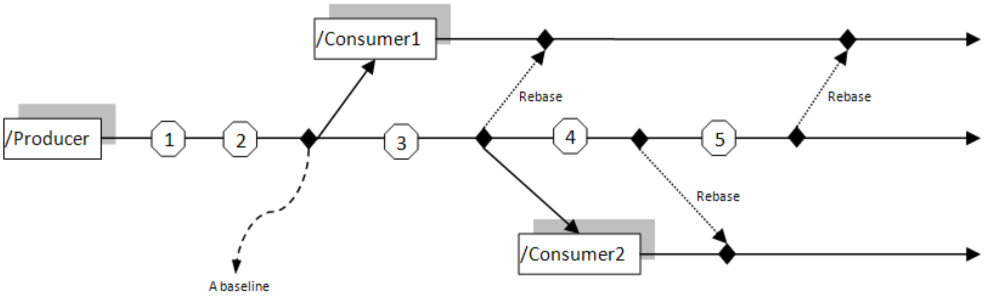


Figure 2.6: Producer-Consumer Pattern

Figure 2.6 shows a multi-project SCM implementation, which is composed of one *Producer* and two *Consumer* projects. The producer project is the one producing versions of component, and consumer projects are the ones only reading the versions created by producer project by rebasing to the baselines (labels) of the component.

2.3.4 Alignment between Governance Structures and SCM Patterns

It is important but also difficult to determine the right structure you need in configuration management, because if you underestimate, then the chaos happens, and if you overestimate the tools and the environment will be useless at some point. According to Highsmith[13] this debate is really about balancing adaptation with anticipation. Software development governance structures surely give the configuration managers the anticipation in deciding how their SCM structures should be established.

In [24, 23], an alignment between software governance structures and SCM patterns is proposed. Their rationale on mapping governance structures and SCM patterns depends on the TCE approach. They considered and calculated two different types of costs (1) the setup cost of SCM infrastructure and (2) the cost of maintenance and integration of SCM structure during development. The calculations showed that:

- When software components cannot be identified at the beginning of the project, *Bottom-Up* governance was recommended, and since components are not identified, the sharing status of components are also not available. So, recommending one integration codeline for development, the *Single-Line* pattern would be the most effective pattern among others.
- When software components can be identified at the beginning of one project and these components are the potential parallel development components, then *Top-Down* decomposition was recommended. So, having a *Mainline* structure would fully support the parallel development of identified shared components.
- When some components of the project can be taken readily from another projects configuration, and these components will not be developed by the new project, then software *Reuse* was recommended. Now the new project is a *Consumer* project and the project doing the development is the *Producer* project in SCM.

2.4 Software Configuration Management Tools and Implementation

SCM, as mentioned above, deals with controlling change and the activities of other software development processes in large and complex software systems. As the importance of SCM is increased, the need to handle the SCM activities with the help of SCM tools had arisen. Commercial or free, various kinds of SCM software are released since then. Early tools had only limited functionality but the modern tools serve very advanced functionalities, so it is now more effective to manage the configuration of software products [12]. More about researches conducted on SCM processes both in academia and market can be found in [12].

A successful SCM system, according to [22], manages the changes to the software product, keeps track of versions and component configurations, coordinates the work of team members, and manages building and releasing the software product deliverables. And it is very important for SCM systems to be flexible and adoptable to the changes in the projects and organization of the project management. The key functionalities of SCM tools are, as suggested by both [1, 22]:

- support for SCM library(repository)
- version control
- change management
- status accounting
- build and release management

These functionalities are now the default requirements of SCM tools, but they may not be enough for some companies in the market. This is due to the fact that companies developing more than one software products simultaneously, need parallel and concurrent development support, which also brings the diversity and merge handling between codelines. In addition to parallel development of software projects, shared component development/usage between projects is getting more common as software reuse is one of the most effective way of building high-quality software in limited time [22]. The ideal SCM system for a company building complex, reusable shared components should support all the software development activities including accessing, versioning, controlling, building a releasing the components of software products.

Companies must be careful and know their needs and expectations from an SCM system, before choosing the right product for them. Because, SCM system is just at the centre of all software development activities, and directly affects the development environment in a company.

Since the problem that we are trying to propose a solution in this thesis, addresses the identification, development, sharing and re-using of software components between projects working in parallel, we should have chosen the right product which would enable us to streamline development of component-based software projects. The most suitable SCM tool for this purpose was the commercial IBM Rational ClearCase [15] tool. We implemented dynamic software configuration management using IBM Rational ClearCase with UCM (Unified Change Management) enabled.

Implementation of the SCM patterns which are mentioned in section 2.3 is explained in next chapter to make understanding of our *Dynamic Software Configuration Management System* proposal easier. In addition, ClearCase terminology that will be used throughout this thesis while explaining our system can be found in Appendix A.

2.5 Related Work

Software Development Governance(SDG) is a new field of research and it is evolving as an important area of research because governance of software projects has a huge impact on the other development processes. Bannerman[4] made a good research on SDG from different aspects. According to his work in [4], SDG emerges as both an opportunity and a challenge in development activities. For example, SDG should concern about aligning business with software, controlling risk and change and providing flexibility [8]. SDG should also assist agile software development in adaptive organizations [26]. Cheng and et al.'s [7] definition for agile SDG is ''the accountability and responsibility of management, adopting agile software development methods, and establishing measurement and control mechanisms in an agile environment''. In addition, [31] evaluates SDG as both a technical creativity and technical challenge.

The importance of alignment between SDG and other software development processes is another field of research. The first step of achieving strategic accordance with other software development processes is considered to be an effective team-work. By establishing chains of roles, responsibilities and communication, SDG achieves its strategic goals [8, 20]. Lehto et al. propose a framework for their agile SDG structures and highlight the challenges in roles and responsibilities [20]. In addition Kofman et al. address this challenge and develop a tool called *Governer* to automate the decision right issues [18]. Kofman et al. also state that smart governance tools should guide and automate software development processes, for their specific purpose decision making of individual through software development activities [18].

On the other hand, in this thesis we concern with alignment of SDG with SCM activities and automate the adaptation of these two activities with a tool we developed.

Software Configuration Management(SCM) is an important organizing software development activity as discussed in section 2.2, and several research has been made on the appropriate usage of SCM in organizations. Configuration management's relation to other software fields such as software architecture is the concern of [3] and this work suggests these areas should not be separated. Similarly, for the component-based software architectures a configuration management approach is suggested in [19, 21]. Larsson has a comprehensive work on component-based software development and especially component-based configuration management techniques. Additionally, since applying right strategy in SCM is very important, several best-practices and patterns are developed and suggested [5, 23, 32, 30]. In this thesis, we looked over 3 different SCM patterns as discussed in section 2.3.

Dynamism of software architectures, especially dynamic components are considered as "challenging in terms of correctness, robustness and efficiency" [25, 6] and dynamic software architectures are investigated by Bradbury et al. in [6] extensively. Considering the importance of dynamism in other fields of software development, we have developed a dynamic approach to software configuration management, which has not been an area of research before in terms of applying dynamism in SCM patterns. While applying SCM patterns, we follow the implemented SDG approach of a software project and we believe our dynamic SCM proposal will find its place in the literature as a novel and a strong idea.

CHAPTER 3

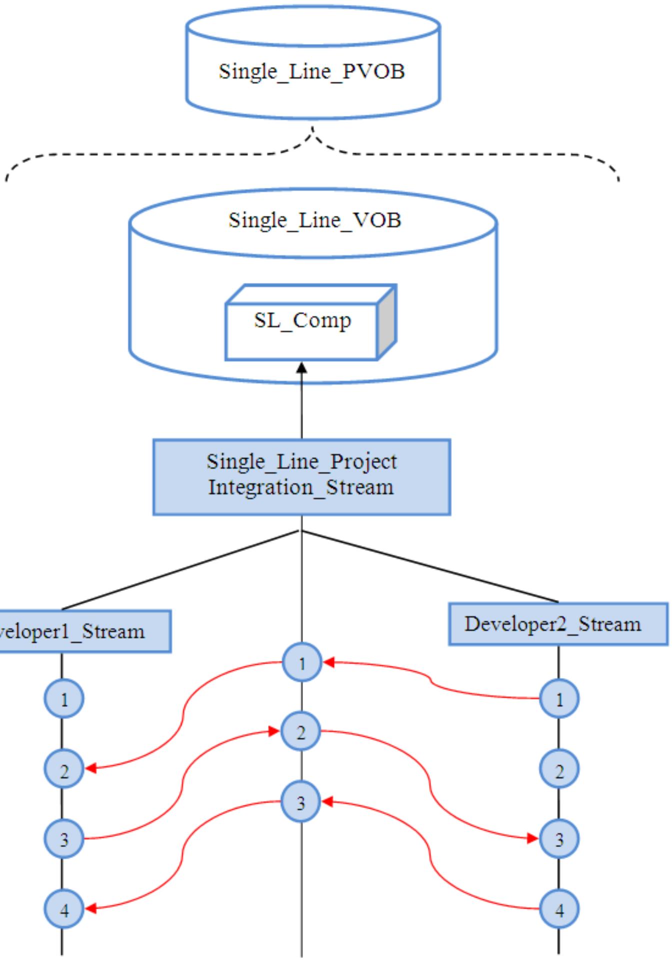
IMPLEMENTATION OF SCM PATTERNS IN CLEARCASE

In Section 2.3, three SCM patterns were emphasized and these patterns and their usage with shared component development will be the primary concern of this thesis. So, to give a better understanding on these patterns, their implementation in IBM Rational ClearCase tool for supporting shared component development will be explained in this section.

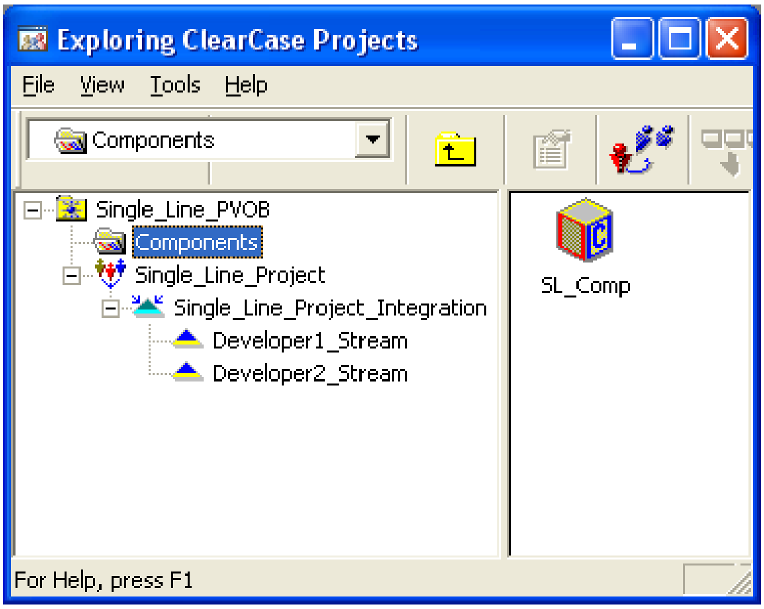
3.1 Single-Line Pattern

Implementation of Single-Line pattern in ClearCase is quite simple as parallel development of *components* is out of concern in this pattern. As explained in section 2.3.1, a single codeline is required for code integration when each developer has his/her own private branch to develop the software. In ClearCase, the codeline corresponds to a *project* with an *integration stream* and developers' private branches correspond to the development streams. Developers deliver their work from their development streams to the integration stream.

The ideal and the easiest implementation of Single-Line pattern would consist of one UCM project. The components would reside in one *VOB* and meta-data of all UCM elements(files elements, folder elements, streams and project) would be held by one *PVOB(Project VOB)*. Figure 3.1(a) illustrates the general ClearCase implementation of Single-Line pattern and Figure 3.1(b) shows the structure of a Single-Line PVOB in *ClearCase Project Explorer* tool. According to the 3.1, *SL_Comp* is developed by only *Single_Line_Project*, thus there exists only one integration stream. Depending on the number of developers, development streams (each development stream is a child stream of integration stream) are created. In this case, development streams are *Developer1_Stream* and *Developer2_Stream*. Each developer *deliver*



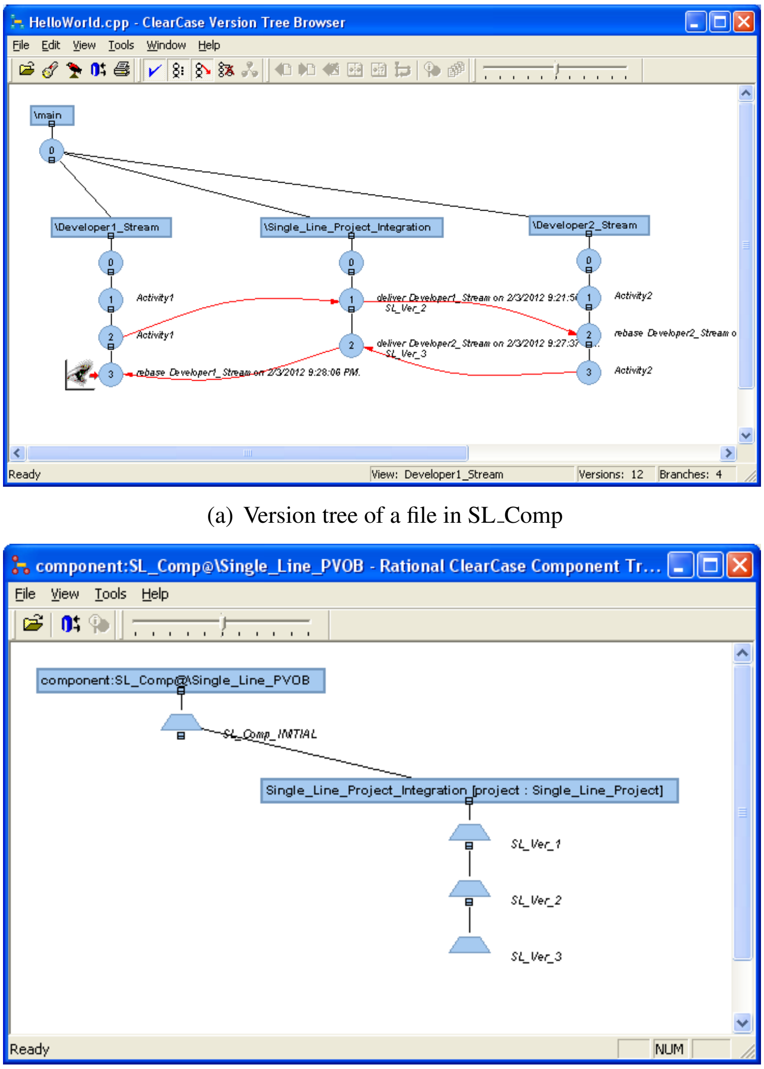
(a) Implementation of Single-Line pattern in ClearCase



(b) Structure of a Single-Line PVOB in ClearCase Project Explorer

Figure 3.1: **(a)**General implementation of Single-Line pattern in ClearCase and **(b)** Structure of a Single-Line PVOB in ClearCase Project Explorer tool

from development to integration stream, or *rebase* from integration to development stream. *Merges* are done in the integration stream if more than one developer changed the component and delivered. The red solid arrows originating from development streams in 3.1(a) correspond to the deliver activities and the other ones originating from integration stream corresponds to the rebase activities. The deliver and rebase operations can be understood from Figure 3.2(a) which shows the version tree of a file which is in SL_Comp. And Figure 3.2(b) shows the baseline tree of the SL_Comp. As seen on the baseline tree, baselines are only established in integration stream of Single_Line_Project, because SL_Comp is developed by only that project.


HelloWorld.cpp - ClearCase Version Tree Browser
File Edit View Tools Window Help
\main
0
\Developer1_Stream
\Single_Line_Project_Integration
\Developer2_Stream
0
1
Activity1
2
Activity1
3
rebase Developer1_Stream on 2/3/2012 9:28:06 PM.
0
1
deliver Developer1_Stream on 2/3/2012 9:21:5
SL_Ver_2
2
deliver Developer2_Stream on 2/3/2012 9:27:37
SL_Ver_3
0
1
Activity2
2
rebase Developer2_Stream o
3
Activity2
Ready
View: Developer1_Stream
Versions: 12
Branches: 4

(a) Version tree of a file in SL_Comp

component:SL_Comp@\Single_Line_PVOB - Rational ClearCase Component Tr...
File View Tools Help
component:SL_Comp@\Single_Line_PVOB
SL_Comp_INITIAL
Single_Line_Project_Integration [project : Single_Line_Project]
SL_Ver_1
SL_Ver_2
SL_Ver_3
Ready
NUM

(b) Baseline tree of SL_Comp

Figure 3.2: **(a)** A file's version tree which is in SL_Comp and **(b)** SL_Comp's baseline tree

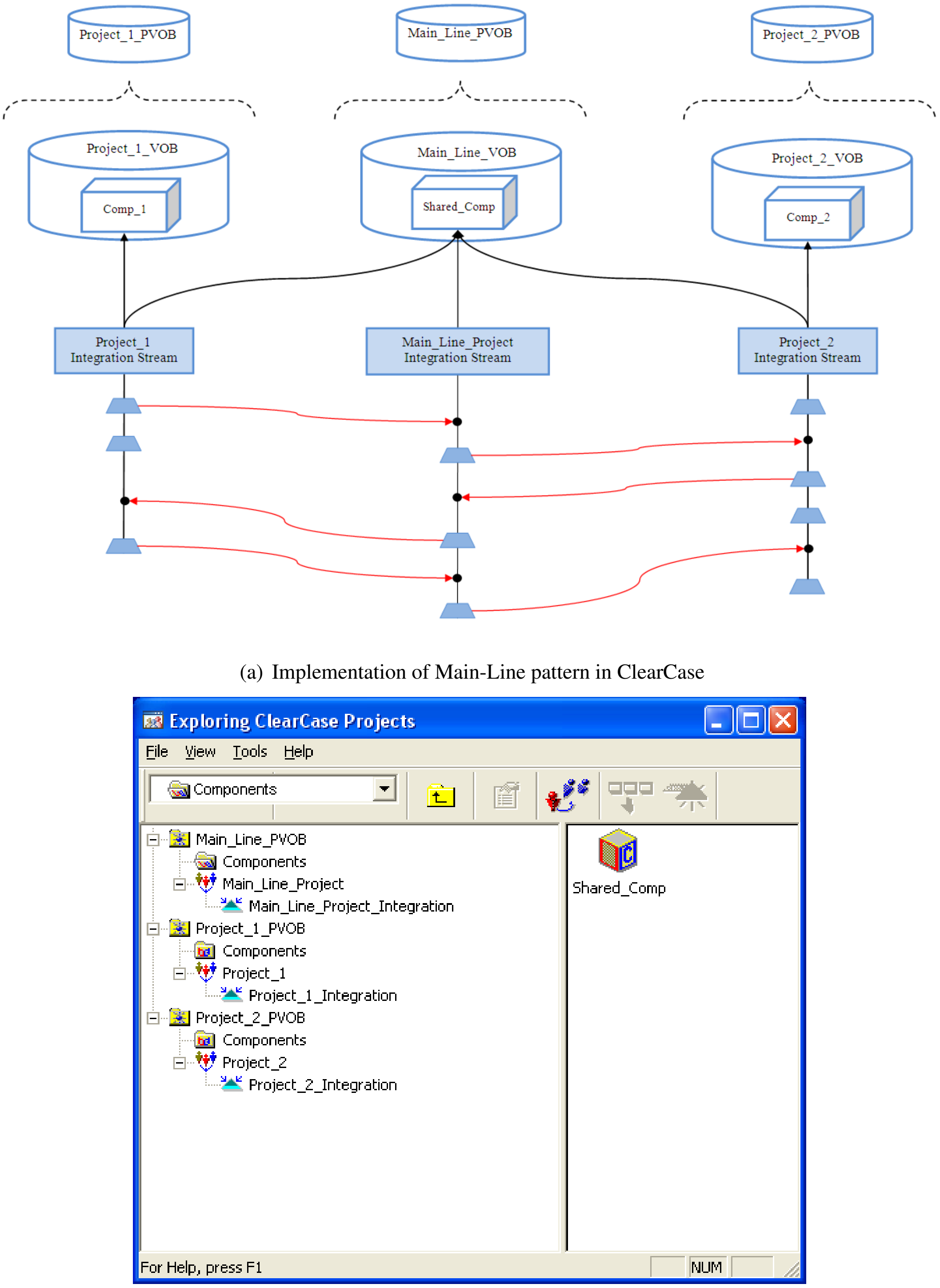
3.2 Main-Line Pattern

As explained in Section 2.3.2, mainline stands for a central codeline which is the basis for sub-branches and their resultant merges and in ClearCase, mainline is implemented as a ClearCase project and integration stream which serves as a merge centre to the *follow-on* projects (sub-branches). Thus, implementation of Main-Line pattern requires at least one shared component and a main-line project associated with it. A follow-on project can be created whenever a new project wants to develop the shared component.

Figure 3.3(a) illustrates a shared component development using Main-Line pattern. *Main_Line_PVOB* is the project VOB of *Main_Line_VOB*, *Main_Line_Project* and *Shared_Comp* and holds meta-data of all of them. *Main_Line_VOB* is the VOB holding shared components and in this case, it holds the *Shared_Comp* (i.e. main-line component) which is the component to be developed in parallel with other projects. *Main_Line_Project Integration Stream* corresponds to the main codeline. Follow-on projects which develop the Shared_Comp in parallel are shown as *Project_1* and *Project_2* and the solid arrows originating from their integration streams point to the components they are developing. It is typical for projects of a company to develop their own components and shared components, thus an ideal SCM system should provide the developers the easiest environment to do the development. In this case, the follow-on projects can easily develop&integrate their unshared components, *Comp_1* and *Comp_2*, in their sub-branches. Comp_1 and Comp_2 reside in the VOBs of follow-on projects and PVOBs hold the meta-data of these projects. To give a better understanding of Main-Line pattern implementation in ClearCase, Figure 3.3(b) shows the structure of Main_Line_PVOB, Project_1_PVOB and Project_2_PVOB in ClearCase Project Explorer tool.

In order to Project_1 and Project_2 edit the Shared_Comp, Shared_Comp should be added to the *configuration* of the projects with read&write permission. When one of the developing projects add a feature to the main-line component that other developing projects should get or the main-line component has been come to a maturity level by one of the projects, the changes should be delivered to the main-line project's integration stream. After the changes are merged and conflicts are resolved (if exists any) in the main-line integration stream, a baseline should be established to serve other projects as a basis to rebase.

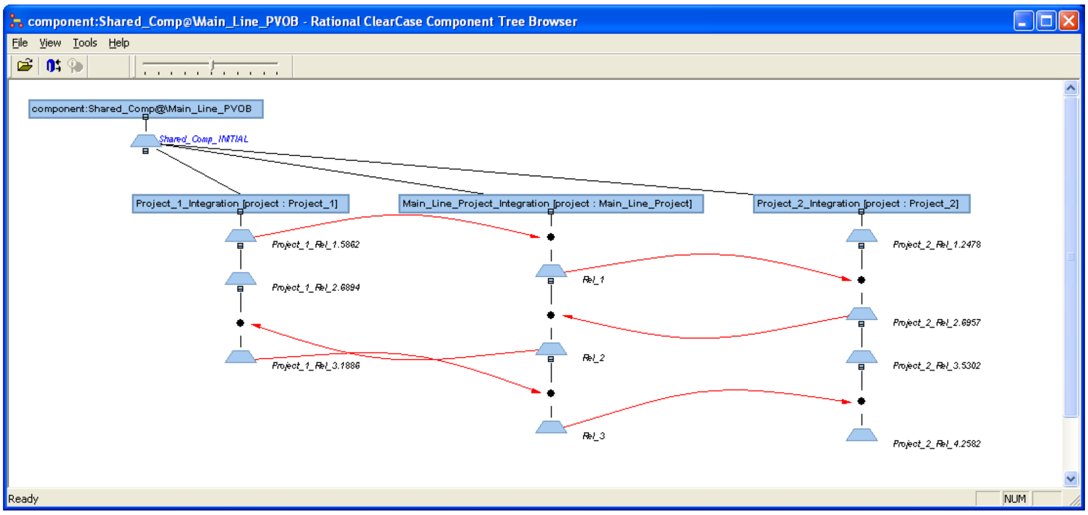
Figure 3.4 shows the shared(main-line) component's the ClearCase baseline tree window



(a) Implementation of Main-Line pattern in ClearCase

(b) Structure of a Main-Line PVOB and Follow-on PVOBs in ClearCase Project Explorer

Figure 3.3: **(a)** General implementation of Main-Line pattern in ClearCase and **(b)** Structure of a main-line PVOB and follow-on PVOBs in ClearCase Project Explorer tool



component:Shared_Comp@Main_Line_PVOB - Rational ClearCase Component Tree Browser
File View Tools Help
component:Shared_Comp@Main_Line_PVOB
Shared_Comp_INITIAL
Project_1_Integration [project : Project_1]
Main_Line_Project_Integration [project : Main_Line_Project]
Project_2_Integration [project : Project_2]
Project_1_Rel_1.5862
Project_1_Rel_2.6894
Project_1_Rel_3.1886
Rel_1
Rel_2
Rel_3
Project_2_Rel_1.2478
Project_2_Rel_2.6957
Project_2_Rel_3.5302
Project_2_Rel_4.2582
Ready
NUM

Figure 3.4: Baseline tree of a Main-Line component (Shared_Comp)

and as can be seen from Figure 3.4, Project_1 and Project_2 deliver their changes to the integration stream of Main_Line_Project and rebase to the baselines taken in the integration stream of Main_Line_Project. For example, first Project_1 delivers its changes to the Main_Line_Project and a baseline named *Rel_1* is established. When Project_2 wants to get changes in Rel_1, Project_2 rebases to that baseline of Main_Line_Project. After Project_2 works on the Shared_Comp and delivers its changes to Main_Line_Project, Project_1 wants to get those changes and rebases to the version *Rel_2* of Shared_Comp. In Figure 3.4, rebases are represented as red arrows originating from Main_Line_Project to follow-on projects, and delivers are again represented as red arrows but originating from follow-on projects to Main_Line_Project.

3.3 Producer-Consumer Pattern

Producer-Consumer pattern, as explained in Section 2.3.3, is applied if a component is developed by only one project and reused by one or more projects. Implementation of Producer-Consumer pattern in ClearCase is quite similar to the Main-Line pattern's implementation as shown in Figure 3.6(a), except shared component can only be accessed by consumer projects as *read-only*. Solid lines originating from integration streams to the components in Figure3.6(a) indicate that components can be modified by the streams and dashed lines indicate that components can only be accessed as read-only, in other words they are reusable components.

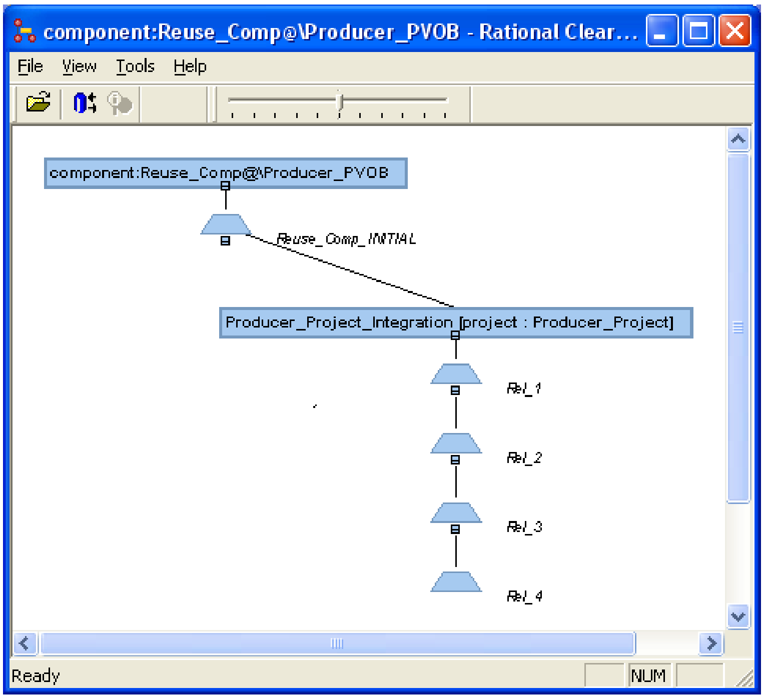
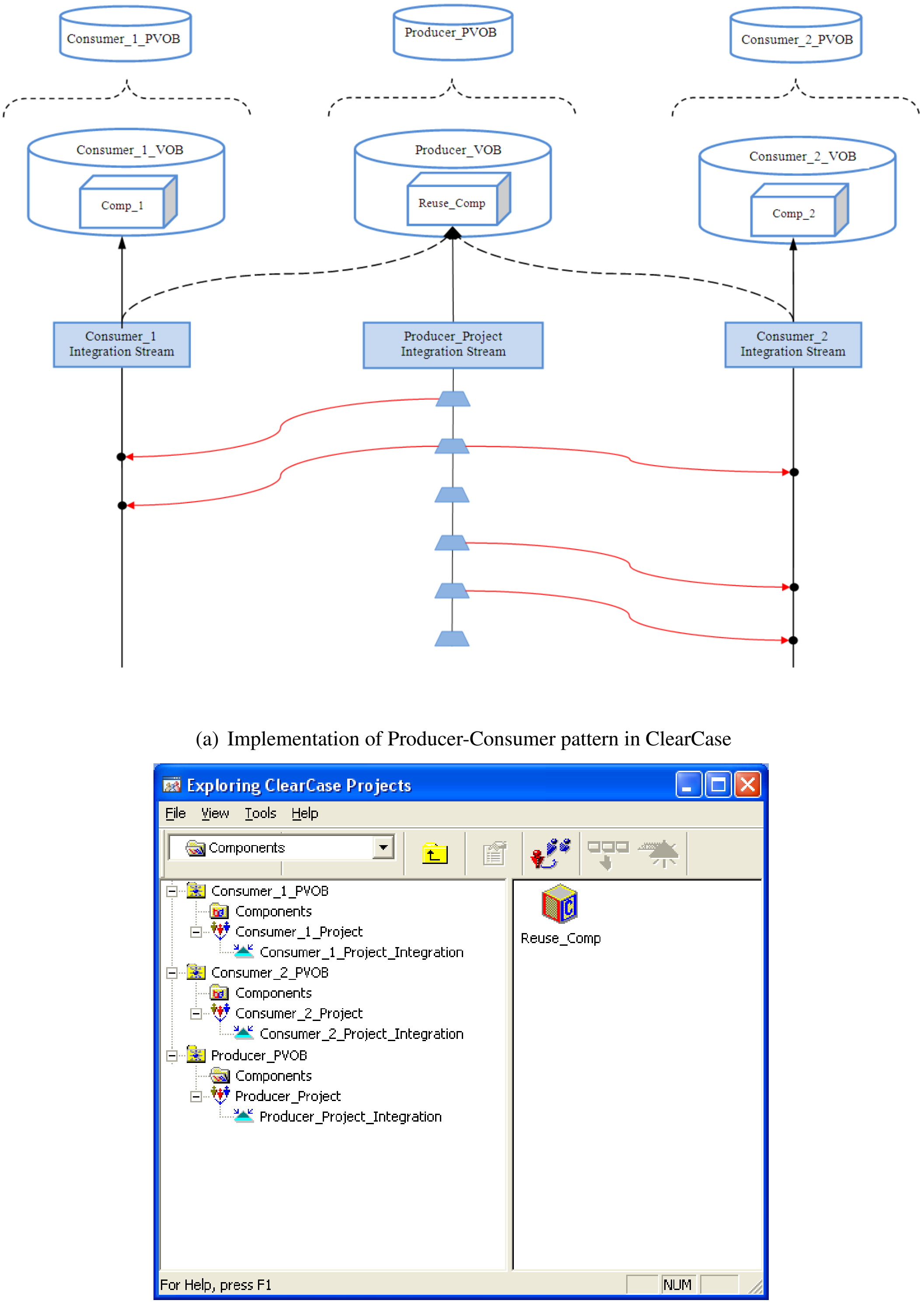

component:Reuse_Comp@\Producer_PVOB - Rational Clear...
File View Tools Help
component:Reuse_Comp@\Producer_PVOB
Reuse_Comp_INITIAL
Producer_Project_Integration [project : Producer_Project]
Rel_1
Rel_2
Rel_3
Rel_4
Ready
NUM

Figure 3.5: Baseline tree of a reuse component

Figure 3.6(b) shows the structure of Producer_PVOB, Consumer_1_PVOB and Consumer_2_PVOB in ClearCase Project Explorer tool. *Reuse_Comp* resides in *Producer_VOB* and can only be changed by Producer_Project. All of the other projects which want to access and use the Reuse_Comp should add the Reuse_Comp to their configuration as read-only.

Whenever a producer project wants to add a new feature and release a new version of the reusable component, takes a baseline indicating the new release and recommends it to the consumer projects. If a consumer project wants the new feature implemented, that will be enough to rebase to the recommended baseline. Figure 3.5, shows the baseline tree of previously mentioned component Reuse_Comp, and as can be seen from the tree, only Producer_Project could develop the component and produced baselines *Rel_1, Rel_2, Rel_3* and *Rel_4*. Any project can rebase to these baselines to access to the component.



(a) Implementation of Producer-Consumer pattern in ClearCase

(b) Structure of a Producer PVOB and Consumer PVOBs in ClearCase Project Explorer

Figure 3.6: **(a)** General implementation of Producer-Consumer pattern in ClearCase and **(b)** Structure of a Producer PVOB and Consumer PVOBs in ClearCase Project Explorer tool

CHAPTER 4

DYNAMIC SOFTWARE CONFIGURATION MANAGEMENT SYSTEM

Identification and usage of software components, direct the way the software projects are governed as explained previously in section 2.1. Even identified at the beginning of the projects, component structures may change throughout the development phase. And the primary reason for changing component structures is the development itself. To be more specific, some of the cases causing the governance structure to change are:

- sub-components may form new components during development
- parallel development of components between new or existing projects may be required
- components may wanted to be reused

A change in component structures on account of the previous reasons, requires a change in governance structures naturally. Component structures change because of the amount of uncertainty dominating the project is decreased. The amount of uncertainty change the you way a project is governed, and as a result, from the SCM point of view, SCM implementation should adopt to the changes in governance structures.

The first case, sub-components of a project evolving into new components during development, will yield a project governance structure to change from bottom-up to top-down approach. The transition is same for the second case as well. In section 2.3.4, alignment between governance structures and SCM patterns were discussed and, bottom-up governance was aligned with Single-Line pattern and top-down governance was with Main-Line pattern.

The third case, starting to use/read a component developed by only one producer project, will yield a project governance structure change to software reuse. Reconfiguring to reuse governance may originate from bottom-up governance or top-down governance. This time, in accordance with alignments between governance structures and SCM patterns, reconfiguration of SCM structures will be from Single-Line to Producer-Consumer pattern, or from Main-Line to Producer-Consumer pattern.

It is plain that, the decrease in uncertainty yields to improvement in development and it is the natural outcome of whole software development process. From the software governance point of view, as the uncertainty is decreased in time, i.e. level of experience is increased, bottom-up governance is replaced by top-down governance or top-down governance is replaced by reuse governance. The opposite direction of replacement is not possible since level of experience never decreases in time. Thus, change in software governance structures can only originate from bottom-up or top-down structure.

In this thesis, we propose to adopt SCM patterns to governance structures automatically. In other words, we want to automate reconfiguration of SCM structure from one pattern to another [17]. SCM structures of software projects are usually implemented using an SCM tool. Changing the SCM structure may sound simple at first, but from the SCM administrators' point of view, it means changing the whole infrastructure of SCM environment and requires a big amount of time to do the reconfiguration manually. Automatically doing the reconfigurations between patterns is the most effective way of doing this job, because it is much less time consuming and error-prone. In addition, there exists no work in the literature about dynamically adjusting SCM structure.

In the upcoming subsections, after a brief overview of the system, 3 modes of the DSCM system will be explained. By 3 modes of DSCM, we mean 3 possible ways of automatically reconfiguring SCM patterns as explained in previous paragraphs.

4.1 Overview of the Dynamic SCM System

Component structures determine the SCM pattern to be implemented, thus SCM process starts the moment components are identified and the project starts. Dynamic software configuration management (DSCM) process also starts when the project starts. But it is the changing component structures that initiates the main functionalities of DSCM, the recommendation of new SCM pattern and updating the current configuration to reflect the change in the SCM pattern.

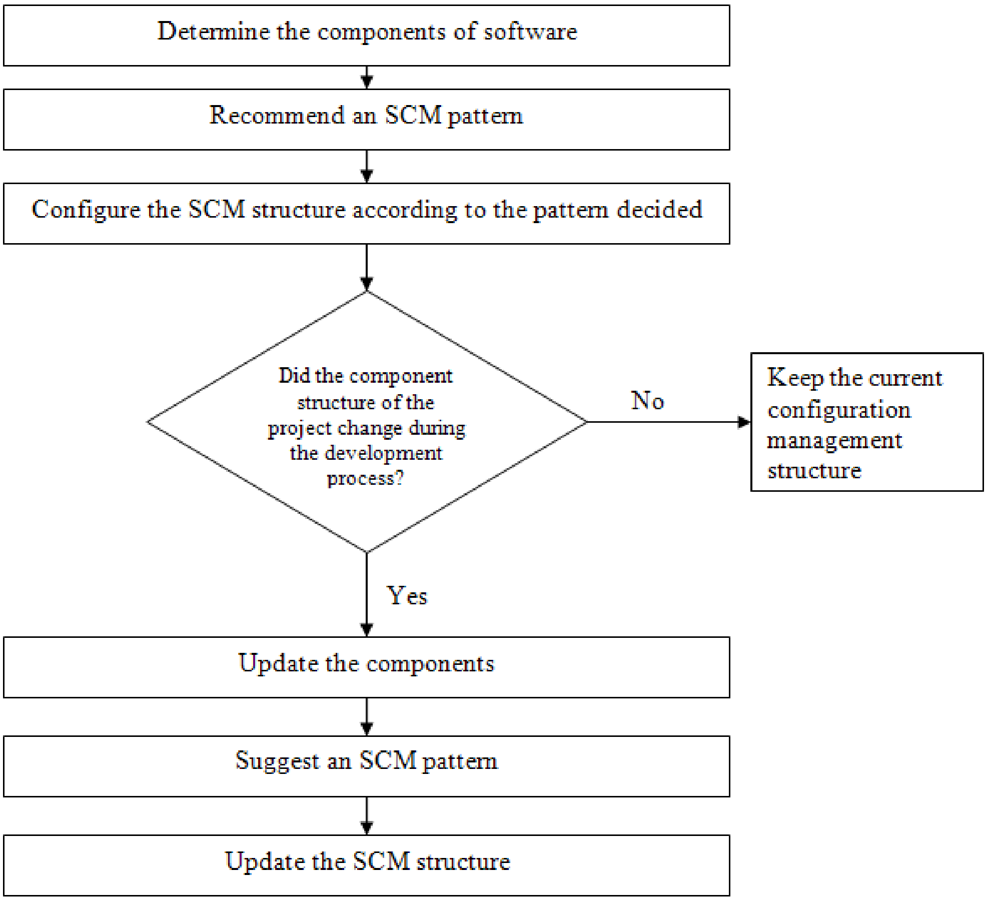
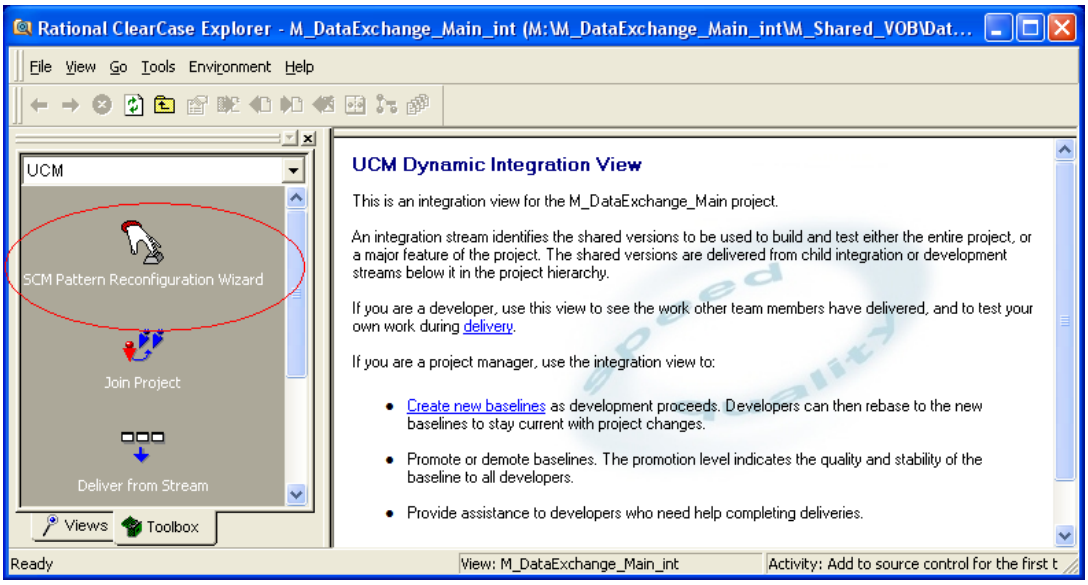

Determine the components of software
Recommend an SCM pattern
Configure the SCM structure according to the pattern decided
Did the component structure of the project change during the development process?
No
Keep the current configuration management structure
Yes
Update the components
Suggest an SCM pattern
Update the SCM structure

Figure 4.1: Life cycle of a dynamic SCM system

Figure 4.1 summarizes the basic life cycle of a dynamic SCM system. At the beginning of the projects, the components are determined, an SCM pattern is recommended according to the components and then the SCM infrastructure is built. During the development, if the component structures change, components will be updated, a new SCM pattern will be recommended and then the recommended SCM pattern will be implemented. The tool we designed, will be responsible for updating SCM structure automatically.

We designed the automatic pattern reconfiguration part of DSCM as a wizard integrated to IBM Rational ClearCase[15] environment. The wizard is attached to the ClearCase Explorer, so that the users of the system can access and start the wizard easily. Figure 4.2 is a snapshot of ClearCase Explorer window integrated with DSCM wizard. DSCM wizard opens when *SCM Pattern Reconfiguration Wizard* is clicked.


Rational ClearCase Explorer - M_DataExchange_Main_int (M:\M_DataExchange_Main_int\M_Shared_VOB\Dat...
File View Go Tools Environment Help
UCM
SCM Pattern Reconfiguration Wizard
Join Project
Deliver from Stream
Views
Toolbox
Ready
View: M_DataExchange_Main_int
Activity: Add to source control for the first t

UCM Dynamic Integration View

This is an integration view for the M_DataExchange_Main project.

An integration stream identifies the shared versions to be used to build and test either the entire project, or a major feature of the project. The shared versions are delivered from child integration or development streams below it in the project hierarchy.

If you are a developer, use this view to see the work other team members have delivered, and to test your own work during delivery.

If you are a project manager, use the integration view to:

- Create new baselines as development proceeds. Developers can then rebase to the new baselines to stay current with project changes.
- Promote or demote baselines. The promotion level indicates the quality and stability of the baseline to all developers.
- Provide assistance to developers who need help completing deliveries.

Figure 4.2: Integration of DSCM wizard with IBM Rational ClearCase Explorer

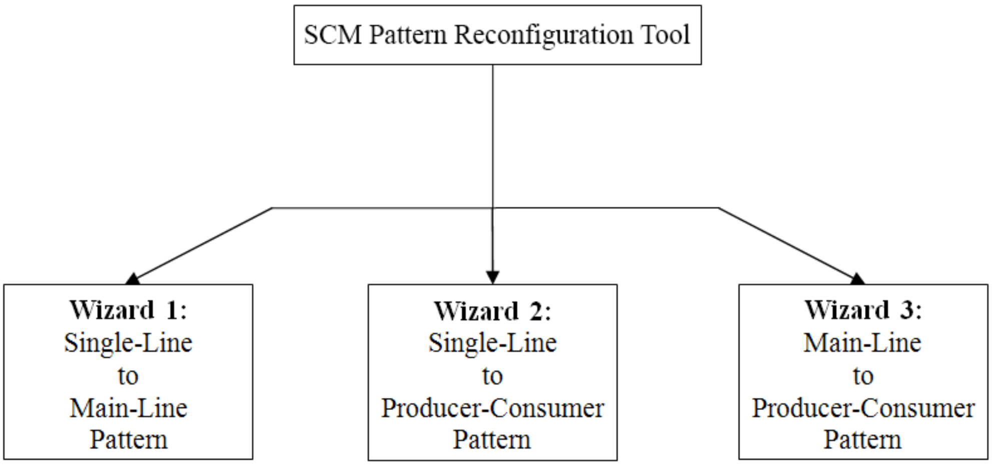

SCM Pattern Reconfiguration Tool
Wizard 1: Single-Line to Main-Line Pattern
Wizard 2: Single-Line to Producer-Consumer Pattern
Wizard 3: Main-Line to Producer-Consumer Pattern

Figure 4.3: The 3 wizards designed to update the SCM structure

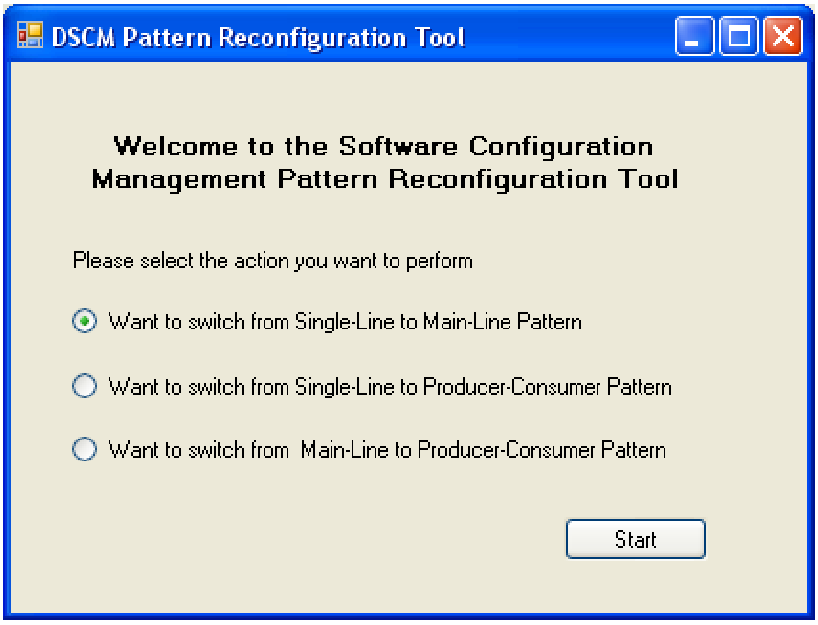
DSCM reconfiguration tool we designed is composed of three wizards which are shown in Figure 4.3:

- Single-Line to Main-Line pattern reconfiguration wizard

- Single-Line to Producer-Consumer pattern reconfiguration wizard

- Main-Line to Producer-Consumer pattern reconfiguration wizard

Since the recommended pattern is known prior to adjust the SCM structure to the new SCM pattern, the user will select the type of the pattern reconfiguring wizard. DSCM pattern reconfiguring tool has a main page composed of three wizard types to enable the users to select the right type of wizard they will be guided. The main(starting) page of the DSCM pattern reconfiguring tool is shown in Figure 4.4.



Welcome to the Software Configuration
Management Pattern Reconfiguration Tool

Please select the action you want to perform

● Want to switch from Single-Line to Main-Line Pattern

○ Want to switch from Single-Line to Producer-Consumer Pattern

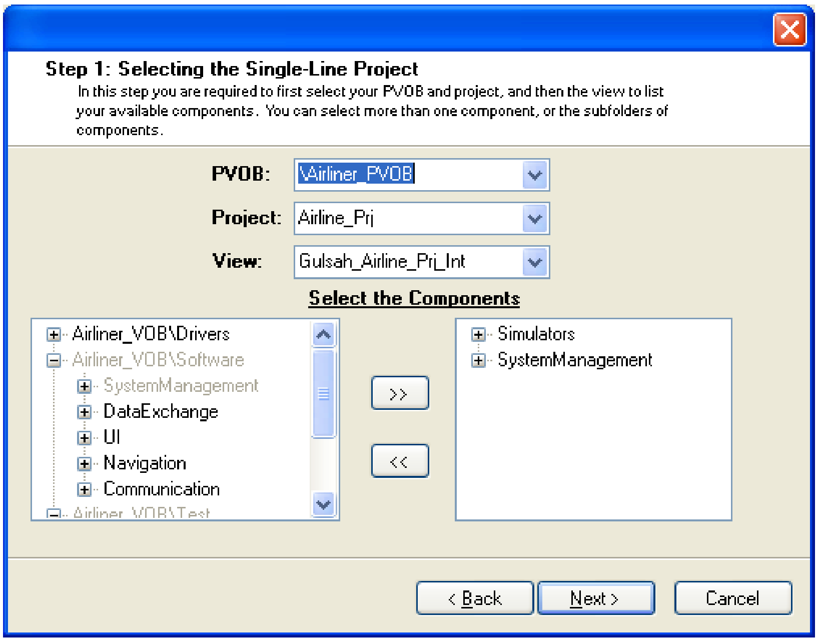
○ Want to switch from Main-Line to Producer-Consumer Pattern

Figure 4.4: The starting page of the DSCM Pattern Reconfiguration Tool

Upon selecting the action to be performed as shown in Figure 4.4, one of the wizards will be started. The users will be guided by these wizards and the new SCM structure will be ready to used when the wizards are completed. In next three subsections, the three reconfiguration wizards will be explained by first showing the user interface elements, then explaining the process carried out in the background and finally showing the resulting SCM structure.

4.2 Reconfiguring from Single-Line to Main-Line Pattern

As can be understood from Figure 4.1, the inputs of reconfiguring from Single-Line to Main-Line pattern wizard are the updated software components. The former single-line developed components or the sub-components of components are going to be transformed into main-line components at the end of the SCM structure update process. Once the to-be main-line components are identified, the SCM structure of these components based on the Single-Line pattern is ready to be transformed to an SCM structure based on Main-Line pattern.



Step 1: Selecting the Single-Line Project

In this step you are required to first select your PVOB and project, and then the view to list your available components. You can select more than one component, or the subfolders of components.

PVOB: \Airliner_PVOB

Project: Airline_Prj

View: Gulsah_Airline_Prj_Int

Select the Components

- Airliner_VOB\Drivers
- Airliner_VOB\Software
  - SystemManagement
  - DataExchange
  - UI
  - Navigation
  - Communication
- Airliner_VOB\Test

- Simulators
- SystemManagement

Figure 4.5: Step 1: Selecting Single-Line Projects and Components

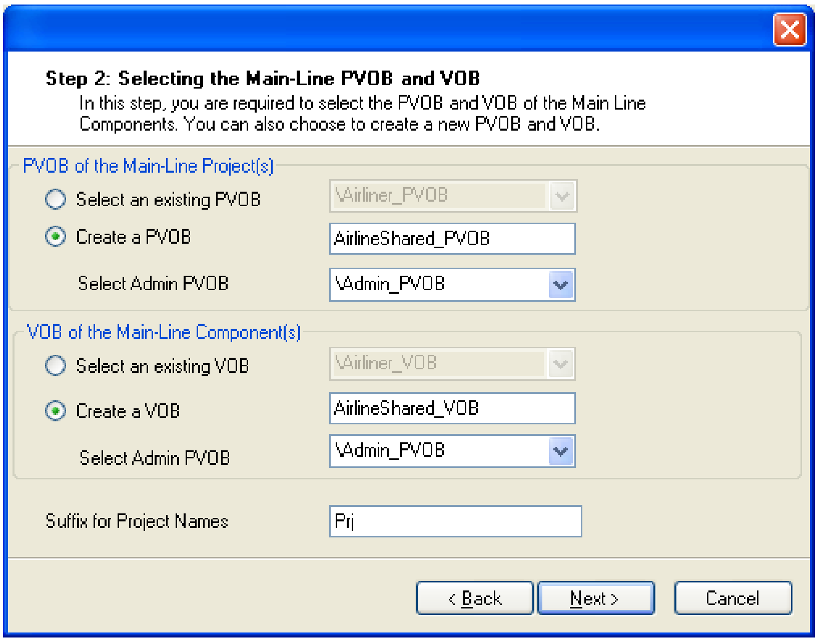
First step of Single-Line to Main-Line pattern reconfiguration wizard is selecting the single-line components and a screen capture of the page is shown in Figure 4.5. In order to select the components to be transformed, the project that is developing the component should be selected. Projects in ClearCase resides in project VOBs (PVOBs) like all other Unified Change Management(UCM) objects [A]. Thus, to access a component developed by a specific project in ClearCase, first the PVOB holding them should be accessed. First page of the wizard is designed to enable users to access the components they are going to select easily.

In ClearCase, components are developed by developers using UCM views, because views provide a work area for users to develop the software components. Each view is configured to fetch one version of the elements from the element's version tree, and thus the contents of the component seen by a view is unique to that specific view. For this reason, users should select the view context in the project to list the inner structure(folder structure) of the components.

On the first page, the box on the left contains the list of components developed by selected single-line project in the form of a tree. The first level nodes on the tree correspond to the components and each node below the first level correspond to the folders under the components. Depending on the update decided on component structures, a component itself or folder(s) under it(sub-components) can be a candidate to be a main-line component, and the candidate main-line components should be moved to the right box with the help of buttons in the middle. Once a folder(sub-component) is selected to be a main-line component, selection of the parent or child folder is restricted. Apart from that, there is no restriction on how many components are selected or remained, because some components can still be developed with Single-Line pattern while others are developed with Main-Line pattern. Each first level node in right box corresponds to a component that will be developed by a main-line project. According to the Figure 4.5, components that are going to be transformed into main-line components are:

- *Simulators* of the component *Test* under *Airliner_VOB*
- *SystemManagement* of the component *Software* under *Airliner_VOB*

When all of the fields on the page are filled and components are selected, *Next* button should be selected to proceed with the second step. *Cancel* and *Back* buttons can also be used to cancel the wizard or go back in the wizard at this step.



Step 2: Selecting the Main-Line PVOB and VOB

In this step, you are required to select the PVOB and VOB of the Main Line Components. You can also choose to create a new PVOB and VOB.

PVOB of the Main-Line Project(s)

○ Select an existing PVOB: \Airliner_PVOB

● Create a PVOB: AirlineShared_PVOB

Select Admin PVOB: \Admin_PVOB

VOB of the Main-Line Component(s)

○ Select an existing VOB: \Airliner_VOB

● Create a VOB: AirlineShared_VOB

Select Admin PVOB: \Admin_PVOB

Suffix for Project Names: Prj

Figure 4.6: Step 2: Providing Main-Line PVOB and VOB information

Second step of Single-Line to Main-Line pattern reconfiguration wizard is deciding on the PVOB and VOBs that are going to be used for main-line project and components and a screen capture of the page is shown in Figure 4.6. It is a choice to determine the PVOBs and VOBs because, user may select to create new PVOBs and/or VOBs, or select to use existing PVOBs or VOBs for the main-line project and components.

Main-Line PVOB will hold the information of to-be created new project (main-line project) and previously selected(in the first step) to-be main-line components. It is optional for the user to create a new main-line PVOB or use an existing main-line PVOB. The aim at the end is to create a new main-line project and if the user finds using an existing PVOB is appropriate and useful considering their project team and structure, selecting the PVOB from the drop-down list will be the best for their choice. Besides, if the user thinks creating a new PVOB for their new main-line project is right for them, the user should enter the name of the new PVOB to the text field labelled *Create a PVOB*. And also, the user should select the administrative PVOB(refer to the Appendix A for administrative PVOB concept) of the new PVOB from drop-down list.

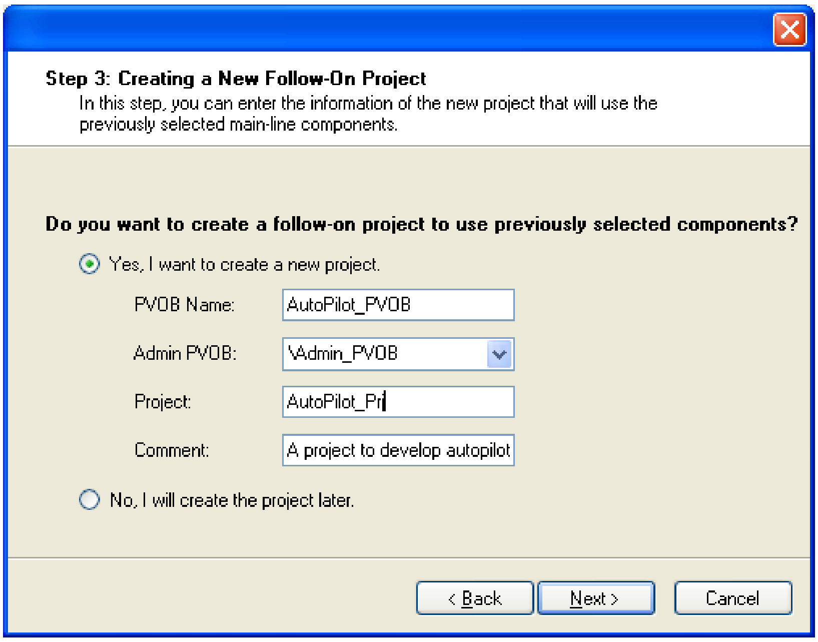
Main-Line VOB will hold the software components since VOBs keep all versions of file elements, directory elements and meta-data associated with them. Assuming a VOB as a folder, components correspond to sub-folders of a VOB from the first level. Like in PVOB case, it is optional for the user to create a new main-line VOB or use an existing VOB to hold the components. If the user wants to place new main-line components under an existing VOB, then the VOB should be selected from the drop-down list labelled *Select and existing VOB*, or name of the VOB should be entered to the text field labelled *Create a VOB* if the user selects to create a new VOB.

At the end of the reconfiguration, for each main-line component selected in first step of the wizard, a new project will be created. In other words, each main-line component will be developed in separate main-line projects. We require users to determine a suffix for their project names, if they have any naming conventions for their project names. To illustrate, if a main-line component's name is *Sample_Comp* and *Prj* is decided for the project name suffix, then the main-line project's name will be *Sample_Comp_Main_Prj*. The identifier *Main* gives the users of this project the idea of this project is a main-line project. More about the resulting structure will be explained at the end of this section.

According to Figure 4.6,

- A PVOB named *AirlineShared_PVOB* will be created as main-line PVOB.
- A VOB named *AirlineShared_VOB* will be created to store main-line components.
- *Admin_PVOB* will be the administrator VOB of both PVOB and VOB.
- *Prj* will be the suffix for main-line project names. (i.e. *Simulator_Main_Prj* will be one of the project names)

When all of the fields on the page are filled or selected, *Next*button should be selected to proceed with the third step. *Cancel* and *Back* buttons can also be used to cancel the wizard or go back in the wizard at this step.



Step 3: Creating a New Follow-On Project

In this step, you can enter the information of the new project that will use the previously selected main-line components.

Do you want to create a follow-on project to use previously selected components?

● Yes, I want to create a new project.

PVOB Name: AutoPilot_PVOB

Admin PVOB: \Admin_PVOB

Project: AutoPilot_Prj

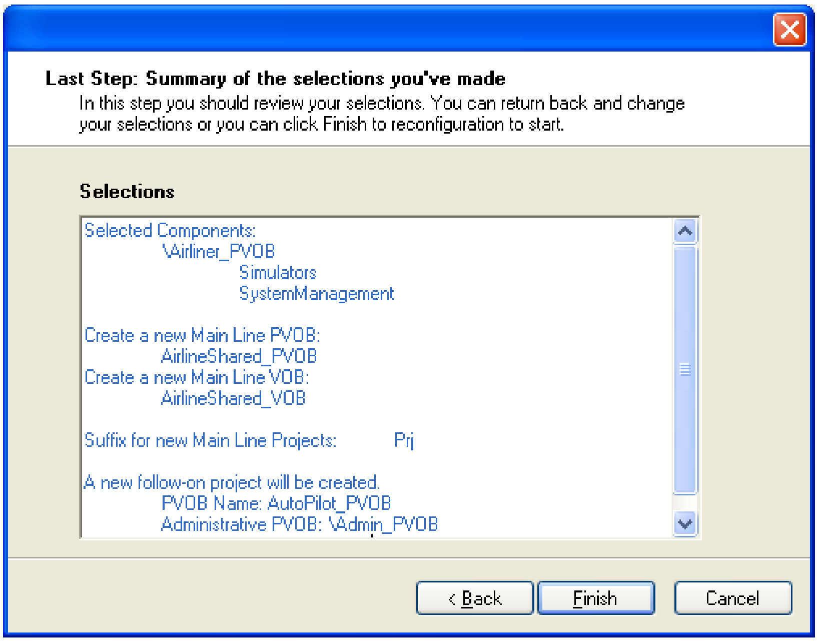
Comment: A project to develop autopilot

○ No, I will create the project later.

Figure 4.7: Step 3: Creating a Follow-On Project

Third step of Single-Line to Main-Line pattern reconfiguration wizard is asking the users whether they are going to develop a new project using new main-line components and a screen capture of the page is shown in Figure 4.7. Main-Line components can be developed by more than one project but, all projects developing the components should deliver&merge their work to the main-line project. If the new components will also be developed by a new follow-on project, then the users should select the first *radio* button labelled *Yes, I want to create a new* *project* and fill in the fields required. Firstly, name of the PVOB that will hold the project data should be entered and the administrative PVOB of new PVOB should be selected from the drop-down list. Next, name of the project should be given and lastly a comment for the project can be provided by the user. According to Figure 4.7, a new follow-on project named *AutoPilot_Prj* will be created under *AutoPilot_PVOB*.

User can also prefer not to create a new follow-on project, because components may not be developed with multiple projects right now or user may want to create the new follow-on project later. So, selecting the second radio button will be the right choice.



Last Step: Summary of the selections you've made

In this step you should review your selections. You can return back and change your selections or you can click Finish to reconfiguration to start.

Selections

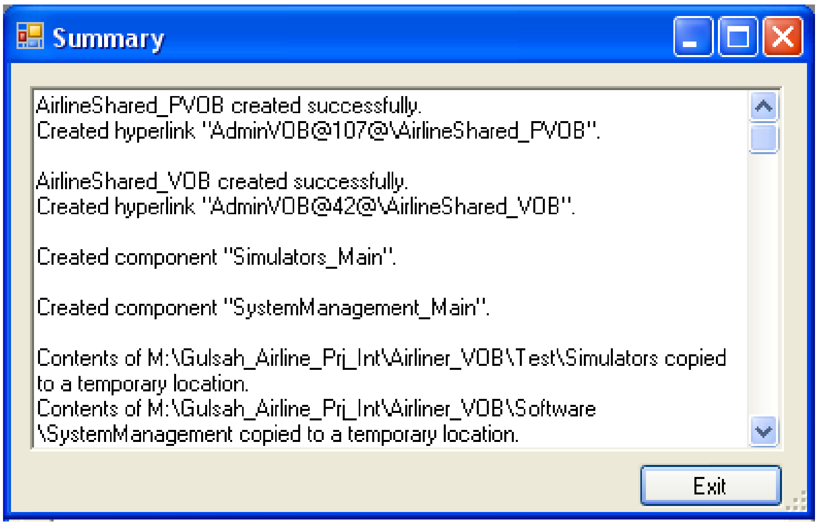
```
Selected Components:
        \Airliner_PVOB
                Simulators
                SystemManagement

Create a new Main Line PVOB:
        AirlineShared_PVOB
Create a new Main Line VOB:
        AirlineShared_VOB

Suffix for new Main Line Projects:        Prj

A new follow-on project will be created.
        PVOB Name: AutoPilot_PVOB
        Administrative PVOB: \Admin_PVOB
```

Figure 4.8: List of selections

After completing first, second and third steps of the wizard, a list of user's selections are shown to the user. This is the last step of Single-Line to Main-Line pattern reconfiguration wizard and a screen-shot of the page is shown in Figure 4.8.

Last step lists all the selections made by the user in previous steps, giving user the chance of reviewing the selections that he/she has made. If user detects a mistake in selections done in the previous steps, he/she can easily go back to that step and fix it.

After the selections are made and reviewed, the user should click on the *Finish* button to let DSCM Single-Line to Main-Line pattern reconfiguration wizard update the SCM structure. Updating the SCM structure may take a while because the following actions are performed



```
AirlineShared_PVOB created successfully.
Created hyperlink "AdminVOB@107@\AirlineShared_PVOB".

AirlineShared_VOB created successfully.
Created hyperlink "AdminVOB@42@\AirlineShared_VOB".

Created component "Simulators_Main".

Created component "SystemManagement_Main".

Contents of M:\Gulsah_Airline_Prj_Int\Airliner_VOB\Test\Simulators copied
to a temporary location.
Contents of M:\Gulsah_Airline_Prj_Int\Airliner_VOB\Software
\SystemManagement copied to a temporary location.
```

Figure 4.9: The summary of performed activities by DSCM Pattern Reconfiguration Tool

by the wizard in listed order:

- Creation of the VOBs:
  - If new PVOB creation was required by the user, Main-Line PVOB is created with the name entered in Step 2.
  - A *hyper-link* of type *AdminVOB* is created between PVOB and the selected administrator PVOB.
  - If new VOB creation was required by the user, Main-Line VOB which will hold the components contents is created with the name entered in Step 2.
  - A *hyper-link* of type *AdminVOB* is created between VOB and the selected administrator PVOB.
- Creation of the Main-Line components
  - For each component/sub-component that is selected in first step of the wizard, create a component whose contents will be stored in the Main-Line VOB. These components will be managed by the Main-Line PVOB. To indicate these components are Main-Line components, *_Main* suffix is added to the name of the components, i.e. a component(or sub-component) formerly named *Tools* is now named as *Tools_Main*.

- When new components are created, the contents of them were empty. To fill in the contents of Main-Line components, their original contents(file and directory elements) which were developed by the Single-Line project are copied to a temporary location. They are moved under the new components after their Main-Line projects are created.

- Creation of the Main-Line projects

  - For each component created in previous step, a project which will be the Main-Line project of the components as all Follow-on projects will deliver&merge their works to this project. The name of each project consists of the name of component, the *_Main* and the suffix entered in second step of the wizard, i.e., with a suffix *_Prj* the name of project will be *Tools_Main_Prj*.
  - An *integration stream* is created for each Main-Line project.
  - A *view* is created for each integration stream of the Main-Line project.
  - An *activity* is created to add the contents of components to source control.
  - Contents of components in the temporary location as mentioned in previous step (Creation of the Main-Line components), are moved under the Main-Line components and all of the elements are added to source control by using the activity which has just been created.
  - A baseline is made and recommended in the integration stream of each Main-Line project as it will form the basis to the follow-on projects to rebase.

- Creation of the Follow-on project (If required)

  - A PVOB for holding the follow-on project is created with the name entered in Step 3.
  - A *hyper-link* of type *AdminVOB* is created between PVOB and the selected administrator PVOB.
  - Follow-on project which has the Main-Line components in its modifiable component list is created.
  - An integration stream is created for the Follow-on project to enable developers to join the development easily.

  - Rebased to the latest baselines of Main-Line components and baselines are recommended in the integration stream.

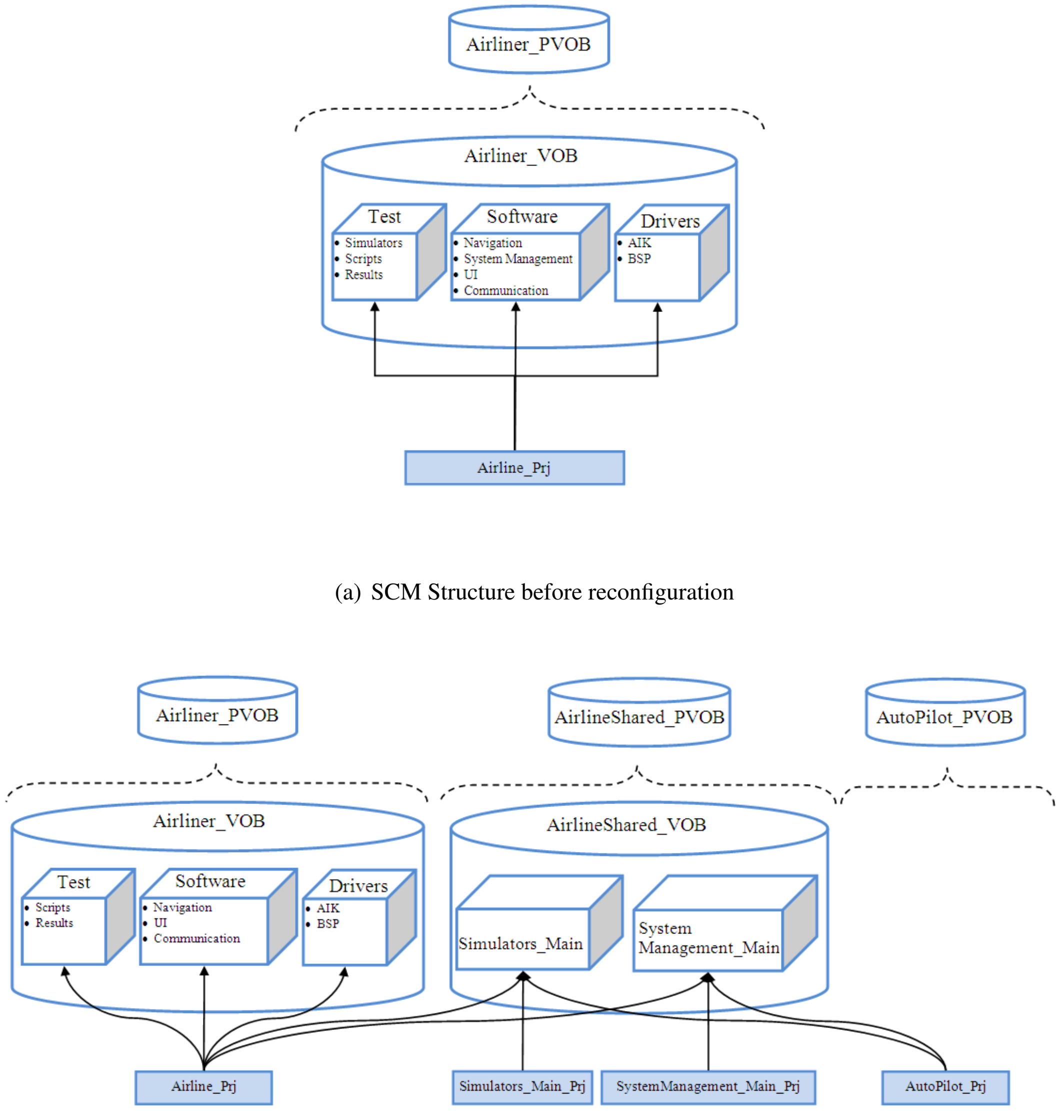
- Changing the configuration of first(Single-Line) project to adjust new SCM structure
  - Integration stream of the project is rebased to the latest baseline of new Main-Line components.
  - New components are added to the project's modifiable components list.
  - Latest baselines of new components are recommended in the integration stream to the child streams.

When all of the actions above are performed successfully by the wizard, the result of each performed action is shown to the user as shown in Figure 4.9. This informative summary page enables user to review what has been done by the wizard. An example summary page output is given in Appendix B.

To give a better understanding of the resultant SCM structure, SCM structure before and after the reconfiguration process is given in Figure 4.10. SCM structure before reconfiguration can be explained as follows:

- *Airliner_PVOB* was the project VOB of the components *Test, Drivers* and *Software* and Airline_Prj.
- *Simulators, Scripts* and *Results* were folders(sub-component) in *Test* component.
- *Navigation, SystemManagement, UI* and *Communication* were folders(sub-component) in *Software* component.
- *AIK, BSP* were folders(sub-component) in *Drivers* component.
- *Airline_Prj* was developing *Test, Drivers* and *Software* components.

After deciding that *Simulators* sub-component of *Test* component and *SystemManagement* sub-component of *Software* component are eligible to be separate components and these components should be managed by a main-line project to serve other contributing projects as a central codeline, reconfiguring from Single-Line to Main-Line pattern was performed and the



(a) SCM Structure before reconfiguration

(b) SCM Structure after reconfiguration

Figure 4.10: **(a)** SCM structure of the single-line project(Airline_Prj) before reconfiguration and **(b)** SCM structure of main-line and follow-on projects after reconfiguration

resultant SCM structure is shown in Figure 4.10(b). The resultant structure can be explained as follows:

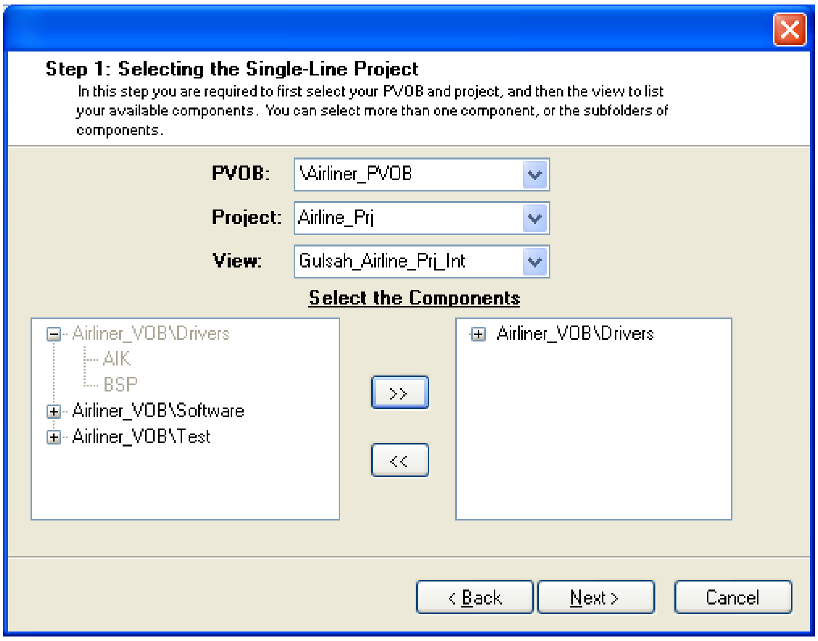
- *AirlineShared_PVOB* is the project VOB of the main-line components *Simulators_Main* and *SystemManagement_Main* and their projects.
- *Simulators_Main_Prj* is the main-line project of Simulators_Main component.
- *SystemManagement_Main_Prj* is the main-line project of SystemManagement_Main component.
- *AutoPilot_PVOB* is the project VOB of the *AutoPilot_Prj* project.
- *AutoPilot_Prj* and *Airline_Prj* are follow-on projects for contributing development of main-line components.

4.3 Reconfiguring from Single-Line to Producer-Consumer Pattern

As explained in Section 4.2, trigger to pattern transitions is the updated component structures. If components in a VOB and/or sub-components of components which are being developed by one project are decided to be developed by only one project and used by other projects, then a SCM structure reconfiguration from Single-Line pattern to Producer-Consumer pattern should be performed. Reconfiguring from Single-Line pattern to Producer-Consumer pattern is performed by a wizard which can be started by selecting second radio button which is labelled ”Want to switch from Single-Line to Producer-Consumer Pattern” in Figure 4.4. Single-Line to Producer-Consumer pattern reconfiguration wizard's user interface elements is very similar to the Single-Line to Main-Line pattern reconfiguration wizard, because the initial state of the SCM structure is the same.

First step of Single-Line to Producer-Consumer pattern reconfiguration wizard is selecting the single-line components and a screen capture of the page is shown in Figure 4.11. In order to select the components to be reused, the project that is developing the component and the PVOB of the project should be selected. A view should also be selected to list the contents of components.

After the selection of view, the contents of all components developed by selected project are



Step 1: Selecting the Single-Line Project

In this step you are required to first select your PVOB and project, and then the view to list your available components. You can select more than one component, or the subfolders of components.

PVOB: \Airliner_PVOB

Project: Airline_Prj

View: Gulsah_Airline_Prj_Int

Select the Components

- Airliner_VOB\Drivers
  - AIK
  - BSP
- Airliner_VOB\Software
- Airliner_VOB\Test

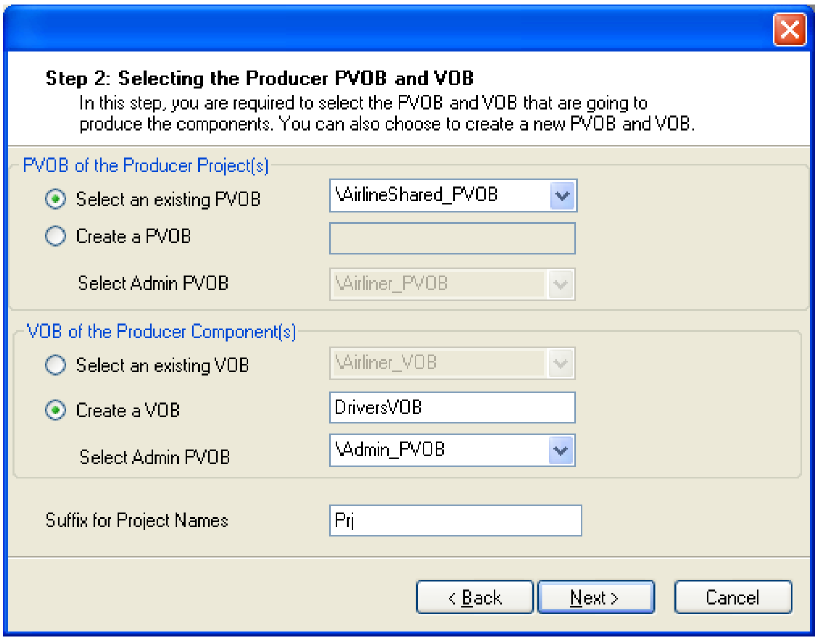
- Airliner_VOB\Drivers

Figure 4.11: Step 1: Selecting Single-Line Projects and Components

listed as a tree structure on the left list box. The first level nodes on the tree correspond to the components and each node below the first level correspond to the folders under the components. Depending on the decided update on component structures, a component itself or folder(s) under it(sub-components) can be a candidate to be a reusable component, and the candidate reusable components should be moved to the right box with the help of buttons in the middle. Once a folder(sub-component) is selected to be a reusable component, selection of the parent or child folder is restricted. Apart from that, there is no restriction on how many components are selected or remained, because some components can still be developed with single-line project while others are developed with a producer project. Each first level node in right box corresponds to a component that will be developed by a producer project. According to the Figure 4.11, only *Drivers* component under *Airliner_VOB* will be transformed into a reusable component.

When all of the fields on the page are filled and components are selected, *Next*button should be selected to proceed with the second step. *Cancel* and *Back* buttons can also be used to cancel the wizard or go back in the wizard at this step.

Second step of Single-Line to Producer-Consumer pattern reconfiguration wizard is deciding on the PVOB and VOBs that are going to be used for Producer project and components, a



Step 2: Selecting the Producer PVOB and VOB

In this step, you are required to select the PVOB and VOB that are going to produce the components. You can also choose to create a new PVOB and VOB.

PVOB of the Producer Project(s)

● Select an existing PVOB: \AirlineShared_PVOB

○ Create a PVOB:

Select Admin PVOB: \Airliner_PVOB

VOB of the Producer Component(s)

○ Select an existing VOB: \Airliner_VOB

● Create a VOB: DriversVOB

Select Admin PVOB: \Admin_PVOB

Suffix for Project Names: Prj

Figure 4.12: Step 2: Providing Producer Project's PVOB and VOB information

screen capture of the page is shown in Figure 4.12. The user has two choices while deciding on producer PVOB and VOB. First one of the choices is to create new PVOB and/or VOB, the other choice is to use existing PVOB and/or VOB for the producer project and components.

Producer PVOB will hold the information of producer project and previously selected(in the first step) to-be reusable component(s). It is optional to the user to create an new producer PVOB or use an existing PVOB. The aim at the end is to create a new producer project and if the user finds using an existing PVOB is appropriate and useful considering their project team and structure, an existing PVOB should be selected form the drop-down list if PVOBs. Besides, if the user wants to create a new PVOB for the producer project(s), the user should enter the name of the new PVOB to the text field labelled *Create a PVOB*. And also, the user should select the administrative PVOB(refer to the Appendix A for administrative PVOB concept) of the new PVOB from drop-down list.

Producer VOB will hold the reusable components. Like in PVOB case, it is optional to the user to create a new producer VOB or use an existing VOB to hold the components. If the user wants to place new reusable components under an existing VOB, then the VOB should be selected from the drop-down list labelled *Select and existing VOB*, or name of the VOB should be entered to the text field labelled *Create a VOB* if the user selects to create a new

VOB.

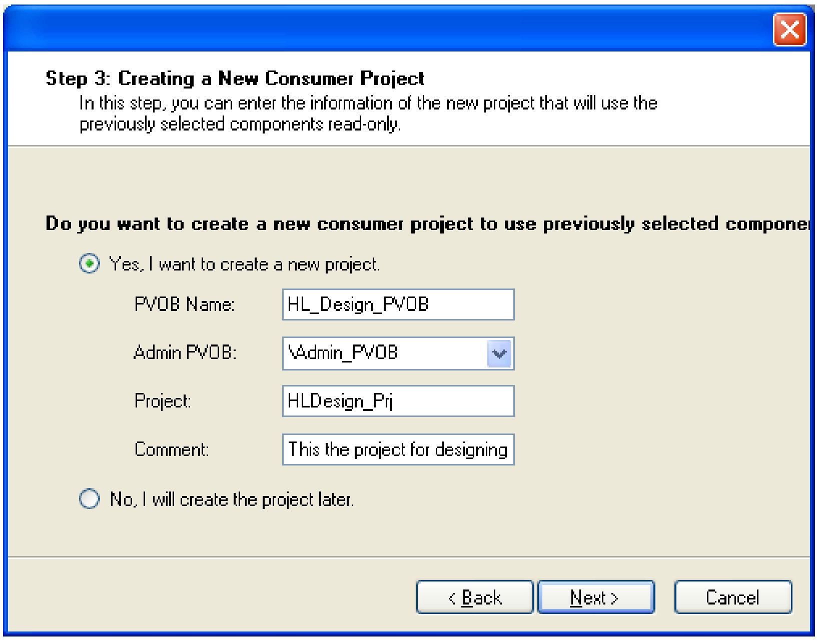
At the end of the reconfiguration, for each new component selected in first step of the wizard, a new project will be created. In other words, each reusable component will be developed in separate producer projects. We require users to determine a suffix for their project names as in previous wizard, if they have any naming conventions for their project names. To illustrate, if a reusable component's name is *Reuse_Comp* and *Prj* is decided for the project name suffix, then the producer project's name will be *Reuse_Comp_Producer_Prj*. The identifier *Producer* gives the users of this project the idea of Producer-Consumer pattern. More about the resulting structure will be explained at the end of this section.

To sum up the selections made in Figure 4.12:

- An existing PVOB named *AirlineShared_PVOB* will be used as producer PVOB.
- A VOB named *DriversVOB* will be created to store reusable component.
- *Admin_PVOB* will be the administrator VOB *DriversVOB*.
- *Prj* will be the suffix for producer project names. (i.e. *Drivers_Producer_Prj* will be the project name of reusable component's producer project)

When all of the fields on the page are filled or selected, *Next*button should be selected to proceed with the third step. *Cancel* and *Back* buttons can also be used to cancel the wizard or go back in the wizard at this step.

Third step of Single-Line to Producer-Consumer pattern reconfiguration wizard is asking the users whether the reusable components will be reused by an additional consumer project or will not and a screen capture of the page is shown in Figure 4.13. Since reusable components can be developed by only one project (producer project), any other project that wants to use the component should access the component as read-only. During reconfiguring, if it is decided to reuse the component by a new consumer project, then the user should select the first *radio* button labelled *Yes, I want to create a new project* and fill in the fields required. Firstly, name of the PVOB that will hold the project data should be entered and the administrative PVOB of new PVOB should be selected from the drop-down list. Next, name of the project should be given and lastly a comment for the project can be provided by the user. According to Figure 4.13, a new consumer project named *HLDesign_Prj* will be created under *HL_Design_PVOB*.



Step 3: Creating a New Consumer Project

In this step, you can enter the information of the new project that will use the previously selected components read-only.

Do you want to create a new consumer project to use previously selected compone

● Yes, I want to create a new project.

PVOB Name: HL_Design_PVOB

Admin PVOB: \Admin_PVOB

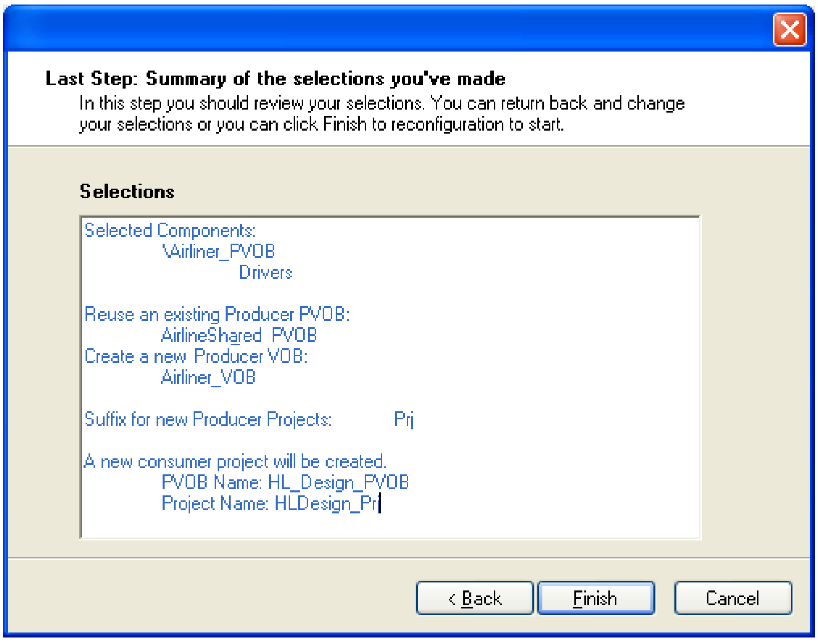
Project: HLDesign_Prj

Comment: This the project for designing

○ No, I will create the project later.

Figure 4.13: Step 3: Creating a consumer project

User can also prefer not creating a new consumer project, because components may not be reused by additional projects right now, or user may want to create the new consumer project later. So, the second radio button should be selected for this choice.



Last Step: Summary of the selections you've made

In this step you should review your selections. You can return back and change your selections or you can click Finish to reconfiguration to start.

Selections

```
Selected Components:
          \Airliner_PVOB
                    Drivers

Reuse an existing Producer PVOB:
          AirlineShared_PVOB
Create a new  Producer VOB:
          Airliner_VOB

Suffix for new Producer Projects:         Prj

A new consumer project will be created.
          PVOB Name: HL_Design_PVOB
          Project Name: HLDesign_Prj
```

Figure 4.14: List of selections

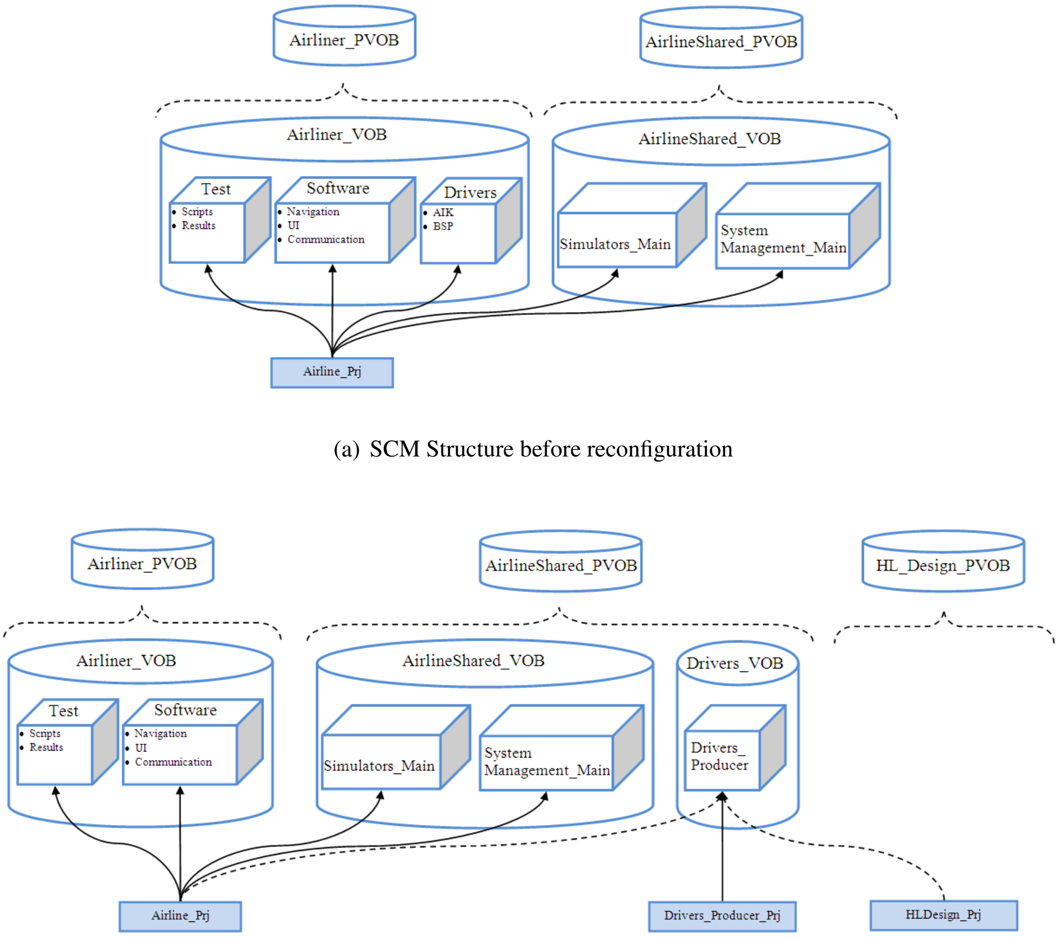
After completing first, second and third steps a list of user's selections are shown to the user as in the previous pattern reconfiguration wizard. This is the last step of Single-Line to Producer-Consumer pattern reconfiguration wizard and a screen-shot of the page is shown in Figure 4.14. Last step lists all the selections made by the user in previous steps, giving user the chance of reviewing the selections that he/she has made. If user detects a mistake in selections done in the previous steps, he/she can easily go back to that step and fix it. Giving this opportunity to the user is important because a single mistake may cause the new SCM structure to be built from scratch. After the selections are made and reviewed, the user should click on the *Finish* button to let DSCM Single-Line to Producer-Consumer pattern reconfiguration wizard update the SCM structure. Updating the SCM structure may take a while because the following actions are performed by the wizard in listed order:

- Creation of the VOBs
  - If new PVOB creation was required by the user, Producer PVOB is created with the name entered in Step 2.
  - A *hyper-link* of type *AdminVOB* is created between PVOB and the selected administrator PVOB.
  - If new VOB creation was required by the user, VOB which will hold the reusable components is created with the name entered in Step 2.
  - A *hyper-link* of type *AdminVOB* is created between VOB and the selected administrator PVOB.
- Creation of the reusable components
  - For each component/sub-component that is selected in first step of the wizard, component whose contents will be stored in the producer VOB is created. These components will be managed by the producer PVOB. To indicate these components are developed by producer projects, *_Producer* suffix is added to the names of the components, i.e. a component(or sub-component) formerly named *Tools* is now named as *Tools_Producer*.
  - When new components are created, the contents of them were empty. To fill in the contents of Main-Line components, their original contents(file and directory

elements) which were developed by the Single-Line project are copied to a temporary location. They are moved under the new components after their producer projects are created.

- Creation of the Producer projects
  - For each component created in previous step, a project which will develop the projects is created. The name of each project consists of the name of component, the *_Producer* and the suffix entered in second step of the wizard, i.e., with a suffix *_Prj* the name of project will be *Tools_Producer_Prj*.
  - An *integration stream* is created for each producer project.
  - A *view* is created for each integration stream of the producer project.
  - An *activity* is created to add the contents of components to source control.
  - Contents of components in the temporary location as mentioned in previous step (Creation of the reusable components), are moved under the reusable components, and all of the elements are added to source control by using the activity which has just been created.
  - A baseline is made and recommended in the integration stream of each producer project as it will form the basis to the consumer projects to rebase.
- Creation of the consumer project (If required)
  - A PVOB for holding the consumer project is created with the name entered in Step 3.
  - A *hyper-link* of type *AdminVOB* is created between PVOB and the selected administrator PVOB.
  - Consumer project is created.
  - Reusable components are added to the configuration of consumer project as read-only.
  - An integration stream is created for the consumer project to enable developers to join the development easily.
  - Rebased to the latest baselines of reusable components and baselines are recommended in the integration stream.

- Changing the configuration of first(Single-Line) project to adjust new SCM structure
  - Integration stream of the project is rebased to the latest baseline of new reusable components.
  - Latest baselines of new components are recommended in the integration stream to the child streams.



(a) SCM Structure before reconfiguration

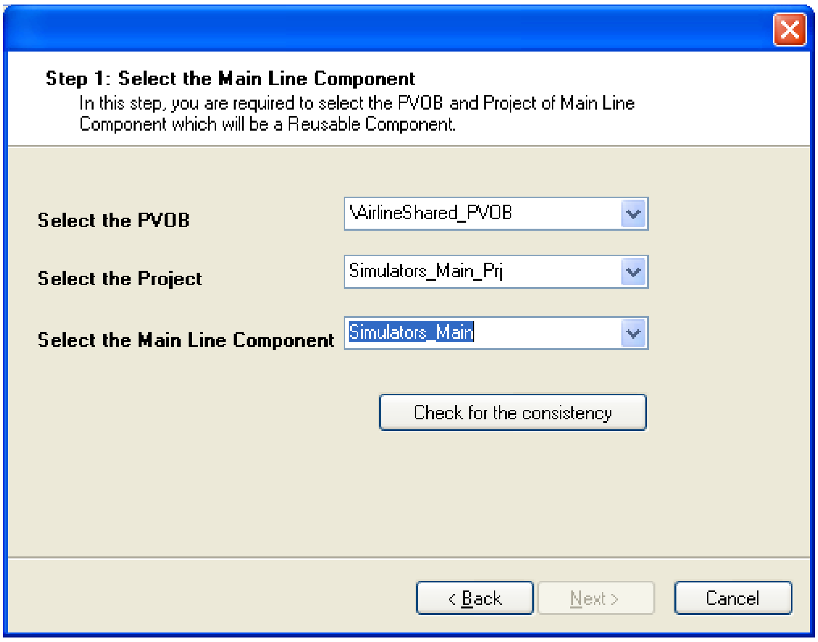
(b) SCM Structure after reconfiguration

Figure 4.15: **(a)** SCM structure of the single-line project(Airline_Prj) before reconfiguration and **(b)** SCM structure of producer and consumer projects after reconfiguration

4.4 Reconfiguring from Main-Line to Producer-Consumer Pattern

When developing a component in a main-line structure, after some amount of time, the component may come to a level of maturity, by maturity we mean a stable release of the component. That kind of components require changes and/or fixes very rarely and it would be more easy to do the changes in a single codeline and make other projects only rebase to the new releases of the component. So it is essential to transform main-line components to reusable components and the Dynamic SCM system we propose makes this adjustment easily.

Reconfiguration from Main-Line pattern to Producer-Consumer pattern is done with a wizard as in previous transitions. The wizard is opened with selecting the third radio button which is labelled ”Want to switch from Single-Line to Producer-Consumer Pattern” in Figure 4.4.



Step 1: Select the Main Line Component
In this step, you are required to select the PVOB and Project of Main Line Component which will be a Reusable Component.

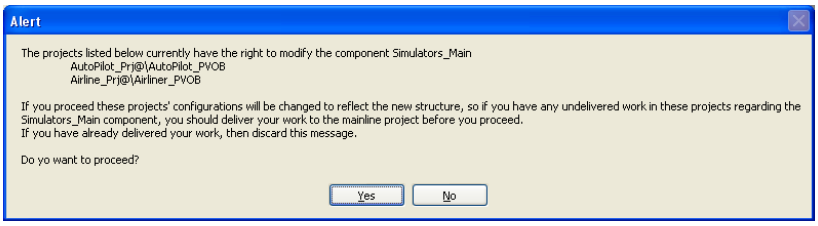
Select the PVOB: \AirlineShared_PVOB

Select the Project: Simulators_Main_Prj

Select the Main Line Component: Simulators_Main

Figure 4.16: Step 1: Selecting Main-Line Component

From the user interface point of view, first step of Main-Line to Producer-Consumer pattern reconfiguration wizard is selecting the main-line components and a screen capture of the page is shown in Figure 4.16. In order to select the components to be reused, the project that is developing the component and the PVOB of the project should be selected. Unlike previous reconfiguration wizards, this wizard handles the reconfiguration for one main-line component at a time. When the main-line component which is going to be a reusable component is selected, the *Next* button is not enabled immediately. The user should first click on the *Check for the consistency* button to see which projects are currently working on the development of the component. Those project teams should be notified prior to the reconfiguring process because when the reconfiguration completes they will not be able to do any changes to the component any more.



Alert

The projects listed below currently have the right to modify the component Simulators_Main
AutoPilot_Prj@\AutoPilot_PVOB
Airline_Prj@\Airliner_PVOB

If you proceed these projects' configurations will be changed to reflect the new structure, so if you have any undelivered work in these projects regarding the Simulators_Main component, you should deliver your work to the mainline project before you proceed.
If you have already delivered your work, then discard this message.

Do yo want to proceed?

Figure 4.17: Step 1: Checking for the projects developing the main-line component

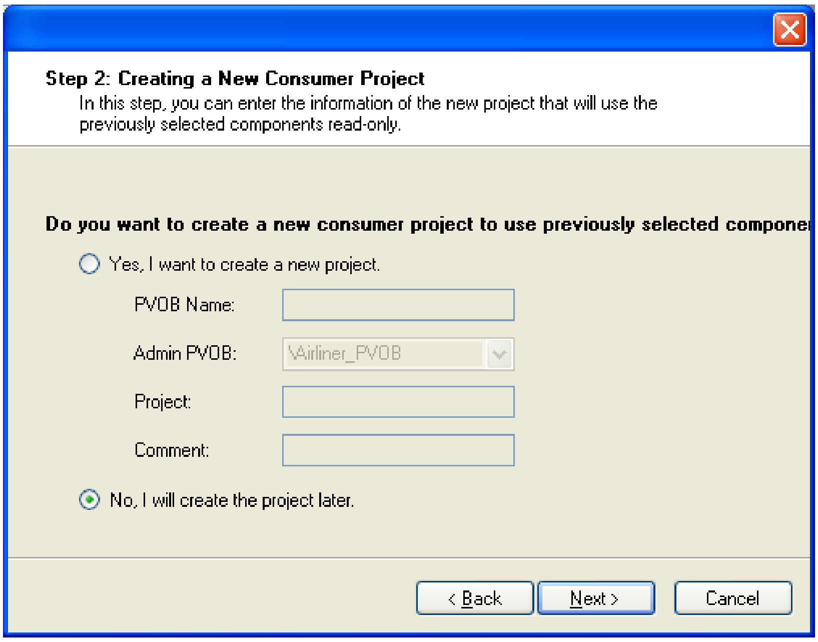
Figure 4.17 shows the alert window when *Check for the consistency* button is clicked. The list of projects which are currently working on the main-line component is given in the alert window. The user is warned with this alert to make the user notify other users of this main-line component. If they have any undelivered work, they should deliver their changes to the main-line project for the last time. According to the Figure 4.16, *Simulators_Main* component is selected to be a reusable component and consistency check showed that this component is developed by the following follow-on projects:

- AutoPilot_Prj
- Airliner_Prj

After reconfiguring is completed, these projects will be consumer projects of *Simulators_Main* component.When user says *Yes* in the warning window, *Next*button gets enabled and should be selected to proceed with the second step. *Cancel* and *Back* buttons can also be used to cancel the wizard or go back in the wizard at this step.

Second step of the wizard corresponds to the third step of previous wizards. Figure 4.18 shows the page of second step which asks the user whether the reusable component will be reused by an additional consumer project or not. If user selects to create a new project, then user should provide the information required. If user selects not to create a new project then all of the above fields are disabled and user can navigate to the next step. According to the Figure 4.18, a new consumer project will not be created.

After completing first and second steps a list of user's selections are shown to the user as in the previous pattern reconfiguration wizards. This is the last step of Main-Line to Producer-Consumer pattern reconfiguration wizard and the page is shown in Figure 4.19. Last step lists



Step 2: Creating a New Consumer Project
In this step, you can enter the information of the new project that will use the previously selected components read-only.

Do you want to create a new consumer project to use previously selected compone

○ Yes, I want to create a new project.

PVOB Name:

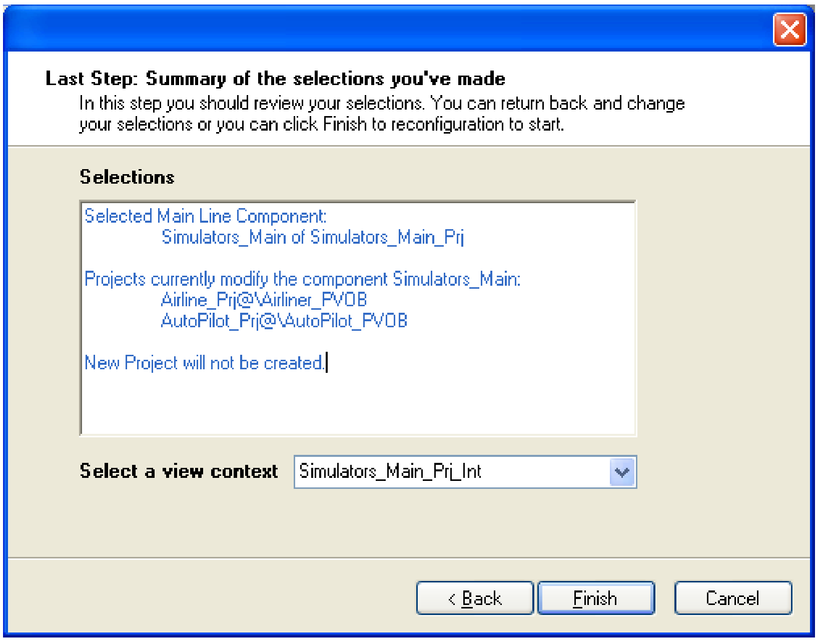
Admin PVOB: \Airliner_PVOB

Project:

Comment:

● No, I will create the project later.

Figure 4.18: Step 2: Creating a consumer project



Last Step: Summary of the selections you've made
In this step you should review your selections. You can return back and change your selections or you can click Finish to reconfiguration to start.

Selections

```
Selected Main Line Component:
          Simulators_Main of Simulators_Main_Prj

Projects currently modify the component Simulators_Main:
          Airline_Prj@\Airliner_PVOB
          AutoPilot_Prj@\AutoPilot_PVOB

New Project will not be created.
```

Select a view context Simulators_Main_Prj_Int

Figure 4.19: List of selections

all the selections made by the user in previous steps, giving user the chance of reviewing the selections that he/she has made. If user detects a mistake in selections done in the previous steps, he/she can easily go back to that step and fix it. Giving this opportunity to the user is important because a single mistake may cause the new SCM structure to be built from scratch. In this step besides listing the selections of previous steps, the user is required to select a view context where the reconfiguring process will take place. After selections are reviewed and a view context is selected from the list, the user should click on the *Finish* button to let DSCM Main-Line to Producer-Consumer pattern reconfiguration wizard update the SCM structure. Updating the SCM structure may take a while depending on the size(number of file and directory elements) in the reusable components, because of the *trigger* that is going to be attached to all elements of component.

Operations to perform reconfiguring from Main-Line pattern to Producer-Consumer pattern differ from other reconfigurations explained in sections 4.2 and 4.3. In previous reconfigurations, to accomplish a main-line and producer-consumer structure, components were moved to their new locations (VOBs) and a project (main-line or producer) was created for each component. Besides, as can be understood from the previous sections, the main difference between a Main-Line pattern and Producer-Consumer pattern is that follow-on projects were allowed to make changes in the components and deliver it to main-line project but consumer projects were not allowed to make changes in the component. In this reconfiguration, since the main-line component has its own project and follow-on projects contributing to the development of the component by delivering their work to the project of main-line component, there is no need to create a new producer project if follow-on projects' rights to change the component are taken from them. When a follow-on project can not make a change in the main-line component, this automatically makes a main-line project a producer project and a follow-on project a consumer project.

Disabling follow-on projects from changing the reusable component is established with triggers. A *trigger* is a monitor that causes one or more procedures or actions to be run whenever a certain Rational ClearCase operation is performed [14]. Triggers can be used to restrict operations to specific users. Triggers can also be used on certain UCM operations to enforce customized development policies for project teams [14]. In our case, we used triggers to restrict any *checkouts* to the reusable components except for the producer projects. We implemented the trigger as a pre-operation trigger of a checkout operation so that, before any checkout operation is performed a check is done to ensure that any element of a component is not about to change by a project which is in the restricted list of the component.

Triggers are attached to every element under a component. In order to attach a trigger to every element under a component, a *trigger type* should be defined. A trigger type is defined with the following command for this section's example reconfiguration:

```
cleartool mktrtype -element -preop checkout -execwin
"ccperl <path> Airline_Prj@Airliner_PVOB AutoPilot_Prj@AutoPilot_PVOB"
chk_prj_Simulators_Main
```

- *mktrtype* is a command of *cleartool* which is used to make trigger types.
- *element* indicates that this trigger type can be applied to elements.
- *preop* indicates that this trigger will be a pre-operation trigger.
- *checkout* indicates that this trigger is fired before checkout operations.
- *execwin* indicates that this trigger's script will be working on Windows environment.
- *ccperl <path> Airline_Prj@Airliner_PVOB AutoPilot_Prj@\AutoPilot_PVOB* is script to call from command-line and <path> is the path to the trigger's perl script. *Airline_Prj@Airliner_PVOB* and *AutoPilot_Prj@\AutoPilot_PVOB* are the projects which are going to be disabled from checkouts and they are given as command-line arguments to the perl script.
- *chk_prj_Simulators_Main* is the name of trigger type.

After defining the trigger type, trigger is attached to all elements under *Simulators_Main* component.

```
cleartool mktrigger -recurse chk_prj_Simulators_Main
M:\Simulators_Main_Prj_int\AirlineShared_VOB\Simulators_Main
```

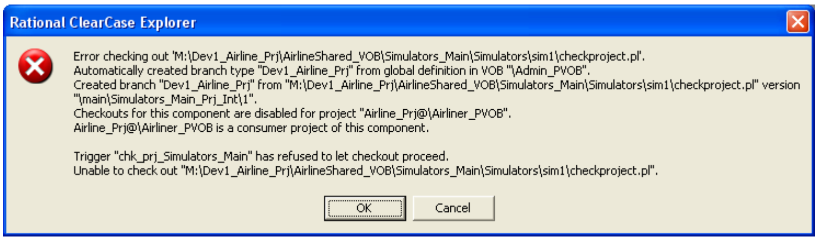
- *mktrigger* is a command of *cleartool* which is used to make triggers.

- *recurse* indicates that this trigger will be applied recursively for each element under *Simulators_Main* component.

- *chk_prj_Simulators_Main* is the name of trigger type that is going to be attached to elements.

- *M:\Simulators_Main_Prj_int\AirlineShared_VOB\Simulators_Main* is the long path of the component combined with the view name.

When triggers are attached to the elements of reusable component, follow-on projects which are Airline_Prj and AutoPilot_Prj will be consumer projects and the main-line project of Simulators_Main component will be the producer project. Then if the user selected to create a new consumer project in second step of the wizard, see Figure 4.18, a consumer project is created as described in section 4.3. In this section's example reconfiguration process, a consumer project is not created.

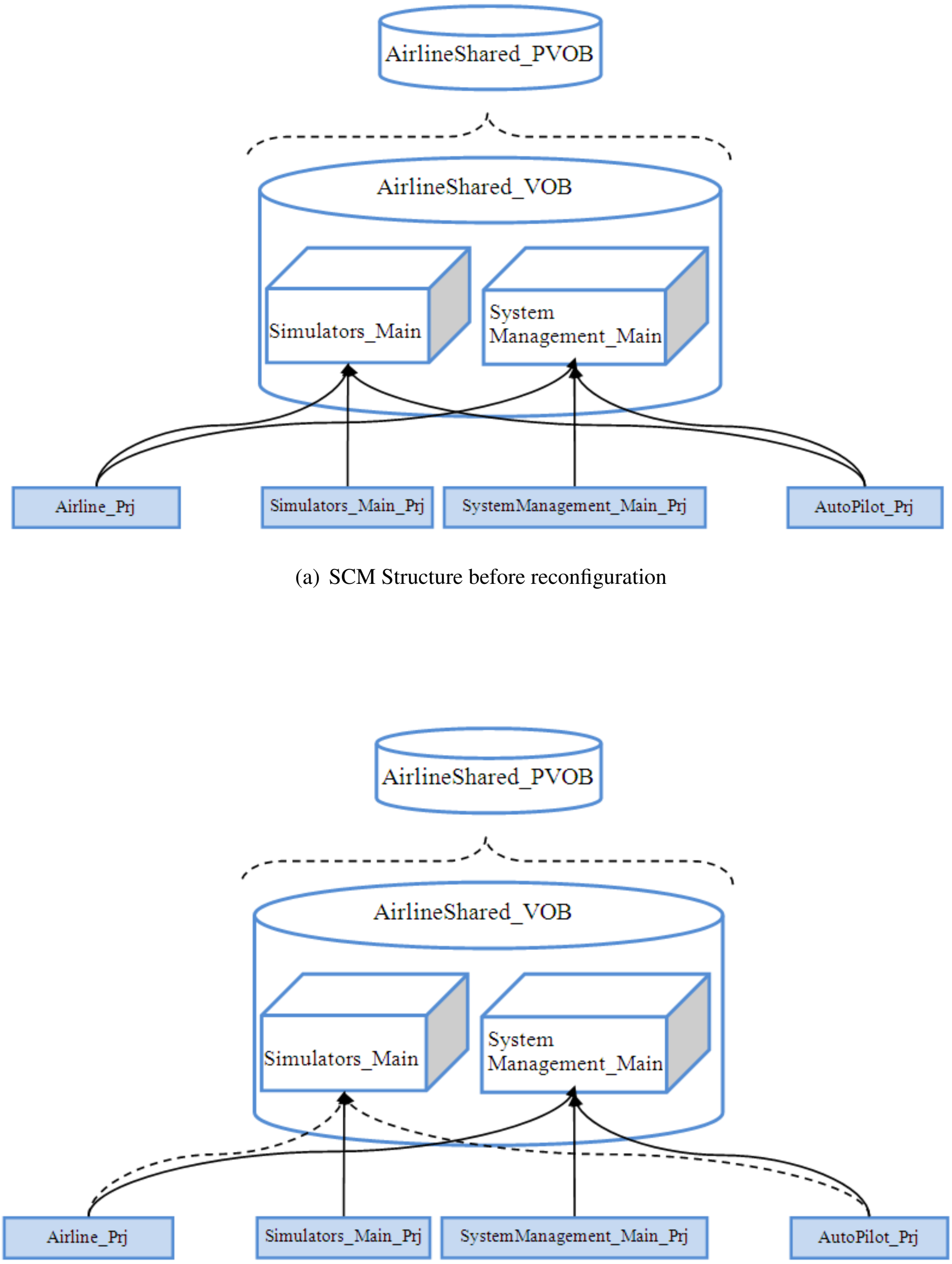


```
Error checking out 'M:\Dev1_Airline_Prj\AirlineShared_VOB\Simulators_Main\Simulators\sim1\checkproject.pl'.
Automatically created branch type "Dev1_Airline_Prj" from global definition in VOB "\Admin_PVOB".
Created branch "Dev1_Airline_Prj" from "M:\Dev1_Airline_Prj\AirlineShared_VOB\Simulators_Main\Simulators\sim1\checkproject.pl" version
"\main\Simulators_Main_Prj_Int\1".
Checkouts for this component are disabled for project "Airline_Prj@\Airliner_PVOB".
Airline_Prj@\Airliner_PVOB is a consumer project of this component.

Trigger "chk_prj_Simulators_Main" has refused to let checkout proceed.
Unable to check out "M:\Dev1_Airline_Prj\AirlineShared_VOB\Simulators_Main\Simulators\sim1\checkproject.pl".
```

Figure 4.20: Checkout warning

Figure 4.21 illustrates the SCM structure of our example system before and after the reconfiguration. Before reconfiguring to Producer-Consumer pattern, Airline_Prj, Simulators_Main_Prj and AutoPilot_Prj had read&write access rights to the Simulators_Main component. After reconfiguration, only Simulators_Main_Prj can write on the component since it is the producer component, but Airline_Prj and AutoPilot_Prj can only read (rebase to) the component since they are the consumer projects. This example's summary page output is given in Appendix B.

When a checkout operation is wanted to be performed from one of the consumer projects, the trigger script is fired and disabled users from checking the element out with giving a warning to the user. The warning window is shown in Figure 4.20.



(a) SCM Structure before reconfiguration

(b) SCM Structure after reconfiguration

Figure 4.21: **(a)** SCM structure of the main-line projects before reconfiguration and **(b)** SCM structure of producer and consumer projects after reconfiguration

CHAPTER 5

CONCLUSION

From the software governance perspective, flexibility emerges as an important feature, because of the dependencies and dynamics between projects of companies. Software development activities should be adaptable to the changing software governance structures and in this thesis we have addressed the adaptation of software configuration management activities to the governance structures dynamically.

If the governance structure of a project changes during the development phase and SCM structure of a project is not adapted accordingly, project team(s) would face problems while performing crucial SCM activities such as integration, merge and release activities. Misuse of SCM patterns would be hardening software development activities instead of facilitating them.

When a single-line component or sub-component is required to be developed by multiple projects and Main-Line pattern is not implemented, options of other projects to develop the component in parallel will be:

- copying contents of shared component or sub-component to a new VOB which will be used by other project and developing that component apart from the original project
- adding component to their configurations directly and branching from the first project's integration stream

Both options have drawbacks in common activities of SCM. First option brings the problem of re-work when a new feature is added by one of the projects which is required by other projects as well. One of the problems that second option causes is new projects have to add redundant sub-components to its configuration if only a sub-component is required to be a main-line component. Another problem of second option raises when more than two projects wants to develop Simulators component in parallel. Project which will form the basis for triple integration and merge operations is imprecise. *Cascading* 2.3.2 project and release problem is another outcome of not using Main-Line pattern. On the other hand, our DSCM system solves these problems easily by reconfiguring SCM structure to Main-Line pattern because the existence of a main-line project eliminates the problems of re-work and ambiguity of integration&merge operations. In addition, none of the projects have to add or access redundant code because sub-components are transformed into components by our system.

Similarly, when a single-line component or sub-component is required to be reused by multiple projects and Producer-Consumer pattern is not implemented, options of other projects to reuse the component in parallel will be:

- copying contents of shared component or sub-component to a new VOB which will be used by other project
- adding component to their configurations directly and rebasing to the desired baseline of component from the first project's integration stream

Again, both options have drawbacks which are resolved by our DSCM system. The problem of re-work rises when first option is preferred over migration to Producer-Consumer pattern. One of the problems of selecting second option rises when the reusable software is a sub-component of a component. If sub-component is not transformed into a reusable component, all of the other projects will have to add the whole component encapsulating that sub-component which is redundant. Also, projects may access software parts which they should not access. By reconfiguring single-line components or sub-components to reusable components using our DSCM approach, these problems are eliminated because Producer-Consumer pattern helps re-usability of components and resolves the problems of redundancy and re-usability in an efficient way.

When a main-line component or sub-component is required to be reused by follow-on projects and developed by the main-line project and adaptation to Producer-Consumer pattern is not performed, follow-on projects may do undesired changes on the re-usable components and deliver to the main-line project. The probability of this undesired delivers problem will be very high if preventive measures are not taken. The triggers we defined and applied prevent follow-on projects from making uncontrolled changes to the reusable component(s).

These problems are only a few of the problems that will raise if SCM is not flexible enough to address the needs of the project governance. Efficiency of using appropriate SCM pattern with software governance structures were studied before, and using inappropriate SCM pattern will reduce the efficiency that can be obtained from software transactions. Our dynamic software configuration management approach solves these and other problems easily. First of all, whenever a shared or reusable component development is required, a proposed SCM model is ready to be implemented. Secondly, we automatized the process to migrate from one SCM structure to another which makes reconfigurations easier. We proposed a dynamic SCM model and developed a tool integrated with a commercial tool, IBM Rational ClearCase, showing that our model is applicable and compatible with SCM tools. Since we have developed a meta-model for a dynamic SCM system, our pattern reconfiguration tool can be utilized to be integrated with other SCM tools easily in future. To integrate our Dynamic SCM system with other SCM tools, the tools should meet the following requirements:

- SCM tool should support *component-based* development
- SCM tool should support *branching*
- SCM tool should provide an *Application Programming Interface (API)* or *Command Line Interface (CLI)*

If the above requirements are met by the SCM tools, our Dynamic SCM model can be applicable to adapt configuration management structures to the changing development governance structures.

Currently, the DSCM tool we designed takes the information of which sub-components or components will be transformed into main-line or reusable components as an input from the user. In other words, the information of changing structures and dependencies of components is given to our system by the users. The first step beyond our approach would be gathering these information from the configuration management system itself instead of asking from user. *Dependency Structure Matrices (DSM)* are designed for extracting dependency information from the code and for organizing the components of projects according to the dependencies extracted. Thus, DSMs can be helpful in identifying the dependency changes and component identification which can directly trigger our DSCM system. DSM approach can be integrated to our system in the future to extract inputs automatically from projects.

REFERENCES

[1] *Guide to the Software Engineering Body of Knowledge 2004 Version SWEBOK ®A project of the IEEE Computer Society Professional Practices Committee*, chapter 7 - Software Configuration Management. IEEE Press, 2004.

[2] IEEE Standard for Configuration Management in Systems and Software Engineering. *IEEE Std 828-2012 (Revision of IEEE Std 828-2005)*, pages 1 –71, 16 2012.

[3] André Van Der Hoek and Dennis Heimbigner and Alexander L. Wolf. Software Architecture, Configuration Management, and Configurable Distributed Systems: A Ménage a Trois. Technical report, 1998.

[4] Bannerman, Paul L. Software Development Governance: A Meta-Management Perspective. In *Proceedings of the 2009 ICSE Workshop on Software Development Governance*, SDG '09, pages 3–8, Washington, DC, USA, 2009. IEEE Computer Society.

[5] S. Berczuk and B. Appleton. *Software Configuration Management Patterns: Effective Teamwork and Practical Integration (Software Patterns Series)*. Addison-Wesley Longman, Amsterdam, illustrated edition edition, 2002.

[6] Bradbury, Jeremy S. and Cordy, James R. and Dingel, Juergen and Wermelinger, Michel. A Survey of Self-Management in Dynamic Software Architecture Specifications. In *Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, WOSS '04, pages 28–33, New York, NY, USA, 2004. ACM.

[7] Cheng, Tjan-Hien and Jansen, Slinger and Remmers, Marc. Controlling and Monitoring Agile Software Development in Three Dutch Product Software Companies. In *Proceedings of the 2009 ICSE Workshop on Software Development Governance*, SDG '09, pages 29–35, Washington, DC, USA, 2009. IEEE Computer Society.

[8] Chulani, Sunita and Williams, Clay and cYaeli, Avi. Software Development Governance and Its Concerns. In *Proceedings of the 1st international workshop on Software Development Governance*, SDG '08, pages 3–6, New York, NY, USA, 2008. ACM.

[9] S. Dart. Concepts in Configuration Management Systems. In *Proceedings of the 3rd International Workshop on Software Configuration Management*, SCM '91, pages 1–18, New York, NY, USA, 1991. ACM.

[10] S. Dart. *The Past, Present, and Future of Configuration Management*. Technical report. Carnegie Mellon University, Software Engineering Institute, 1992.

[11] C. Erbaş and B. c. Erbaş. Software Development under Bounded Rationality and Opportunism. In *Proceedings of the 2009 ICSE Workshop on Software Development Governance*, SDG '09, pages 15–20, Washington, DC, USA, 2009. IEEE Computer Society.

[12] J. Estublier, D. Leblang, A. v. d. Hoek, R. Conradi, G. Clemm, W. Tichy, and D. Wiborg-Weber. Impact of Software Engineering Research on the Practice of Software Configuration Management. *ACM Trans. Softw. Eng. Methodol.*, 14(4):383–430, Oct. 2005.

[13] J. Highsmith. *Agile Software Development Ecosystems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[14] IBM. IBM Rational ClearCase Help Page. `https://publib.boulder.ibm.com/infocenter/cchelp/v7r1m0/index.jsp?topic=/com.ibm.rational.clearcase.help.ic.doc/helpindex_clearcase.htm`.

[15] IBM. IBM Rational ClearCase Home Page. `http://www-01.ibm.com/software/awdtools/clearcase/index.html`.

[16] IBM. IBM Rational ClearCase Terminology Page. `http://www.ibm.com/developerworks/rational/library/08/0129_clearcase_glossary/index.html`.

[17] F. G. Kandemir. Automatic Transition between Software Configuration Management Patterns. In *Proceedings of the 2nd Symposium on Software Quality and Software Development Tools*, 2010.

[18] Kofman, Alexander and Yaeli, Avi and Klinger, Tim and Tarr, Peri. Roles, Rights, and Responsibilities: Better Governance Through Decision Rights Automation. In *Proceedings of the 2009 ICSE Workshop on Software Development Governance*, SDG '09, pages 9–14, Washington, DC, USA, 2009. IEEE Computer Society.

[19] Larsson, Magnus and Crnkovic, Ivica. Component Configuration Management. In *Proceedings of the ECOOP Conference, Workshop on Component Oriented Programming*, Nice, France, June 2000.

[20] Lehto, Ilkka and Rautiainen, Kristian. Software Development Governance Challenges of a Middle-Sized Company in Agile Transition. In *Proceedings of the 2009 ICSE Workshop on Software Development Governance*, SDG '09, pages 36–39, Washington, DC, USA, 2009. IEEE Computer Society.

[21] Magnus Larsson. *Applying Configuration Management Techniques to Component-Based Systems*. Licentiate thesis, Department of Information Technology, Uppsala University, Dec. 2000. Also published as report MRTC 00/24 at Mälardalens högskola.

[22] A. K. Midha. Software Configuration Management for the 21st Century. *Bell Labs Technical Journal*, 2:154–155, 1997.

[23] N. Pala Er and C. Erbaş. Aligning Software Configuration Management with Governance Structures. In *Proceedings of the 2010 ICSE Workshop on Software Development Governance*, SDG '10, pages 1–8, New York, NY, USA, 2010. ACM.

[24] N. Pala Er, C. Erbaş, and B. Ç. Erbaş. *Modern Software Engineering Concepts and Practices: Advanced Approaches*, chapter Software Development Governance: A Case Study for Tools Integration, pages 315–332. IGI Global, 2011.

[25] Szyperski, Clemens. Component Technology: What, Where, and How? In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 684–693, Washington, DC, USA, 2003. IEEE Computer Society.

[26] Tarr, Peri and Williams, Clay and Hailpern, Brent. Toward Governance of Emergent Processes and Adaptive Organizations. In *Proceedings of the 1st international workshop on Software development governance*, SDG '08, pages 21–24, New York, NY, USA, 2008. ACM.

[27] W. Tichy. Tools for Software Configuration Management. In *Proc. Int'l Workshop Software Version and Configuration Control*, pages 1–20. Teubner Verlag, 1988.

[28] W. Tichy. Software Configuration Management Overview. Technical report, 2008.

[29] S. Vance. Advanced SCM Branching Strategies. Technical report, Perforce Software, Inc., 1998.

[30] Walrad, Chuck and Strom, Darrel. The Importance of Branching Models in SCM. *Computer*, 35(9):31–38, Sept. 2002.

[31] Williams, Clay. Technical Creativity and the Challenge of "Governing" Software Development. In *Proceedings of the 1st international workshop on Software development governance*, SDG '08, pages 15–15, New York, NY, USA, 2008. ACM.

[32] L. Wingerd and C. Seiwald. High-level Best Practices in Software Configuration Management. In B. Magnusson, editor, *System Configuration Management*, volume 1439 of *Lecture Notes in Computer Science*, pages 57–66. Birkhauser Basel, 1998. 10.1007/BFb0053878.

APPENDIX A

IBM Rational ClearCase Terminology

The following list of terminologies are taken directly from IBM Rational ClearCase Glossary Page [16].

Activity

An object that tracks the work required to complete a development task. An activity includes a text headline, which describes the task, and a change set, which identifies all versions that developers create or modify while working on the activity.

Administrative VOB

A versioned object base (VOB) that contains global type objects. Local copies of global type objects can be created in any VOB that has an AdminVOB hyperlink to the administrative VOB that defines the global type object.

Artifact

An entity that is used or produced by a software development process. Examples of artifacts are models, source files, scripts, and binary executable files.

Baseline

An object that represents a stable configuration for one or more components. A baseline identifies activities and one version of every element that is visible in one or more components.

Branch

An object that specifies a linear sequence of versions of an element. Each branch is an instance of a branch type object. build

Checked-out version

A copy of a file that corresponds to a version of an element. See also Version.

Checkin

The action that creates a new version of an element on any branch of its version tree.

Component

A ClearCase object that is used to group a set of related directory and file elements within a Unified Change Management (UCM) project. Typically, the elements that make up a component are developed, integrated, and released together. A project must contain at least one component, and it can contain multiple components. Projects can share components.

Delivery operation

A ClearCase operation in which developers merge the work from their own development streams to the project's integration stream or to a feature-specific development stream. If required, the deliver operation invokes the Merge Manager to merge versions.

Development stream

An object that determines which versions of elements appear in a development view and maintains a list of a developer's activities. The development stream configures the development view to select the versions associated with the foundation baselines plus any activities and versions that developers create after they join the project or rebase their development stream.

Dynamic view

A view that uses a network file system to access versions of elements.

Element

An object that encompasses a set of versions, organized into a version tree.

Element type

A property of an element that specifies how versions of that element are constructed.

Foundation baseline

A baseline that configures a stream. Foundation baselines specify the versions and activities that appear in a view.

History

Metadata in a versioned object base (VOB) that consists of event records for objects in that VOB.

Label

An instance of a label type object, which provides a user-defined name for a version. See also object.

Main branch

The starting branch of a version tree of an element. The default name for this branch is main.

Metadata

Data that describes the characteristics of data; descriptive data.

Object

An item stored in a versioned object base (VOB). An object can be identified by an object-selector string, which includes a prefix that indicates the kind of object, the object's name, and a suffix that indicates the VOB in which the object resides. Examples: lbtype:REL1@/vobs/vega on UNIX and lbtype:REL1@\vega on Windows. See also label.

Project

An object that contains configuration information, activities, and policies required to manage a development effort.

Project VOB (PVOB)

A versioned object base (VOB) that stores Unified Change Management (UCM) objects, such as projects, streams, activities, and change sets.

PVOB

See Project VOB

Rebase

A ClearCase operation that makes a development work area current with the set of versions represented by a more recent baseline in another stream, usually the project's integration stream or a feature-specific development stream.

Snapshot view

A view that uses a local file system to access versions of elements.

Stream

An object that specifies configuration rules for a UCM view.

Trigger

A monitor that specifies one or more standard programs or built-in actions to be executed whenever a certain ClearCase operation is performed.

UCM

See Unified Change Management.

Unified Change Management

A process for organizing software development teams and their work products. Members of a project team use activities and components to organize their work.

Version

An object that implements a particular revision of an element. The versions of an element are organized into a version tree structure. See also checked-out version.

Versioned object base (VOB)

A repository that stores versions of file elements, directory elements, derived objects, and metadata associated with these objects.

Version tree

A graphic representation of a versioned object that shows all branches and the versions on each branch.

View

A ClearCase object that provides a work area for one or more users. For each element in a VOB, a view's configuration specification selects one version from the element's version tree.

VOB

See versioned object base.

VOB database

The part of a versioned object base (VOB) storage directory in which metadata and VOB objects are stored.

APPENDIX B

Summaries of SCM Pattern Reconfiguration Wizards

B.1 Summary of Single-Line to Main-Line Pattern Reconfiguration Wizard

```
AirlineShared_PVOB created successfully.
Created hyperlink ”AdminVOB@107@\AirlineShared_PVOB”.

AirlineShared_VOB created successfully.
Created hyperlink ”AdminVOB@42@\AirlineShared_VOB”.

Created component ”Simulators_Main”.

Created component ”SystemManagement_Main”.

Contents of M:\Gulsah_Airline_Prj_Int\Airliner_VOB\Test\Simulators
copied to a temporary location.
Contents of M:\Gulsah_Airline_Prj_Int\Airliner_VOB\Software\
SystemManagement copied to a temporary location.
Created project ”Simulators_Main_Prj”.
Changed modifiable component list for project ”Simulators_Main_Prj”.


Created stream ”Simulators_Main_Prj_Int”.

Selected Server Storage Location ”gkcomp_ccstg_c_views”.
Created view.
Host-local path: gkcomp:c:\ClearCase_Storage\views\GKCOMP\
Gulsah+Kandemir\Simulators_Main_Prj_Int.vws
Global path: \\gkcomp\ccstg_c\views\GKCOMP\Gulsah+Kandemir\
Simulators_Main_Prj_Int.vws
Attached view to stream ”Simulators_Main_Prj_Int”.

Created activity ”Adding_to_source_control_for_the_first_time”.
Set activity ”Adding_to_source_control_for_the_first_time” in view
”Simulators_Main_Prj_Int”.

Simulators is successfully imported.
Created baseline ”Simulators_Main_Prj_2_5_2012” in component
”Simulators_Main”.
Begin incrementally labeling baseline
”Simulators_Main_Prj_2_5_2012”.
Done incrementally labeling baseline
”Simulators_Main_Prj_2_5_2012”.


Changed stream ”Simulators_Main_Prj_Int@\AirlineShared_PVOB”.

Created project ”SystemManagement_Main_Prj”.
Changed modifiable component list for project ”SystemManagement_Main_Prj”.

Created stream ”SystemManagement_Main_Prj_Int”.

Selected Server Storage Location ”gkcomp_ccstg_c_views”.
Created view.
Host-local path: gkcomp:c:\ClearCase_Storage\views\GKCOMP\
Gulsah+Kandemir\SystemManagement_Main_Prj_Int.vws
Global path: \\gkcomp\ccstg_c\views\GKCOMP\Gulsah+Kandemir\
SystemManagement_Main_Prj_Int.vws
Attached view to stream ”SystemManagement_Main_Prj_Int”.

Created activity ”Adding_to_source_control_for_the_first_time.1734”.
Set activity ”Adding_to_source_control_for_the_first_time.1734”
in view ”SystemManagement_Main_Prj_Int”.

SystemManagement is successfully imported.
```

```
Created baseline ”SystemManagement_Main_Prj_2_5_2012”
in component ”SystemManagement_Main”.
Begin incrementally labeling baseline
”SystemManagement_Main_Prj_2_5_2012”.
Done incrementally labeling baseline
”SystemManagement_Main_Prj_2_5_2012”.


Changed stream ”SystemManagement_Main_Prj_Int@\AirlineShared_PVOB”.

AutoPilot_PVOB created successfully.
Created hyperlink ”AdminVOB@107@\AutoPilot_PVOB”.

Created project ”AutoPilot_Prj”.
Changed modifiable component list for project ”AutoPilot_Prj”.

Created stream ”AutoPilot_Prj_Int”.

Changed modifiable component list for project ”Airline_Prj”.
Propagating changes to the integration stream ”Airline_Prj_Integration”.
Changes to modifiability of components must be propagated to streams
and their views. Use ’cleartool chstream -generate’ to update the stream with
’cleartool setcs -stream’ for each view attached to the stream or see the stream
and view properties pages.
Changed modifiable component list and propagated changes to streams
in project ”Airline_Prj@\Airliner_PVOB”.

Adding baseline ”Simulators_Main_Prj_2_5_2012” of new component ”Simulators_Main”
Updating rebase view’s config spec...
Creating integration activity...
Setting integration activity...
Merging files...
No versions require merging in stream ”Airline_Prj_Integration”.
Build and test are necessary to ensure that any merges and configuration changes were completed correctly.
When build and test are confirmed, run ”cleartool rebase -complete”.
```

```
Rebase in progress on stream ”Airline_Prj_Integration”.
Started by ”Gulsah Kandemir” at 2/5/2012 7:04:37 PM.
Merging files...

No versions require merging in stream ”Airline_Prj_Integration”.
Checking in files...

Clearing integration activity...

Updating stream’s configuration...

Cleaning up...

Rebase completed.

Adding baseline ”SystemManagement_Main_Prj_2_5_2012” of new component
”SystemManagement_Main”.
Updating rebase view’s config spec...
Creating integration activity...
Setting integration activity...
Merging files...
No versions require merging in stream ”Airline_Prj_Integration”.
Build and test are necessary to ensure that any merges and configuration changes were completed correctly.
When build and test are confirmed, run ”cleartool rebase -complete”.


Rebase in progress on stream ”Airline_Prj_Integration”.
Started by ”Gulsah Kandemir” at 2/5/2012 7:04:40 PM.
Merging files...
No versions require merging in stream ”Airline_Prj_Integration”.
Checking in files...
Clearing integration activity...
Updating stream’s configuration...
Cleaning up...
Rebase completed.
```

Changed stream ”Airline_Prj_Integration@\Airliner_PVOB”.

Changed stream ”AutoPilot_Prj_Int@\AutoPilot_PVOB”.

B.2 Summary of Single-Line to Producer-Consumer Pattern Reconfiguration Wizard

DriversVOB created successfully. Created hyperlink ”AdminVOB@42@\DriversVOB”.

Created component ”Drivers_Producer”.

Contents of M:\Gulsah_Airline_Prj_Int\Airliner_VOB\Drivers copied to a temporary location. Created project ”Drivers_Producer_Prj”. Changed modifiable component list for project ”Drivers_Producer_Prj”.

Created stream ”Drivers_Producer_Prj_Int”.

Selected Server Storage Location ”gkcomp_ccstg_c_views”.
Created view.
Host-local path: gkcomp:c:\ClearCase_Storage\views\GKCOMP\Gulsah+Kandemir\
Drivers_Producer_Prj_Int.vws
Global path: \\gkcomp\ccstg_c\views\GKCOMP\Gulsah+Kandemir\
Drivers_Producer_Prj_Int.vws
Attached view to stream ”Drivers_Producer_Prj_Int”.

Created activity ”Adding_to_source_control_for_the_first_time.1319”.
Set activity ”Adding_to_source_control_for_the_first_time.1319” in view
”Drivers_Producer_Prj_Int”.

Drivers is successfully imported.
Created baseline ”Drivers_Producer_Prj_2_6_2012” in component ”Drivers_Producer”.
Begin incrementally labeling baseline ”Drivers_Producer_Prj_2_6_2012”.
Done incrementally labeling baseline ”Drivers_Producer_Prj_2_6_2012”.

```
Changed stream ”Drivers_Producer_Prj_Int@\AirlineShared_PVOB”.

HL_Design_PVOB created successfully.
Created hyperlink ”AdminVOB@107@\HL_Design_PVOB”.

Created project ”HLDesign_Prj”.
Changed modifiable component list for project ”HLDesign_Prj”.

Changed modifiable component list for project ”HLDesign_Prj”.
Changed modifiable component list and propagated changes to streams in project ”HLDesign_Prj@\HL_Design_PVOB”.

Created stream ”HLDesign_Prj_Int”.

Adding baseline ”Drivers_Producer_Prj_2_6_2012” of new component ”Drivers_Producer”
Updating rebase view’s config spec...
Creating integration activity...
Setting integration activity...
Merging files...
No versions require merging in stream ”Airline_Prj_Integration”.
Build and test are necessary to ensure that any merges and configuration changes were completed correctly.
When build and test are confirmed, run ”cleartool rebase -complete”.


Rebase in progress on stream ”Airline_Prj_Integration”.
Started by ”Gulsah Kandemir” at 2/6/2012 11:40:14 PM.
Merging files...
No versions require merging in stream ”Airline_Prj_Integration”.
Checking in files...
Clearing integration activity...
Updating stream’s configuration...
Cleaning up...
Rebase completed.
```

Changed stream ”Airline_Prj_Integration@\Airliner_PVOB”.

Changed stream ”HLDesign_Prj_Int@\HL_Design_PVOB”.

B.3 Summary of Main-Line to Producer-Consumer Pattern Reconfiguration Wizard

Created trigger type ”chk_prj_Simulators_Main”.

Added trigger ”chk_prj_Simulators_Main” to inheritance list of ”M:\Simulators_Main_Prj_Int\AirlineShared_VOB\Simulators_Main”.
Added trigger ”chk_prj_Simulators_Main” to attached list of ”M:\Simulators_Main_Prj_Int\AirlineShared_VOB\Simulators_Main”.
Added trigger ”chk_prj_Simulators_Main” to inheritance list of ”M:\Simulators_Main_Prj_Int\AirlineShared_VOB\Simulators_Main\Simulators”.
Added trigger ”chk_prj_Simulators_Main” to attached list of ”M:\Simulators_Main_Prj_Int\AirlineShared_VOB\Simulators_Main\Simulators”.
Added trigger ”chk_prj_Simulators_Main” to inheritance list of ”M:\Simulators_Main_Prj_Int\AirlineShared_VOB\Simulators_Main\Simulators\sim1”.
Added trigger ”chk_prj_Simulators_Main” to attached list of ”M:\Simulators_Main_Prj_Int\AirlineShared_VOB\Simulators_Main\Simulators\sim1”.
Added trigger ”chk_prj_Simulators_Main” to inheritance list of ”M:\Simulators_Main_Prj_Int\AirlineShared_VOB\Simulators_Main\Simulators\sim2”.
Added trigger ”chk_prj_Simulators_Main” to attached list of ”M:\Simulators_Main_Prj_Int\AirlineShared_VOB\Simulators_Main\Simulators\sim2”.